9

Inserting, Deleting, and Updating Records

After you've established a connection to the database (discussed in Chapter 7, ªConnecting to the Databaseº), you can interact with it. This chapter shows you how to insert, delete, and update database records in three different ways: When you're working with a data source, when you're working primarily at the database level, and when you're working primarily at the adaptor level.

This chapter is organized into the following sections:

· ªWorking with a Data Sourceº shows you how to insert, delete, and update database records with a data source.

· ªWorking at the Database Levelº shows you how to insert, delete, and update database records at the database level.

· ªWorking at the Adaptor Levelº shows you how to insert, delete, and update database records at the adaptor level.

Note that this chapter doesn't discuss how to retrieve records from the database. Fetching is the subject of Chapter 8, ªRetrieving Records.º In order to get the most out of this chapter, you should first read those sections in Chapter 8 that are appropriate for your application.

Working with a Data Source

A data source is an object that presents a standard interface to a store of enterprise objects. The data source class supplied with the Framework, EODatabaseDataSource, provides an interface to the Framework's access layer and, ultimately, to a relational database. Regardless of how the underlying data is stored, however, all data sources have the ability to fetch, insert, update, and delete objects.

For most database applications, the data source is an instance of the EODatabaseDataSource class. However, the data source can be any object that conforms to the EODataSources protocol. (Because EODatabaseDataSource is the only data source supplied with the Framework, however, the procedures in this and the following sections assume that you are working with an EODatabaseDataSource.)

Programs that use a data source to operate on data in the database rely heavily on the following methods, which are documented in the EODataSources protocol specification in the *Enterprise Objects Framework Reference*:

```
createObject
insertObject:
```

updateObject:
deleteObject:
saveObjects

Methods for constructing and working with database data source objects are documented in the EODatabaseDataSource class specification in the *Enterprise Objects Framework Reference*.

By default, all EODatabaseDataSources begin transactions automatically upon the first insert, delete, or update (if needed). You can verify this for a particular data source by sending it a **beginsTransactionsAutomatically** message and checking the returned value. If your data source doesn't automatically start transactions for you, your application needs to drop down to the database level and control transactions manually. For more information on this technique, see Chapter 10, ªWorking Across Levels.º

Even though EODatabaseDataSources begin transactions automatically, you still have to indicate when the transaction has completed. You do this by sending **saveObjects** to the data source. If your data source conforms to the EORollbackDataSources protocol (as does EODatabaseDataSource), until you send **saveObjects** you can issue a **rollback** message to abort the transaction and undo those changes to the database that were made during the current transaction.

The following three sections (ªInserting Objects with a Data Source,º ªUpdating Objects with a Data Source,º and ªDeleting Objects with a Data Sourceº) illustrate how you operate on database data using a data source. In those sections, all examples work with enterprise objects of the Department class; this class provides **setDepartmentName:**, **setDeptID:**, and **setLocationID:** accessor methods. All examples also assume that you have write access to the database, as specified in the model.

Even if you have write access to the database, individual properties for an entity may be read-only. When inserting and updating objects, attributes that are marked read-only in the model aren't modified, and your code isn't notified if you attempt to insert or update a record with read-only attributes.

If you attempt to insert or update a record that contains a NULL value for an attribute that is required by the server to be non-NULL, the adaptor generates an error and the operation fails.

Inserting Objects with a Data Source

The basic procedure to use when inserting objects with a data source is:

1. Use **createObject** to generate the object to be inserted. Initialize the object appropriately, making sure that you assign it a primary key.
2. Insert the object into the database with **insertObject:**.
3. Complete the transaction with **saveObjects**.

Here's one way to insert a record into the database, given a data source that handles Department enterprise objects:

```
EODatabaseDataSource *myDataSource;    /* Assume this exists. */
Department *myObject;
BOOL ok;

/* Construct the object. */
myObject = [myDataSource createObject];
[myObject setDepartmentName:@"Publications"];
[myObject setDeptID:700];
[myObject setLocationID:1101];

/* Insert it into the database. */
ok = [myDataSource insertObject:myObject];
```

```
if (!ok) {
    /* Object wasn't inserted. Handle the error. */
}

/* Make the change permanent. */
ok = [myDataSource saveObjects];

if (!ok) {
    /* Changes weren't made. Handle the error. */
}
```

Note the use of **createObject** to create the empty enterprise object that's going to be inserted into the database. Any object that's to be added to the database using a data source must have been created using **createObject**. This means that in order to insert the contents of an existing enterprise object, created with **alloc**, into the database, you must create an empty object with **createObject** and copy the contents of your existing object into it. You can then insert the new object into the database.

However your code constructs the enterprise object to be inserted, the object must have a primary key and the key must be unique within the database.

Updating Objects with a Data Source

The basic procedure to use when updating objects with a data source is:

1. Change the object's values (you can even change the object's primary key).
2. Replace the object in the database with **updateObject:**.
3. Complete the transaction with **saveObjects**.

Here's one way to alter the contents of a record in the database, given a data source that handles Department enterprise objects:

```
EODatabaseDataSource *myDataSource;    /* Assume this exists. */
Department *myObject;                  /* Assume this exists. */
BOOL ok;

/* Change the value(s). */
[myObject setLocationID:1102];

/* Send the object back. */
ok = [myDataSource updateObject:myObject];

if (!ok) {
    /* Object wasn't updated. Handle the error. */
}

/* Make the change permanent. */
ok = [myDataSource saveObjects];

if (!ok) {
    /* Changes weren't made. Handle the error. */
}
```

When working with a data source, you must fetch an object from the database before you update it. Also, an EODatabaseDataSource won't let you update objects if it was initialized with a read-only entity.

Deleting Objects with a Data Source

The basic procedure to use when deleting objects with a data source is:

1. Remove the object from the database with **deleteObject:**.
2. Commit the transaction with **saveObjects**.

Here's one way to remove a record from the database, given a data source that handles Department enterprise objects:

```
EODatabaseDataSource *myDataSource;     /* Assume this exists. */
Department *myObject;                   /* Assume this exists. */
BOOL ok;

/* Delete the object. */
ok = [myDataSource deleteObject:myObject];

if (!ok) {
    /* Object wasn't deleted. Handle the error. */
}

/* Make the change permanent. */
ok = [myDataSource saveObjects];

if (!ok) {
    /* Changes weren't made. Handle the error. */
}
```

An EODatabaseDataSource won't let you delete objects if it was initialized with a read-only entity. To find out if your data source will let you delete objects, send it a **canDelete** message. For more information, see the EODataSources protocol specification in the *Enterprise Objects Framework Reference*.

Working at the Database Level

The database level deals with data packaged as enterprise objects. At the database level, you can create multiple database connections, contexts, and channels. The database level also allows you to exercise a great deal of control over the way that objects are created and used by your application.

Programs that operate on data in the database while working primarily at the database level rely heavily on the following methods, which are documented in the EODatabaseChannel class specification in the *Enterprise Objects Framework Reference*:

insertObject:
lockObject
updateObject:
deleteObject:

All of these operations notify the database channel's delegate before and after the operation.

Before you can operate on data in the database, you must have a transaction in progress. At the database level, you start a transaction by issuing **beginTransaction** to the database context linked to the channel you're using. To complete a transaction, send either **commitTransaction** or **rollbackTransaction**, as appropriate.

The following three sections (ªInserting Objects at the Database Level,º ªUpdating Objects at the Database Level,º and ªDeleting Objects at the Database Levelº) illustrate how you insert, update, and delete database records using database-level methods. All examples work with enterprise objects of the Department class; this class provides all of the appropriate accessor methods. The examples also assume that you have write access to the database, as specified in the model.

Even if you have write access to the database, individual attributes for an entity may be read-only.

When inserting and updating objects, attributes that are marked read-only in the model aren't modified, and your code isn't notified if you attempt to insert or update a record with read-only attributes.

If you attempt to insert or update a record that contains a NULL value for an attribute that is required by the server to be non-NULL, the adaptor generates an error and the operation fails.

Inserting Objects at the Database Level

The basic procedure to use when inserting objects using database-level methods is:

1. Construct the object to be inserted.
2. Begin a transaction.
3. Insert the object into the database with **insertObject:**.
4. Complete the transaction.

Here's one way to insert an object into the database, given an open database channel:

```
EODatabaseChannel *databaseChannel;    /* Assume this exists. */
EODatabaseContext *databaseContext;
Department *myObject;
BOOL ok;

databaseContext = [databaseChannel databaseContext];

/* Construct the object. */
myObject = [[Department alloc] init];
[myObject setDepartmentName:@"Publications"];
[myObject setDeptID:700];
[myObject setLocationID:1101];

[databaseContext beginTransaction];

ok = [databaseChannel insertObject:myObject];

if (!ok) {
    /* Object wasn't inserted. */
    [databaseContext rollbackTransaction];
    /* Handle error. */
} else {
    [databaseContext commitTransaction];
}
```

The above example shows one way to construct the enterprise object to be inserted. However your code constructs it, the object must have a primary key and the key must be unique within the database.

Note that **myObject**'s corresponding entity wasn't explicitly identified anywhere in the above example. To determine this information, the Enterprise Objects Framework compares the name of your enterprise object's class (ªDepartment,º in this example) with the class names specified for the entities in the model associated with your channel. When it finds a match, it then knows which entity the object corresponds to, and thus which database table the object is to be inserted into.

Updating Objects at the Database Level

The basic procedure to use when updating objects using database-level methods is:

1. Begin a transaction.

2. If necessary, lock the object so that it won't be changed by another process while your code is attempting to change it. This lock remains in effect until the next **commitTransaction** or **rollbackTransaction**. See Chapter 2, ªFrom Database to Objectsº for a discussion of update strategies and locking.

3. Change the object's values (because the object is uniquely identified at the time it's fetched, you can change the object's primary key).

4. Update the object in the database with **updateObject:**.

5. Complete the transaction.

Here's one way to alter the contents of a record in the database, given an open database channel:

```
EODatabaseChannel *databaseChannel;    /* Assume this exists. */
Department *myObject;                  /* Assume this exists. */
EODatabaseContext *databaseContext;
BOOL ok;

databaseContext = [databaseChannel databaseContext];

[databaseContext beginTransaction];

/* If snapshots are disabled, you must fetch the object here, */
/* within the transaction. */

/* If necessary, lock the object so no one else can change it. */
[databaseChannel lockObject:myObject];

/* Change the value(s). */
[myObject setLocationID:1102];

/* Send the changes to the database. */
ok = [databaseChannel updateObject:myObject];

if (!ok) {
    /* Object wasn't updated. */
    [databaseContext rollbackTransaction];
    /* Handle error. */
} else {
    [databaseContext commitTransaction];
}
```

You must fetch the object from the database before you can update it. When you fetch an object, a snapshot is taken (thus allowing you to change any aspect of the object, including its primary key). You don't have to lock the object if there's no chance that the object will be modified by some other process during the transaction. If you don't lock the object and the Framework detects a conflict when you try to update, the update will fail.

You can exercise a degree of control over how your snapshots are taken as well as how and when locking is performed. For more information, see Chapter 2, ªFrom Database to Objects,º or the EODatabaseContext class specification in the *Enterprise Objects Framework Reference*.

Deleting Objects at the Database Level

The basic procedure to use when deleting objects using database-level methods is:

1. Begin a transaction.

2. Delete the object with **deleteObject:**.

3. Complete the transaction.

Here's one way to remove a record from the database, given an open database channel:

```
EODatabaseChannel *databaseChannel;     /* Assume this exists. */
Department *myObject;                   /* Assume this exists. */
EODatabaseContext *databaseContext;
BOOL ok;

databaseContext = [databaseChannel databaseContext];

[databaseContext beginTransaction];

/* Delete the object. */
ok = [databaseChannel deleteObject:myObject];

if (!ok) {
    /* Object wasn't deleted. */
    [databaseContext rollbackTransaction];
    /* Handle error. */
} else {
    [databaseContext commitTransaction];
}
```

You must fetch the object from the database before you can delete it.

Working at the Adaptor Level

While the database level deals with data packaged as enterprise objects, the adaptor level deals with database rows packaged as dictionaries. The adaptor level classes define a server-independent interface for working with relational database systems. Server-specific subclasses encapsulate the behavior of database servers, offering a uniform way of interacting with servers while still allowing applications to exploit their unique features.

Programs that operate on data in the database while working primarily at the adaptor level rely heavily on the following methods, which are documented in the EOAdaptorChannel class specification in the *Enterprise Objects Framework Reference*:

selectAttributes:describedByQualifier:fetchOrder:lock:
insertRow:forEntity:
updateRow:describedByQualifier:
deleteRowsDescribedByQualifier:

All of these operations notify the adaptor channel's delegate before and after the operation.

Before you can operate on data in the database, you must have a transaction in progress. At the adaptor level, you start a transaction by sending a **beginTransaction** message to the adaptor context. To complete a transaction, send either **commitTransaction** or **rollbackTransaction**, as appropriate.

The following three sections (ªInserting Rows at the Adaptor Level,º ªUpdating Rows at the Adaptor Level,º and ªDeleting Rows at the Adaptor Levelº) illustrate how you insert, update, and delete database records using adaptor-level methods. All examples assume that you have write access to the database, as specified in the model.

Even if you have write access to the database, individual attributes for an entity may be read-only. When inserting and updating rows, attributes that are marked read-only in the model aren't modified, and your code isn't notified if you attempt to insert or update a record with read-only attributes.

If you attempt to insert or update a record that contains a NULL value for an attribute that is required by the server to be non-NULL, the adaptor generates an error and the operation fails.

Inserting Rows at the Adaptor Level

The basic procedure to use when adding records using adaptor-level methods is:

1. Construct the dictionary object for the record to be inserted.
2. Begin a transaction
3. Insert the row into the database with **insertRow:forEntity:**.
4. Complete the transaction.

Here's one way to insert a record into the database, given an open adaptor channel:

```
EOAdaptorChannel *adaptorChannel;    /* Assume this exists. */
EOModel *model;                      /* Assume this exists. */
EOAdaptorContext *adaptorContext;
EOEntity *myEntity;
NSMutableDictionary *myRow;
BOOL ok;

adaptorContext = [adaptorChannel adaptorContext];
myEntity = [model entityNamed:@"Department"];

/* Construct the row. */
myRow = [NSMutableDictionary dictionary];
[myRow setObject:@"Publications" forKey:@"departmentName"];
[myRow setObject:[NSNumber numberWithInt:700] forKey:@"deptID"];
[myRow setObject:[NSNumber numberWithInt:1101] forKey:@"locationID"];

[adaptorContext beginTransaction];

ok = [adaptorChannel insertRow:myRow forEntity:myEntity];

if (!ok) {
    /* Row wasn't inserted. */
    [adaptorContext rollbackTransaction];
    /* Handle error. */
} else {
    [adaptorContext commitTransaction];
}
```

The above example shows one way to construct the dictionary object to be inserted. However your code constructs it, the row must have a unique primary key.

Updating Rows at the Adaptor Level

The basic procedure to use when updating rows using adaptor-level methods is:

1. Begin a transaction.
2. Construct a qualifier that uniquely identifies the row to be updated.
3. Fetch the row (locking it in your **selectAttributes:¼** message) and change its values, or construct a new version of the row (because the row being updated is identified by the qualifier constructed in the previous step, you can change the row's primary key).
4. Update the row in the database with **updateRow:describedByQualifier:**.
5. Complete the transaction.

Here's one way to alter a record in the database, given an open adaptor channel:

```
EOAdaptorChannel *adaptorChannel;    /* Assume this exists. */
```

```
EOModel *model;                     /* Assume this exists. */
EOAdaptorContext *adaptorContext;
EOEntity *myEntity;
NSArray *myAttributes;
EOQualifier *myQualifier;
NSMutableDictionary *myUpdatedRow;
BOOL ok;

adaptorContext = [adaptorChannel adaptorContext];
myEntity = [model entityNamed:@"Department"];
myAttributes = [myEntity attributes];

[adaptorContext beginTransaction];

/* Fetch the row. */
myQualifier = [[EOQualifier alloc] initWithEntity:myEntity
    qualifierFormat:@"deptID = 700"];
ok = [adaptorChannel selectAttributes:myAttributes
    describedByQualifier:myQualifier fetchOrder:nil lock:YES];
if (ok) {
    myUpdatedRow = [adaptorChannel fetchAttributes:myAttributes
        withZone:NULL];
    [adaptorChannel cancelFetch];

   /* Change the value(s). */
    [myUpdatedRow setObject:[NSNumber numberWithInt:1102]
        forKey:@"locationID"];

   /* Send the changes to the database. */
    ok = [adaptorChannel updateRow:myUpdatedRow
        describedByQualifier:myQualifier];
}

if (!ok) {
    /* Row wasn't updated. */
    [adaptorContext rollbackTransaction];
    /* Handle error. */
} else {
    [adaptorContext commitTransaction];
}
```

In the above example, a department record is fetched from the database, the department's location ID is changed, and the record is sent back to the database. When the record is fetched, the argument for the **lock:** keyword of the **selectAttributes:¼** message is set to YES; this prevents other processes from changing the record before this code can write the updated record back to the database. Note that the lock actually remains in effect until the transaction is completed (at the next **rollbackTransaction** or **commitTransaction** message).

Although you typically fetch a record and then update it, as illustrated in the above example, the fetch isn't required. You can instead construct a dictionary object with the new values for the record, and use this object to update the record. You only have to specify the values that need to be changed. So, for example, you can replace the code that fetches the row and changes the location ID in the above example with the following, and the outcome is the same:

```
/* Identify the row. */
myQualifier = [[EOQualifier alloc] initWithEntity:myEntity
    qualifierFormat:@"deptID = 700"];

/* Construct a dictionary that identifies the value to be changed. */
myUpdatedRow = [NSDictionary dictionary];
[myUpdatedRow setObject:[NSNumber numberWithInt:1102]
    forKey:@"locationID"];
```

When using **updateRow:describedByQualifier:**, the record being updated must exist in the database. Also, the qualifier you supply must resolve to a single record only. If either of these conditions isn't true, the adaptor generates an error and the update fails.

Deleting Rows at the Adaptor Level

The basic procedure to use when deleting rows using adaptor level methods is:

1. Begin a transaction.
2. Construct a qualifier that uniquely identifies the rows to be deleted.
3. Remove the rows from the database with **deleteRowsDescribedByQualifier:**.
4. Complete the transaction.

Here's one way to remove records from the database, given an open adaptor channel:

```
EOAdaptorChannel *adaptorChannel;    /* Assume this exists. */
EOModel *model;                      /* Assume this exists. */
EOAdaptorContext *adaptorContext;
EOEntity *myEntity;
NSArray *myAttributes;
EOQualifier *myQualifier;
BOOL ok;

adaptorContext = [adaptorChannel adaptorContext];
myEntity = [model entityNamed:@"Department"];
myAttributes = [myEntity attributes];

[adaptorContext beginTransaction];

/* Delete the row. */
myQualifier = [[EOQualifier alloc] initWithEntity:myEntity
    qualifierFormat:@"deptID = 700"];
ok = [adaptorChannel deleteRowsDescribedByQualifier:myQualifier];

if (!ok) {
    /* Row wasn't deleted. */
    [adaptorContext rollbackTransaction];
    /* Handle error. */
} else {
    [adaptorContext commitTransaction];
}
```

Unlike the insert and update methods, **deleteRowsDescribedByQualifier:** operates on more than one row. Thus, you can delete an entire set of database records simply by constructing a qualifier that identifies the records in the set.