Creating a Java Client WebObjects Application

Apple Developer's Library

Apple and the publishers have tried to make the information contained in this manual as accurate and reliable as possible, but assume no responsibility for errors or omissions. They disclaim any warranty of any kind, whether express or implied, as to any matter whatsoever relating to this manual, including without limitation the merchantability or fitness for any particular purpose. In no event shall they be liable for any indirect, special, incidental, or consequential damages arising out of purchase or use of this manual or the information contained herein. Apple will from time to time revise the software described in this manual and reserves the right to make such changes without obligation to notify the purchaser.

Copyright © 1998 by Apple Computer, Inc., 1 Infinite Loop, Cupertino, CA 95014.
All rights reserved.
[7040.00]

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher or copyright owner. Printed in the United States of America. Published simultaneously in Canada.

NeXT, the NeXT logo, OPENSTEP, Enterprise Objects, Enterprise Objects Framework, Objective-C, WEBSCRIPT, and WEBOBJECTS are trademarks of NeXT Software, Inc. Apple is a trademark of Apple Computer, Inc., registered in the United States and othe countries. PostScript is a registered trademark of Adobe Systems, Incorporated. Windows NT is a trademark of Microsoft Corporation. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited. ORACLE is a registered trademark of Oracle Corporation, Inc. SYBASE is a registered trademark of Sybase, Inc. Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. All other trademarks mentioned belong to their respective owners.

Restricted Rights Legend: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 [or, if applicable, similar clauses at FAR 52.227-19 or NASA FAR Supp. 52.227-86].

This manual describes the Java Client feature of WebObjects 4.0.

Written by Terry Donoghue and Katie McCormick
With help from Bruce Arthur, Eric Noyau, Patrick Gates, Andreas Wendker, and Ray Kiddy
Technical illustrations by Karin Stroud
Production by Terri FitzMaurice
Art, production, and editorial management by Gary Miller
Technical publications management by Ron Hayden

Table of Contents

Chapter 1

Overview of Java Client

The Java Client feature of WebObjects distributes the objects of an Enterprise Objects Framework application between an application server and one or more clients—typically Java™ applications or Web browsers. It is based on a distributed client-server architecture that uses Java for its client-side objects. This architecture is multi-tier in that processing duties are divided among a client, an application server, and a database server. With a Java Client application, you can partition the business logic and data associated with enterprise objects into a client side and a server side. This partitioning can improve performance and at the same time help to secure legacy data and business rules.

Advantages of Java

To understand the difference that Java makes in client-server architectures, it helps first to consider two of the more common types of client-server applications: the traditional desktop application and the Web application. Rated on a set of desirable characteristics, each class of application has complementary strengths and weaknesses.

Characteristic	Desktop	Web
Interactive	Yes	No
Flexible Controls	Yes	No
Rich User-interface Paradigm	Yes	No
Portable	No	Yes
Easy to Administer	No	Yes
Accessible	No	Yes
Secure	No	Yes

Desktop applications can typically draw upon user-interface frameworks that provide a varied and flexible set of controls, modal dialogs, and multiple windows. On the other hand, HTML has a limited and static set of controls, mainly forms, active images, and hyperlinks.

A Web application is, by definition, portable since it can run on any client browser that implements certain standards and protocols, regardless of the underlying system; desktop applications are usually limited to the platforms

they were built for. Web applications also have high marks for accessibility because they are designed to make it easy for users to get data on networks. Finally, because sensitive data and business logic is confined to the server in Web applications, they tend to be more secure.

Java scores high on each of these characteristics because it can have a strong presence on each side of the client-server divide.The principal advantage of Java is that it runs almost anywhere. The client need only have a compatible Java virtual machine (VM), something that most operating systems and browsers now include as a standard feature. A Java application can run on the server or can be downloaded to the client as an applet. SunSoft's AWT and JFC packages provide a rich source of flexible, interactive controls for developers.

Thus the promise of Java is the best of both worlds ("promise" because the potential of Java is still being realized). So what are some distributed multi-tier Java-based architectures popular today?

Client JDBC applications use a "fat client" architecture. Custom code invokes JDBC on the client, which in turn goes through a driver to communicate with a JDBC proxy on the server; this proxy makes the necessary client-library calls on the server. The shortcomings of this type of architecture are typical of all fat-client architectures. Security is a problem because the bytecodes on the client are easily decompiled, leaving both sensitive data and business rules at risk. The server has to be open to allow all client operations without being able to control what the client is doing. In addition, such an architecture doesn't scale; it is expensive to move data over the channel to the client.

A JDBC Three-tier application (with CORBA as the transport) is a big improvement over Client JDBC. In this architecture the client can be thin since all that is required on the client side is the JFC, non-sensitive custom code (usually for managing the user interface), and CORBA stubs for communicating with the server. Sensitive business logic as well as logic related to database connection are stored on the server. In addition, the server handles all data-intensive computations.

Although JDBC Three-tier is an improvement over Client JDBC, it has its own weaknesses. First it results in too much network traffic. Because this architecture uses "proxy" business objects on the client as handles to the real objects on the server, each client request for an attribute is forwarded to the server, causing a separate round trip and precipitating a "message storm." Second, JDBC Three-tier requires developers to write much of the code

themselves, from code for database access and data packaging to code for user-interface synchronization and change tracking. Finally, JDBC Three-tier does not provide much of the functionality associated with application servers, such as application monitoring and load balancing, nor does it provide HTML integration.

Java Client Architecture

A Java Client application is essentially an Enterprise Objects Framework application distributed across an application server (running a WebObjects application) and one or more Web-browser clients (running applets). As a starting point, consider the following diagram, which depicts a "traditional" desktop Enterprise Objects Framework application.

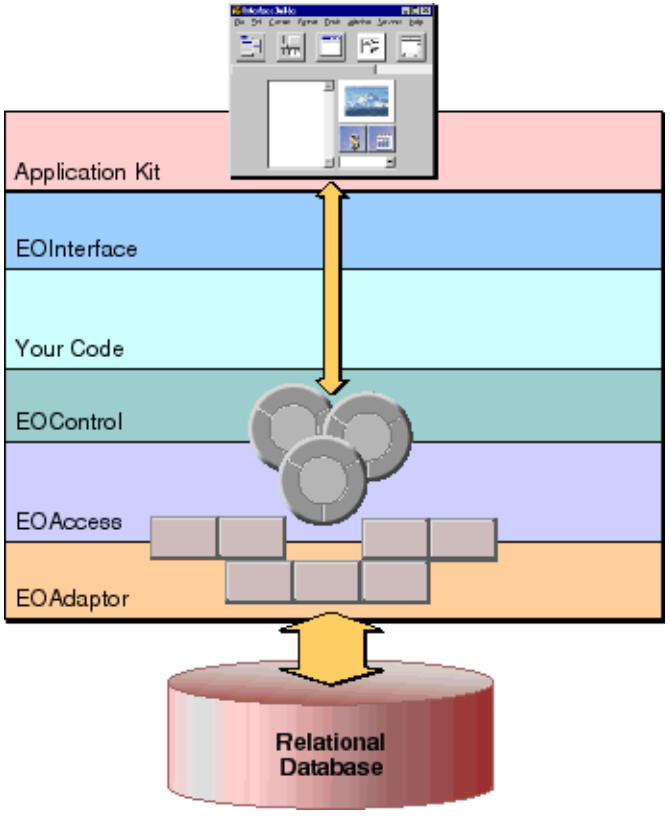


Figure 1. Architecture of Traditional Enterprise Objects Framework Application

In this architecture data is fetched from databases through the EOAdaptor layer, objects of which (adaptors) interact with specific database servers. The EOAccess layer creates enterprise objects from the "raw" fetched data and registers these with the EOControl layer; the access layer, through EOModel and related classes, also provides a mapping between the database schema and enterprise objects. The EOControl layer manages a graph of enterprise objects, tracks changes to them, and directs the access layer to commit changes to those objects. Finally, the EOInterface layer in this traditional desktop application synchronizes the data displayed in the user interface—here objects of the Application Kit framework—with the EOControl layer's graph of enterprise objects.

The design of Java Client breaks up some of these layers and redistributes them across the client and the application server, which occupies the middle tier in the overall architecture. The following figure illustrates how this is done.

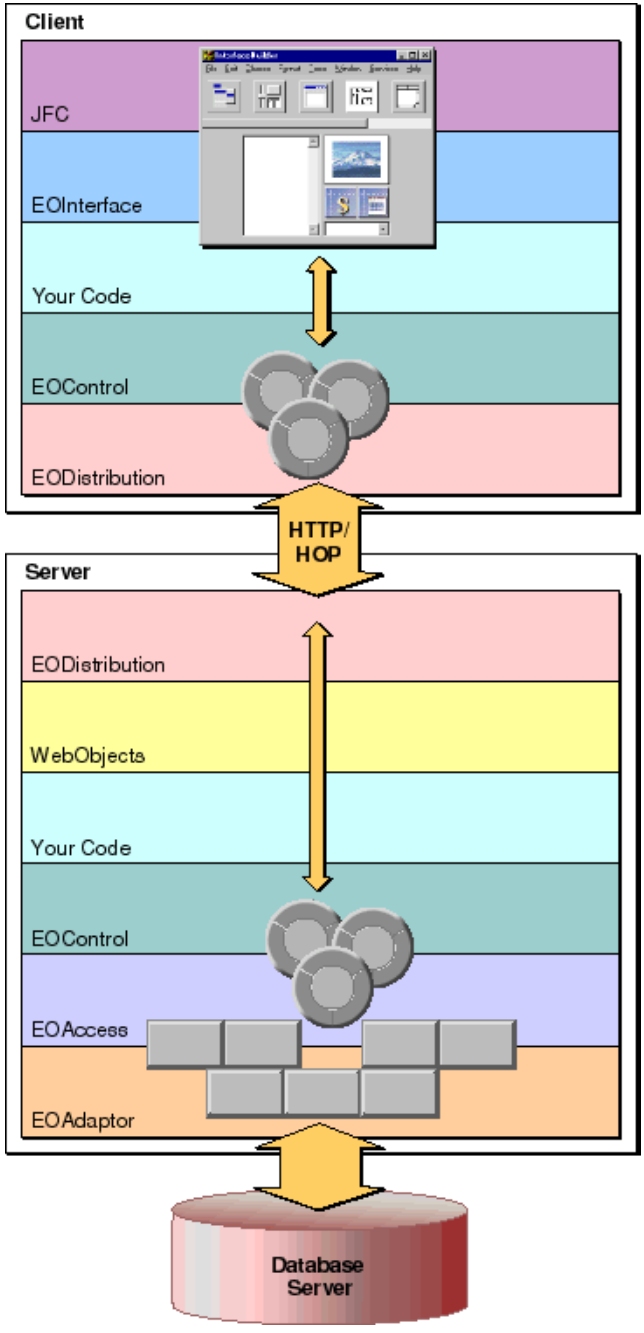


Figure 2. Architecture of Java Client

Java Client moves the pieces that perform object-to-UI-mapping to the client and duplicates the control layer on the client so that the graph of enterprise objects and the management of that graph occurs on both server and client. It also adds a new layer to both client and server, the EODistribution layer, which performs by-copy object distribution and synchronization. The final difference, of course, is the use of the Java Foundation Classes as the user-interface framework. (The object-to-database mapping layer, EOAccess, remains solely on the server.)

The diagram in Figure 2 seems to suggest a lot of complexity, but it is important to keep in mind that the functionality it implies (and the amount of code required to implement it) are inherent in all true multi-tier architectures. Java Client provides most of the code for you. Unlike other multi-tier approaches, you do not have to worry about such things as change tracking, data packaging, and UI synchronization. In most cases, you need only write your business-logic code.

Data Synchronization Between Client and Server

In a Java Client application, when the user makes a query, the fetch specification is passed through the layers on the client (EOInterface to EOControl to EODistribution), largely through successive invocations of **objectsWithFetchSpecification**. The distribution layer on the client forwards the fetch specification to the server's distribution layer—in the default WebObjects case, synchronously via HTTP. From there the normal mechanisms take over and a SQL call is eventually made to the database server. The database server returns the rows of requested data and, as usual, this data is converted to enterprise objects and is registered with the EOControl layer on the server. The server's distribution layer then sends *copies* of the requested objects back to the client. When the EODistribution layer on the server receives the objects, it registers them with the editing context in the control layer and, through the interface layer's display-group and association mechanisms, the user interface is updated with the requested data.

Although requested objects are copied from the server to the client, and these objects exist in parallel object graphs on both server and client, the enterprise objects on the client usually do not exactly mirror the enterprise objects on the server. The objects on the client usually have a subset of the properties of the objects on the server (although the reverse can be true). You can partition your application's enterprise objects so that the objects that exist on the client (or the server) have a restricted set of data and behaviors.

Once the client has fetched data, this data is cached and is represented internally by the client's object graph. As users modify the data (or delete or add "rows" of data), the client's object graph is updated to reflect the new state. When users request that this data be saved, the changed objects are "pushed" to the server. If the business logic on the server validates these changes, the changes are committed to the database.

Note that Java Client automatically pushes updates from the server to the client. It also, by default, pushes changes before client-side objects remotely invoke methods on server-side objects.

Java Client as a WebObjects Application

Out of the box, Java Client runs as a type of WebObjects application. In the multi-tier architecture described earlier, WebObjects provides an application server as well as HTML and HTTP support. The distribution layer on the client provides an HTTP channel to handle communication between the application server and the Java Client applets in client Web pages.

A Java Client WebObjects application gives you considerable flexibility in how you compose the pages of your application. You can combine Java Client applets and static and dynamic (WebObjects) HTML elements in various ways. You can have pages with or without Java Clients or pages with multiple Java Clients, each with its own controller. For example, you could have a login page that takes the user to one of many Java Client pages based on some piece of account data. In addition, Java Client applets are not limited to the downloaded JFC components; as can any applet, they can create dialogs and secondary windows on the fly.

When you create a Java Client project using Project Builder, two things of specific interest (in terms of Java Client) are created for you:

- A subproject of type EOJavaClientSubproject named, by default, **ClientSideJava.subproj**. This subproject contains an Interface Builder "nib" (or interface) file whose file's owner is a custom subclass of EOInterfaceController.
- A **Main.wo** component which contains a subcomponent of type WOJavaClientApplet.

The EOJavaClientFramework is also automatically added to your project.

In an EOJavaClientSubproject, when you create a user interface using Interface Builder, the nib file stores an "archive" of JFC (and other 100% Pure Java™) objects. Also in the subproject are the interface controller and any other custom enterprise-object or other classes that you have implemented in Java.

WOJavaClientApplet is a component that is used to download and create an applet of class com.apple.client.interface.EOApplet. It has a dozen or so potential bindings, some general to applets (such as codebase and size) and others specific to Java Client (such as distribution-channel class and interface-controller class).

When you launch a Java Client application, a WebObjects application (**.woa**) instance is started on the server. Whenever a client requests a page of this application that has a WOJavaClientApplet, the **.class** files implementing the required objects are downloaded to the page and installed on the client. The applet is loaded and started and the user is ready to fetch, add, delete, and modify data.

As you can see from the diagram in Figure 3, each session created and managed by the WebObjects application has, if it is communicating with a Java Client application, its own editing context and its own server-side distribution layer. As described earlier, communication between the server and client is handled through the distribution layers on the server and the client. The WOApplication maintains the object store (EODatabaseContext) for all sessions.

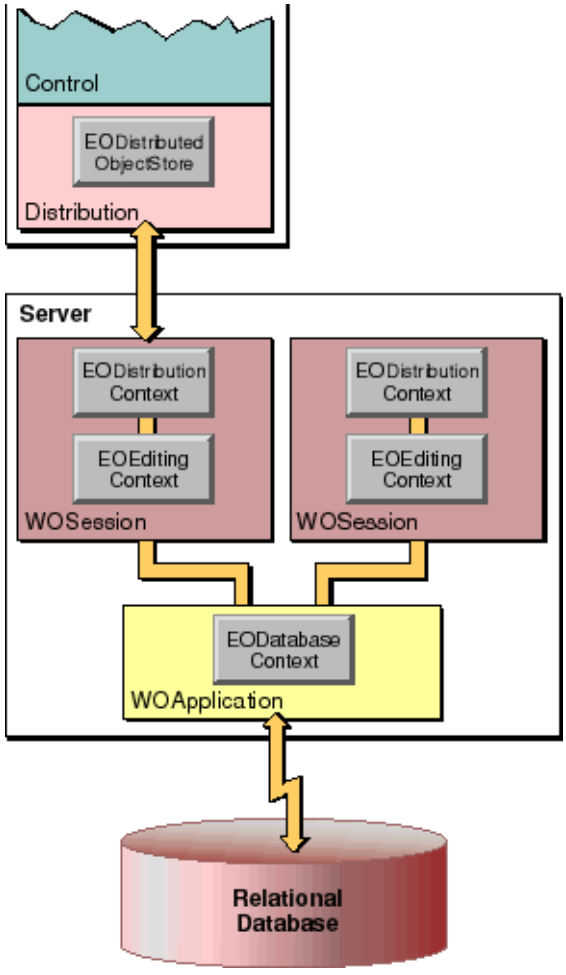


Figure 3. Java Client in a WebObjects Application

The session object is, by default, the delegate of the distribution layer's EODistributionContext, the object that handles communication on the server. The EODistributionContext class defines several security-related delegate methods for validating remote invocations; if you wish, you can implement these methods in your Session class.

Java Client Layers and Classes

The classes specific to Java Client are found in the distribution layers (both client and server) and in the client's control and interface layers.

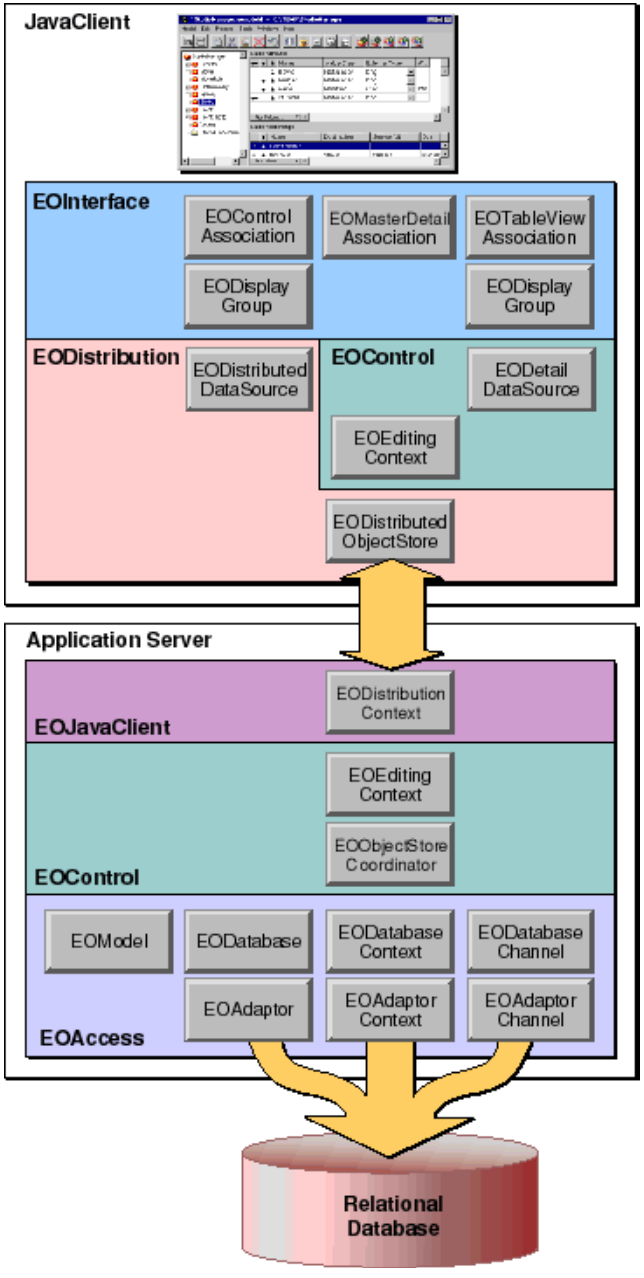


Figure 4. Major Classes in a Java Client Application

Client Interface and Control Layers

The EOInterface and EOControl layers on the client—implemented as the com.apple.client.interface and com.apple.client.control packages—contain classes *almost* identical (in terms of APIs and behavior) to their counterparts on the server, which are implemented as Yellow Box frameworks.

Basically, the EOInterface layer displays, in the user interface, properties of the enterprise objects in the control layer, using display groups and associations. Changes to the object graph are automatically synchronized with the user interface and user-entered data is automatically reflected in the object graph. The primary mechanisms behind this synchronization are display groups (EODisplayGroup) and associations (EOAssociation subclasses).

As in the server, the EOControl layer's primary responsibility is the management of the object graph through an EOEditingContext. It also implements faulting (on-demand fetching) and tracks editing changes.

The differences between the client and server layers are:

- The client Java classes are written in "100% Pure Java" and do not, as the server classes do, use bridging technology to access Objective-C code.
- The EOInterface layer on the client is implemented in terms of the Java Foundation Classes (instead of using Application Kit objects).
- The object store and the data source used by the client EOControl layer are objects in the distribution layer; basically, these objects communicate changes to the object graph across the channel to the server.
- The client layers include APIs that enable remote invocations of server methods.
- The EOControl layer on the client does not implement undo or redo.

The Distribution Layer

The distribution layer (implemented by the EODistribution package on the client and the EOJavaClient framework on the server) is responsible for synchronizing the states of the object graphs on the client and on the application server in the middle tier. The distribution layer moves properties in both directions, that is, as it fetches objects and saves changes.

The distribution layer has a server side and a client side. The classes in the server side of this layer are provided by the EOJavaClient framework (and associated "wrapped" Java classes). The classes on the client side are implemented in Java and live in the com.apple.client.eodistribution package.

Client Distribution Classes

The client-side distribution layer has four public classes.

EODistributionChannel and **EOHTTPChannel**. The distribution layer provides channels through which the application server and the Java clients communicate. The EOHTTPChannel class implements an HTTP channel, which is used by Java Client WebObjects applications, but you can subclass the abstract class EODistributionChannel and implement a channel that uses a different transport protocol (such as CORBA). On the client side EODistributedObjectStore handles communication over the channel; on the server side it's EODistributionContext.

EODistributedObjectStore. On the client the distribution layer provides a distributed object store. It handles interaction with the distribution layer's channel (an EODistributionChannel object), incorporating knowledge of that channel so it can forward messages it receives from the server to its editing contexts and forward messages from its editing contexts to the server.

EODistributedDataSource. A concrete subclass of EODataSource (which is defined in EOControl) that fetches using an EOEditingContext as its source of objects; the editing context, in turn, forwards the fetch requests to its object store (usually an instance of EODistributedObjectStore) where it is ultimately serviced by an EODatabaseContext on the server.

Server Distribution Classes

The EOClientJava framework has four public classes.

EODistributionContext. This class encodes data to send to the client and decodes data it receives from the client over the distribution channel. It also keeps track of the state of the server-side object graph so it can communicate any changes to the client and thus synchronize the object graphs. EODistributionContext (or its delegate) also validate remote invocations originating from client objects.

WOJavaClientApplet. The WebObjects component is used to download and create an applet of class com.apple.client.interface.EOApplet.

EOClassMapper. Gives the corresponding class names on the client and server. The methods in this class are typically of interest to those who are implementing their own channels.

EOReferenceRecording. Use to encode and decode objects in a pure Java environment. The methods in this class are typically of interest to those who are implementing their own channels.

In addition, **EOAccessAdditions.h** contains Objective-C categories on EOEntity, EOClassDescriptions, and EOEntityClassDescription. The methods in these categories return client-specific information stored in model files.

Programming With Java Client

Generally, programming a Java Client WebObjects application requires some skills and knowledge common to both Enterprise Objects Framework and WebObjects programmers. However, it also requires a specific design technique: object partitioning.

Objects on the server and the client can be instances of custom classes or generic enterprise objects (EOGenericRecord). Objects that derive from custom subclasses can have different sets of properties on both the server and the client. Usually, client objects have the more restricted set of data and behaviors, but it is really up to you to decide based on the requirements of the application and your business. As noted earlier, the primary criteria for partitioning are performance and security.

The basic tools and techniques for creating a Java Client application are covered in the tutorial for *Creating a Java Client WebObjects Application*.

Chapter 2

Tutorial

This tutorial shows you how to create a "Java Client" WebObjects application, which is a distributed Enterprise Objects Framework application that uses a Web browser as its display medium. The application is "distributed" in the sense that business logic can be shared among enterprise objects on the Web client (which are implemented in Java) and enterprise objects on the server (which can be implemented in Java or Objective-C). The steps you take to create a Java Client WebObjects application are remarkably similar to the steps you take to create a typical stand-alone (or "fat client") Enterprise Objects Framework application.

The application you'll be creating in this chapter, StudioManager, is based on the Movies sample database distributed with Enterprise Objects Framework (you must have the sample databases installed to do this tutorial). It centers around three types of enterprise objects: Studio, Movie, and Talent. Studios own movies, and they have a budget for buying new movies. Movies feature actors, or "talent." The StudioManager application lets you transfer movies between studios and buy all of the movies starring a particular actor. It also lets you add, modify, and delete studios.

The StudioManager example project upon which this tutorial is based is installed in *NEXT_ROOT*/**Developer/Examples/WebObjects/JavaClient**.

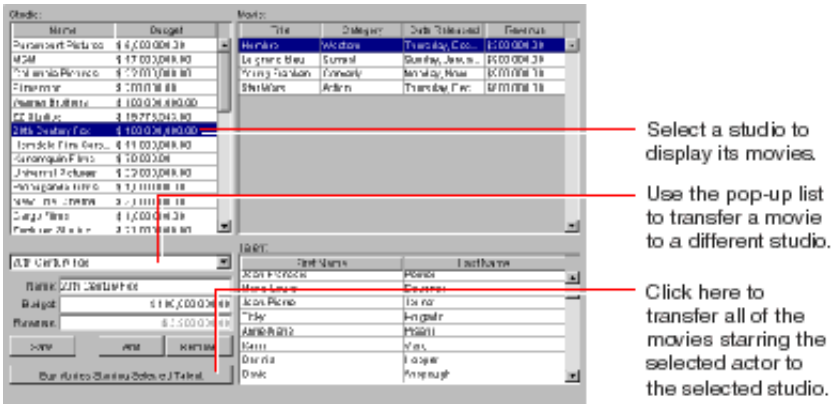
Requirements

To run the StudioManager application, or any Java Client application, you must have server and client systems with certain capabilities beyond the usual requirements for Enterprise Objects Framework applications (database servers, for instance). You must have a client (such as a Web browser) and a server platform that implement Java virtual machines (VM) on which "100% pure Java" applications can run. The client must also support the following standards:

- Java Foundation Classes (JFC), also known as "Swing"
- A transport layer such as HTTP or CORBA,

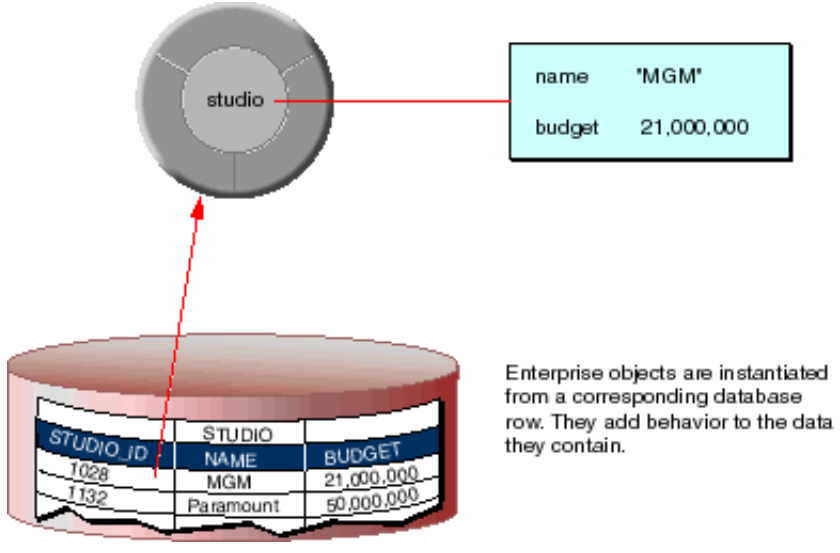
Currently, Java Client applications can be run from Microsoft Internet Explorer and Netscape browsers (with the Java Plug-in from SunSoft), and with the JDK's **appletviewer** and **java** programs.

Related Concepts: Java Client Architectural Overview



Enterprise Objects and Relational Databases

The Studio, Movie, and Talent enterprise object classes correspond to tables in a relational database. For example, the Studio enterprise object corresponds to the STUDIO table in the database, which has NAME and BUDGET columns. The Studio enterprise object class in turn has **name** and **budget** instance variables, or *class properties* (instance variables based on database data are called "class properties"). In an application, Studio objects are instantiated using the data from a corresponding database row, as shown in the following figure:



The enterprise objects in your application do not merely form a static representation of your database data, however. Enterprise objects add behavior to your data. For example, the Studio enterprise object class has a method for calculating the studio's portfolio value based on the revenue of its movies. It also has a method for buying all of the movies starring a specified actor.

In Java Client WebObjects applications, Enterprise Objects Framework manages the interaction between the database (on the server), your enterprise objects (on the server and client), and the user interface (on the client). Its primary responsibilities are as follows:

- Fetching data from relational databases into enterprise objects (on the server)
- Binding data in enterprise objects to the user interface (on the client)
- Keeping objects in the application synchronized with each other, with the database, and with the user interface; this includes keeping enterprise objects on the client synchronized with their counterparts on the server.

What Goes Into the StudioManager Application

As with most Java Client WebObjects applications, you create the Java Client StudioManager application using the following ingredients:

- **A model you produce using the EOModeler application provided with Enterprise Objects Framework.** A model defines a mapping between your enterprise objects and data in a relational database.
- **A user interface.** You use Interface Builder to construct a Web-based user interface that can interact with the Java Client. You must load a special palette (**EOJavaClient.palette**) as well as the standard **EOPalette.palette**. With the **EOJavaClient.palette** you can compose a user interface made from "widgets" derived from the Java Foundation Classes (JFC), informally known as Swing. Your application can also have pages dynamically generated entirely from objects on the server; for help in composing these pages, use WebObjects Builder.

- **Web components.** The Main component is automatically set up to have a WOJavaClientApplet component that is bound to the interface controller on the server. You can add other Web components with or without a Java Client linkage. Also provided are "skeletal" implementation files for the server-side application, session, and direct-action objects as well as API bindings files for server-side components.

- **Source code for enterprise object classes.** In the StudioManager application, these are Studio and Talent. Movie uses the default enterprise object class, EOGenericRecord, since it has no custom behavior. This is described in more detail in later sections.

In addition, the StudioManager application requires a database server on which you've installed the Movies example database. The final ingredients in the application are the Enterprise Objects Framework, WebObjects and Foundation classes, interfaces, and protocols, which you link into your application.

In this tutorial you'll learn the basic things you must do to create a Java Client WebObjects application. You'll discover how to:

- Create a new project using Project Builder.
- Create a new model based on the Movies database using EOModeler.
- Edit your project's nib file in Interface Builder.
- Write source code for the Studio and Movie enterprise object classes.
- Build your project in Project Builder.

Related Concepts: What is an Enterprise Object?

Creating the StudioManager Project

Every Java Client application starts out as a *project*. A project is a repository for all the elements that go into the application, such as source code files, makefiles, frameworks, libraries, packages, the application's user interface, sounds, and images. You use the Project Builder application to create and manage projects.

1. **Start Project Builder.**

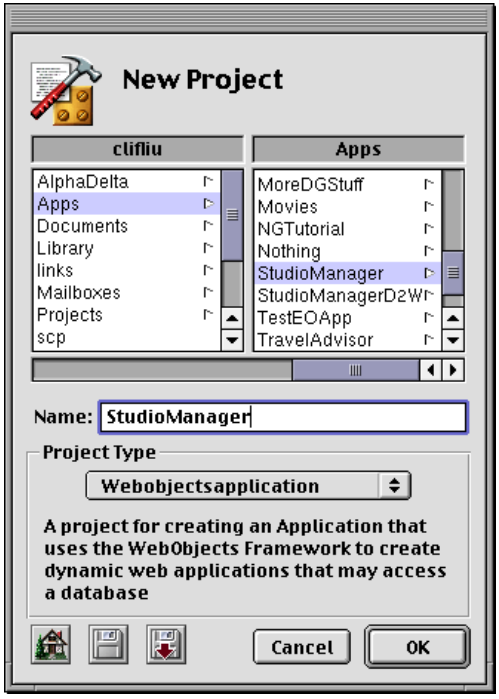
   Choose Project Builder from the WebObjects program group.

You must create or open a project to get Project Builder's main window. The New Project panel allows you to specify a new project's name and location.

2. **Make a new project.**

On Mac OS X Server:

Choose Project ▸ New.
Using the browser, select the directory in which you want the project to reside.
Type the name of the project in the Name field.
Select Webobjectsapplication from the Project Type pop-up list.
Click OK to create the project.



On Windows NT:

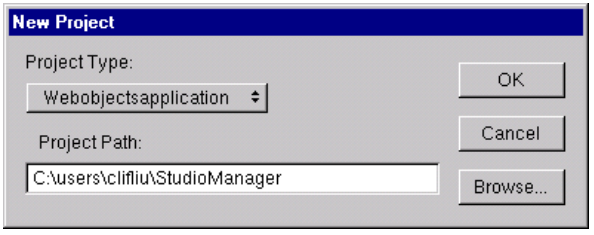
Choose Project ▸ New.

In the Project Path field of the New Project panel, type the file system path where the project is to reside.

Give the project the name “StudioManager” by typing this as the last component of the path.

Select Webobjectsapplication from the Project Type pop-up list.

Click OK to create the project.



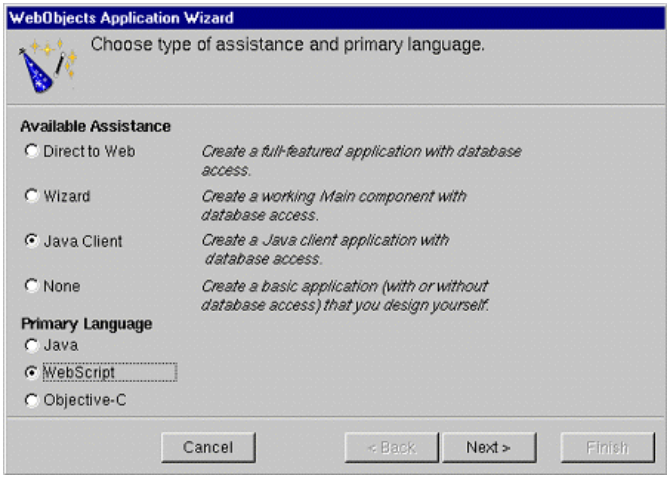
To name the project and give it a directory location, you can either use the Browse button to navigate to the directory in which you want to put the new project, or you can type the full path in the Project Path field. The item selected from the Project Type pop-up list causes Project Builder to include all frameworks and supporting files necessary for that type of project.

Using the Wizard

After you create a new project of the type “Webobjectsapplication” the WebObjects Application Wizard displays a succession of screens. The sequence of screens that you see depends on the options you select. The following sequence shows those screens that are displayed when you request a Java Client application that includes a newly created model and uses a EOF “skeletal” application template.

1. **Select the type of WebObjects application and the primary language.**

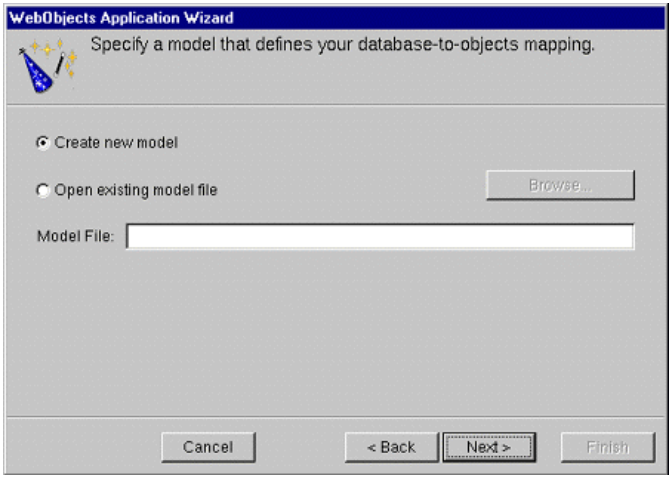
   Select Java Client under Available Assistance.
   Select Java under Primary Language.



You could also select Objective-C or WebScript as the primary language, but these languages are valid only for objects on the server. As noted earlier, objects on the client must always be implemented in Java.

2. **Create or select a model for your application.**

   For this tutorial, click “Create new model” and then click Next.



If you already have a model file, you can select "Open existing model file" and type the file system path to that file in the Model File field; or you can also click Browse and navigate to the file using a browser. This tutorial, however, invites you to follow a sequence of wizard screens that deal with creating a model file. If you chose an existing model file, go to Selecting the Application Template.

You can find a Movies model file (**Movies.eomodeld**) in *NEXT_ROOT*/**Developer/Examples/EnterpriseObject/AppKit/Movies**.

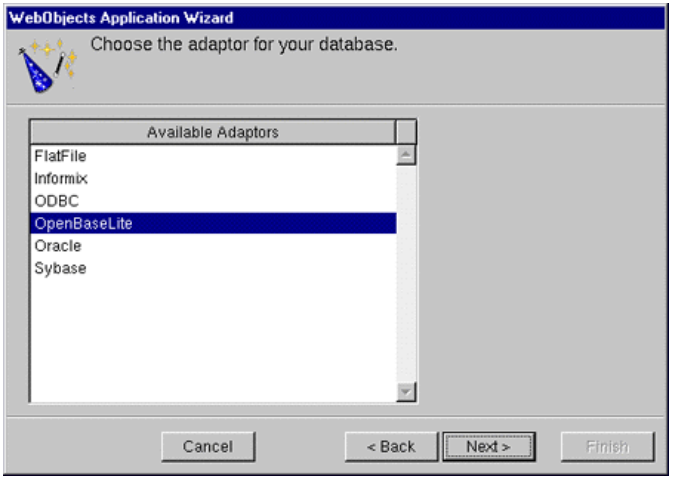
Related Concepts: What is a Model?

Creating a Model

If you decide to create a model for your application, the wizard steps you through a succession of windows. The model file, when created, is given the name of your project, in this case **StudioManager.eomodeld**.

3. Choose an adaptor for your database server.

Select an adaptor for the database you want to use.
Click Next.



An *adaptor* is a mechanism that connects your application to a particular database server. For each type of server you use, you need a separate adaptor. WebObjects provides adaptors for several relational database servers, such as OpenBase Lite, Informix, Oracle, and Sybase servers as well as the source

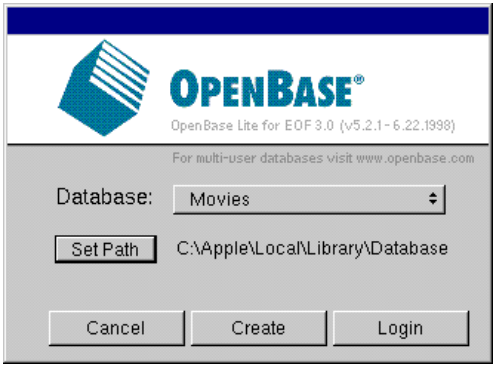
code for a flat-file database adaptor. If you're working on a Windows platform, WebObjects also provides an ODBC adaptor for use with ODBC-compliant database sources. It also provides the source code from which you can build an adaptor for a flat-file database.

All adaptors expect you to specify the database to use. In addition, before you can gain access to the database, you must log in. Different databases require different login information, so each database's login panel looks different. This tutorial uses the single-user OpenBase Lite adaptor for of the Movies database that is pre-installed with Enterprise Objects Framework.

4. **Log into your chosen database server.**

   Select the Movies database.

   Log into the database, filling in any required information.



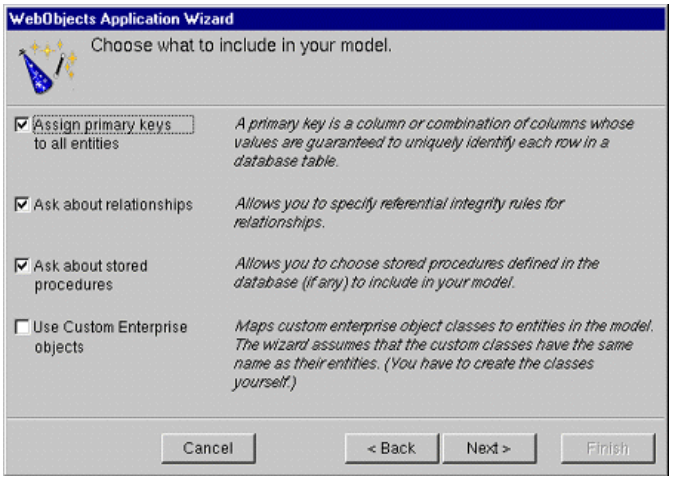
Note: If you are using the pre-installed OpenBase Lite database, click Set Path, browse to the *NEXT_ROOT***Local\Library\Databases** directory, and click Open. "Movies" now appears in the Database pop-up list. Click Login.

Related Concepts: Note to Oracle Users

After you log in, the wizard uses the selected adaptor to read the data dictionary (that is, schema information) from the database. From this dictionary it creates a default model, but before it does it lets you configure that model in four different ways.

5. **Choose options for the creation of your model.**

   Make sure every option is checked except “Use Custom Enterprise objects”.



The basic model the wizard creates contains *entities*, *attributes*, and *relationships*. How complete this model is depends on how completely the schema information is inside your database server. For example, the wizard includes relationships in your model only if the server’s schema information specifies foreign key definitions.

Using the options in this page, you can supplement the basic model with additional information. (Note that the wizard doesn’t modify the underlying database.) The following sections describe each option:

Assign primary keys to all entities

Enterprise Objects Framework uses primary keys as unique identifiers of enterprise objects with which it maps these objects to the appropriate database row. Therefore, you must assign a primary key to each entity you use in your application. The wizard automatically assigns primary keys to the model if it finds primary-key information in the database’s schema information. Checking this box causes the wizard later on to prompt you to choose primary keys if they aren’t defined in the database’s schema information.

Ask about relationships

If there are foreign-key definitions in the database’s schema information, the wizard includes the corresponding relationships in the basic model. However,

a definition in the schema information might not provide enough information for the wizard to set all of a relationship's options. Checking this box causes the wizard to prompt you later to provide the additional information it needs to complete the relationship configurations.

Ask about stored procedures

Checking this box causes the wizard to read stored procedures from the database's schema information, display them, and allow you to choose which to include in your model.

Use custom enterprise objects

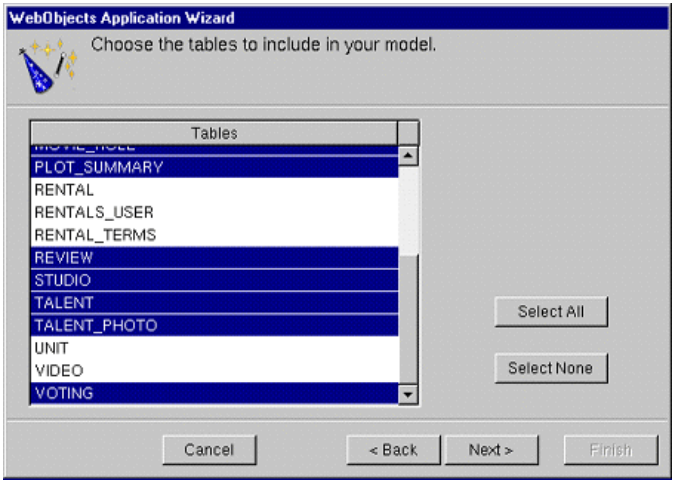
An entity maps a table to enterprise objects by storing the name of a database table (MOVIE, for example) and the name of the corresponding enterprise object class (a Java class such as Movie). When deciding what class to associate with an entity, you have two choices: EOGenericRecord or a custom class. EOGenericRecord is a class whose instances store as key-value pairs an entity's properties and the data associated with each property. They do nothing else.

If you don't check the "Use custom enterprise objects" box, the wizard maps all your database tables to EOGenericRecord. If you do check this box, the wizard maps all your database tables to custom classes. The wizard assumes that each entity is to be represented by a custom class with the same name. For example, a table named MOVIE has an entity named Movie, whose corresponding custom class is also named Movie.

Use a custom enterprise object class only when you need to add business logic; otherwise use EOGenericRecord. Note that this option, if selected, only assigns a class name; it does not create a class. The class of an entity remains EOGenericRecord, even if EOModeler shows a different class name, until you create the "skeletal" class file and add this file to the project. You'll perform this step later using EOModeler.

6. **Select the database tables to include your model.**

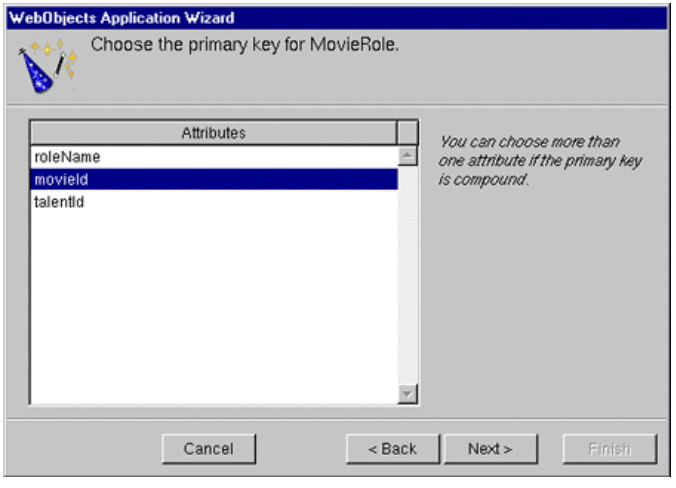
   Select the following tables: DIRECTOR, MOVIE, MOVIE_ROLE, PLOT_SUMMARY, REVIEW, STUDIO, TALENT, TALENT_PHOTO, and VOTING.



After you select the database tables for your model, the next panel displayed depends upon your database. Unless you are using a database that stores primary key information in its database server's schema information, the wizard now asks you to specify a primary key for each entity.

7. **Specify the primary keys for your entities.**

   See the table below.

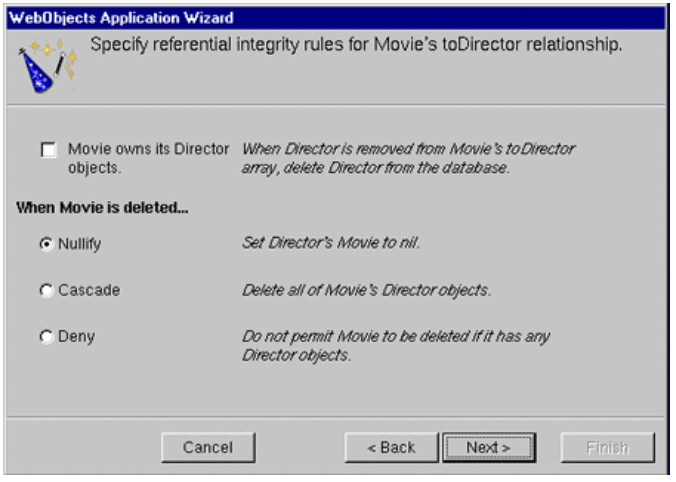


The entities in your model should have the following primary keys assigned:

The entity...	Should have the primary key attributes...
Director	movieId and talentId
Movie	movieId
MovieRole	movieId and talentId
PlotSummary	movieId
Review	reviewId
Studio	studioId
Talent	talentId
TalentPhoto	talentId
Voting	movieId

8. Specify referential integrity rules for the relationships in the model.

Select the Nullify button in each “referential integrity” window that appears.



If you’re using a database that stores foreign key definitions in its database server’s schema information, the wizard reads them and creates corresponding relationships in your model. (The naming convention for relationships varies according to the adaptor you’re using.) The wizard now asks you to specify referential integrity rules for the relationships so it can further configure them.

<Source object> owns its *<destination>* objects

This option specifies that a destination object in a relationship can't exist without its source object; the source object is said to "own" the destination object or objects in the relationship. For example, consider the case of Movie's to-many relationship to MovieRoles, which it owns. When a MovieRole is removed from its Movie's array of MovieRoles, the MovieRole is deleted—deleted in memory and deleted in the database.

When *<source object>* is deleted

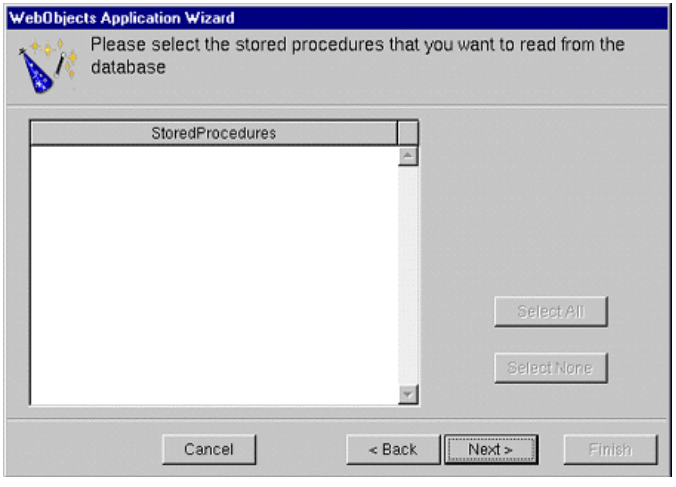
This set of options specifies what to do when the source object in a relationship is deleted.

- **Nullify.** Specifies that when the source object is deleted, any reciprocal relationship that the destination object has with the source object is set to **null**.
- **Cascade.** Specifies that when a source object is deleted, the source's destination objects should also be deleted—again, deleted in memory and correspondingly in the database.
- **Deny.** Specifies that if the relationship's source (for instance, a Talent) has any destination objects (MovieRoles), then the source object can't be deleted.

9. Select the stored procedures you want to include in your model.

This panel lists the stored procedures defined in your database, with all procedures selected by default.

Click Next.

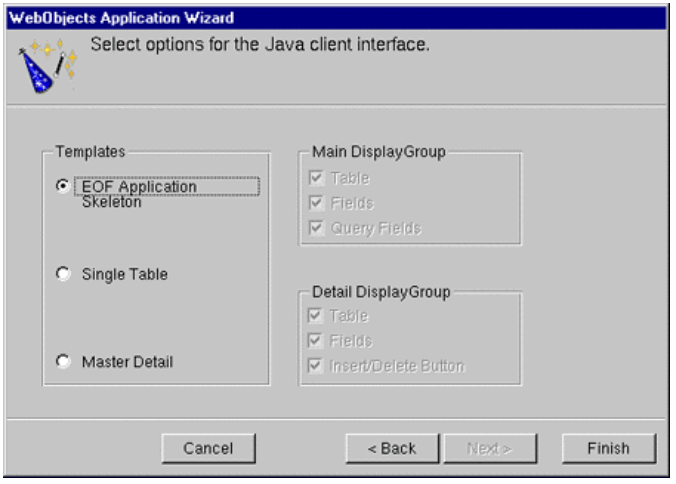


Selecting the Application Template

After you create a model or choose an existing one, the wizard displays a final screen that lets you select the type of template to use for your project.

10. Select the template project to use for the Java Client application.

Make sure the EOF Application Skeleton radio button is selected.
Click Finish.



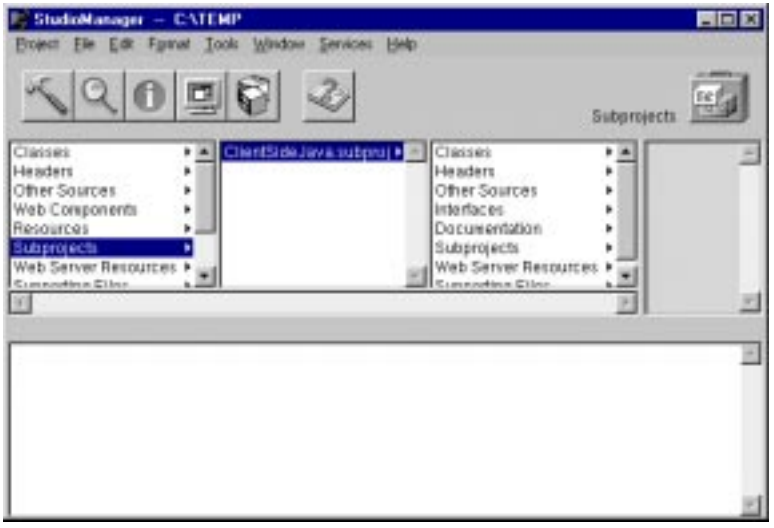
The EOF Application Skeleton creates an application project with a "blank" user interface; you must construct the interface by hand in Interface Builder. For StudioManager, this is the option you want. The other two options automatically generate different types of interfaces:

- Single Table. The wizard guides you through the creation of an application with a single EODisplayGroup represented in a single table view.
- Master Detail. The wizard guides you through the creation of an application that has a master-detail interface.

For more on the last two options, see "Creating a WebObjects Database Application" in *Getting Started With WebObjects*.

The Ingredients of a Java Client Project

Once you've finished with the wizard, Project Builder creates a project directory named after the project—in this case StudioManager—and populates this directory with an assortment of ready-made files and directories. It then displays its main window.



Client Files

The significant addition to a Java Client project is a subproject named **ClientSideJava.subproj**. This subproject comes with two preconfigured files: a **.java** file reflecting the name of the project (in this case, **StudioManager.java**) and, in the Interfaces "suitcase," an Interface Builder archive (or "nib") file, also named after the project (**StudioManager.nib**).

The Nib File

The nib file in a Java Client application seems identical to nib files in stand-alone Yellow Box applications. You drag objects from palettes onto a window "surface" and these palettes and their objects look exactly like objects in stand-alone. However, these similarities of appearances are deceiving.

When the EOJavaClient palette has been loaded into Interface Builder and you create a user interface, the nib file contains two parallel object graphs, one populated with Yellow Box objects and the other with Swing (JFC) objects.

The Swing object graph constitutes a "Java archive" that is loaded onto the client.

The Interface Controller

In a Java Client application an interface controller—an EOInterfaceController object—mediates between the applet interface and the model objects on the client. When you use Project Builder to create a Java Client project, it automatically generates code for a custom EOInterfaceController subclass and makes an object of this class the owner of the nib file. The class is named after the project and includes the package prefix of *project*.**client**.

In the Model-View-Controller design paradigm, the interface controller plays the role of (obviously) controller. It has four outlets:

- To its **component**, which is preset to the window in the nib file and functions as the "view" (it can be set to something else)
- To the client's editing context (**editingContext**), which serves as the "model"
- To the controller display group (**controllerDisplayGroup**), a kind of general-purpose display group that contains the interface controller itself and nothing else; through it the applications can specify user-interface dependencies and can control the interface through associations
- To the master display group (**masterDisplayGroup**) in master-detail interfaces

Server Files

The server-side project files are, as usual, accessible from the first column of the project browser (the main project). Most notable of these is the Main component (**Main.wo**) in the WebComponents suitcase. The **Main.html** file contains this generated HTML code:

```
<HTML>
<HEAD>
  <TITLE>Main</TITLE>
</HEAD>
<BODY>
  <CENTER><WEBOBJECT NAME=Applet></WEBOBJECT></CENTER>
</BODY>
</HTML>
```

The WOJavaClientApplet Component

The "Applet" WEBOBJECT tag in the HTML above represents a WOJavaClientApplet component. Java Client applications use this component to create an applet (of class com.apple.client.interface.EOApplet) and to pass this applet several parameters, some standard, such as size and codebase, and others specific to Java Client applications, such as channel class and interface-controller class.

The **Main.wod** file created by Project Builder contains the following default bindings for WOJavaClientApplet:

```
Applet: WOJavaClientApplet {
    height = 512;
    width = 512;
    interfaceControllerClassName =
"studiomanager.client.StudioManager";
    useJavaPlugin = NO;
}
```

Note that Project Builder automatically provides the binding for **interfaceControllerClassName** (see "The Interface Controller," above for details).

The WOJavaClientApplet bindings specific to the EODistribution layer are:

Property	Value
useJavaPlugin	If YES, generates HTML that causes Internet Explorer and Netscape browsers to use SunSoft's Java Plug-in.
distributionContext	The EODistributionContext that the applet uses to handle requests from the client. If no binding is specified, WOJavaClientApplet instantiates one with the session's default editing context and sets the session as the delegate of the distribution context and itself as the invocation target.
interfaceControllerClassName	The name of the initial EOInterfaceController subclass.
applicationClassName	The name of the EOApplication subclass used for the shared application object.

Property	Value
language	The preferred language for the application. This corresponds to a localized *language***.lproj** directory in the application's resources. When searching for localized resources, Java Client first looks in the **.lproj** directory of the preferred language, next **English.lproj** (if English is not the preferred language), and finally for non-localized resources.
channelClassName	The class name of the distribution channel to be used by the client, EOHTTPChannel by default.

Other Server Files

A Java Client project includes these other server-side files:

- The application, session, and direct action class (**.java**) files; if you selected Objective-C as the primary language, these would be Objective-C implementation (**.m**) and header files.
- In the Resources suitcase, the model file (**StudioManager.eomodeld**)
- Also in the Resources suitcase, the exported bindings file for the Main component (**Main.api**)
- In the Supporting Files suitcase, the makefiles **Makefile**, **Makefile.preamble**, and **Makefile.postamble**.

Related Concepts: Customizing Your Project With Wizards

Verifying and Modifying the Model

Even if you use an existing model or create a model with the wizard after selecting all the automated functions, you still need to do, or at least verify, the following things:

- Make sure that each entity has a primary key.
- Specify the properties that you don't want displayed, particularly keys.
- Add or modify relationships to the Studio, Movie, Talent, and Movie Role entities.
- Generate source files for the Studio and Talent classes.

1. **Open the model file.**

   In Project Builder's project browser, click Resources in the leftmost column

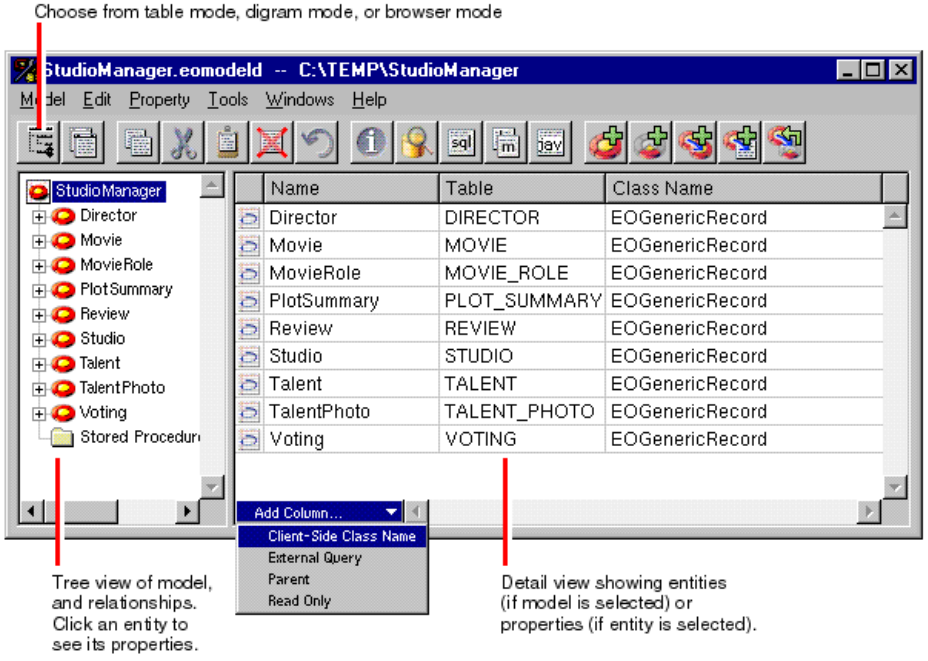
   Select **StudioManager.eomodeld**.

   Double-click the EOModeler document icon, displayed above the right side of the project browser.

When the model document is opened, the Model Editor (in table mode by default) lists the entities you selected from the Movies database, along with the names of their associated database tables and class names.

2. **Add a column for client-side classes.**

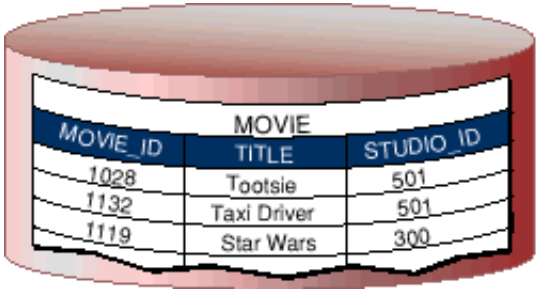
   If no Client Side Class column appears in the Model Editor, choose Client-Side Class Name from the pop-up list at the bottom of the Model Editor.



The steps you should perform with EOModeler are described in more detail in the following sections.

Assigning Primary Keys

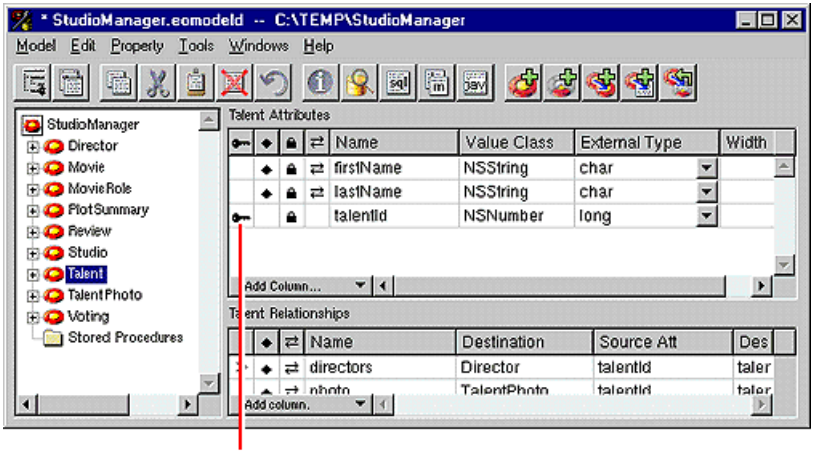
In a relational database, each table has a column or combination of columns whose values are guaranteed to uniquely identify each row in that table. For example, in the Movies database the MOVIE table has as its primary key the column MOVIE_ID. Each row in the MOVIE table has a different value in the MOVIE_ID column, which uniquely identifies that row. Two movies could have the same name, but still be distinguished from each other by their primary keys.



MOVIE_ID is the MOVIE table's primary key. This means that each row has a unique value in the MOVIE_ID column.

Enterprise Objects Framework uses primary keys to uniquely identify enterprise objects and to map them to the appropriate database row. Therefore, you must make sure that each of your entities has a primary key assigned to it in EOModeler. If your database has primary keys defined in it, this information is automatically included when you create a model—in that case, you don't need to assign primary keys yourself.

1. **Make sure that each entity has a primary key.**



The following table lists the primary keys that should be assigned to each of the entities in the model. Note that entities (such as MovieRole) can have a *compound primary key*; that is, a primary key that is composed of more than one attribute. However, EOModeler can assign compound primary keys (such as rowId and movieId) when only a single primary key is necessary. For more discussion of this subject, see the appendix "Entity-Relationship Modeling" in the *Enterprise Objects Framework Developer's Guide*.

The entity...	Should have the primary key attributes...
Director	movieId and talentId
Movie	movieId
MovieRole	movieId and talentId
PlotSummary	movieId
Review	reviewId
Studio	studioId
Talent	talentId
TalentPhoto	talentId
Voting	movieId

Removing Primary and Foreign Keys as Class Properties

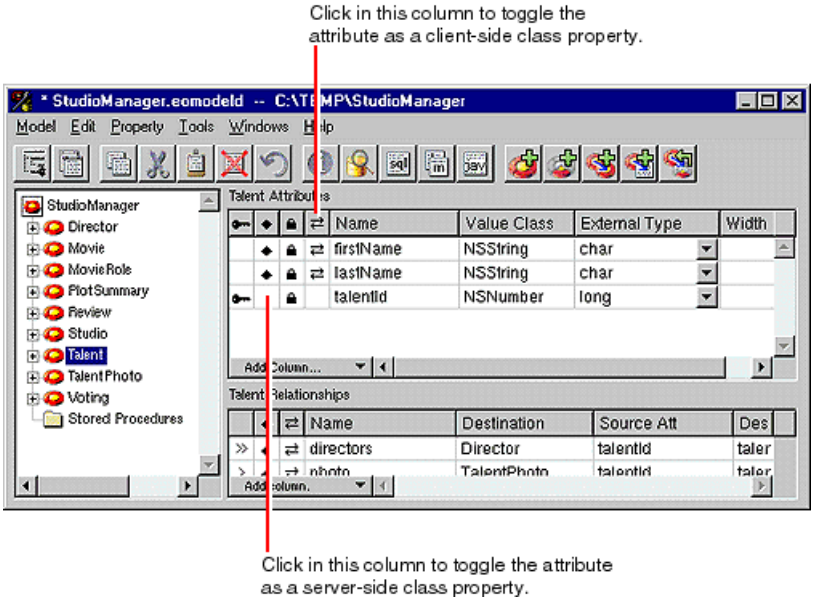
By default, EOModeler makes class properties for all of an entity's attributes (except for non-database attributes that you add to the entity). When an attribute is a class property, it means that the property will be included in your class definition and that it can be fetched from the database. To put it another way, only attributes that are marked as class properties become part of your enterprise objects.

You should only mark as class properties those attributes whose values are meaningful in the objects that are created when you fetch from the database. Attributes that are essentially database artifacts, such as primary and foreign keys, shouldn't be marked as class properties unless the key has meaning to the user and must be displayed in the user interface. For more discussion of primary and foreign keys, see the section "Adding Relationships" on page 64.

Eliminating primary and foreign keys as class properties has no adverse effect on how Enterprise Objects Framework manages enterprise objects in your application.

2. Remove primary and foreign keys as class properties.

In the model-entity view of the Model Editor, select the entity you want to modify.
Identify an attribute (typically a primary or foreign key) that you do not want to be a class property
Click the diamond icon next to the attribute to remove it as a server-side class property.
Click the double-arrow icon next to the attribute to remove it as a client-side class property.
Save the model by choosing Save from the Model menu.



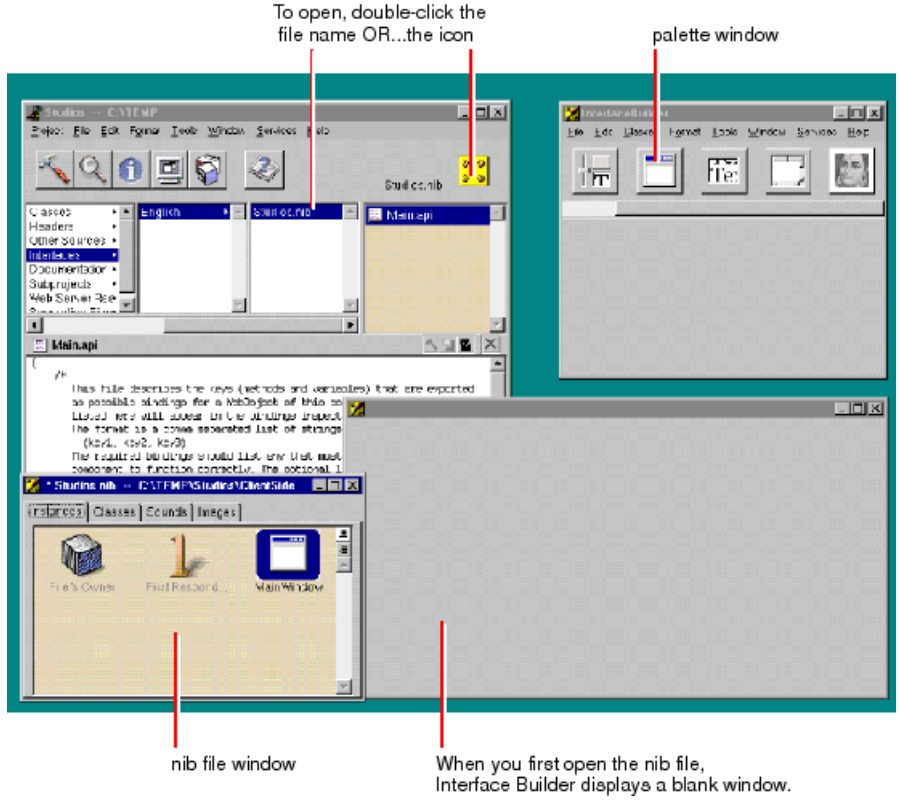
You'll be returning to EOModeler to enhance your model in later exercises, but for now you're ready to build the first stage of the StudioManager application.

Creating the User Interface

When you create a Java Client WebObjects application project, Project Builder puts a nib file in the Interfaces suitcase of the ClientSideJava subproject. A nib file is primarily a description of a user interface (or part of a user interface); it is created by the Interface Builder application and it can be archived along with other resources of your application. The nib file in the ClientSideJava subproject, however, is quite unlike the nib files in typical applications. When the EOJavaClient palette is loaded and you construct a user interface, the objects that a nib file contains are derived from the Yellow Box frameworks *and* the Java Foundation Classes (JFC), or Swing. They thus can be downloaded to Java Client applets that live on the client.

1. **Open the StudioManager.nib file.**

   In the project browser navigate to Subprojects ClientSideJava Interfaces ▶ English.
   Select **StudioManager.nib**.
   Double-click to open.



By default, a blank window appears when Interface Builder is launched. This is the window you'll use to create your user interface.

The Interface Builder application is located in the WebObjects program group. The icon for the application is this:

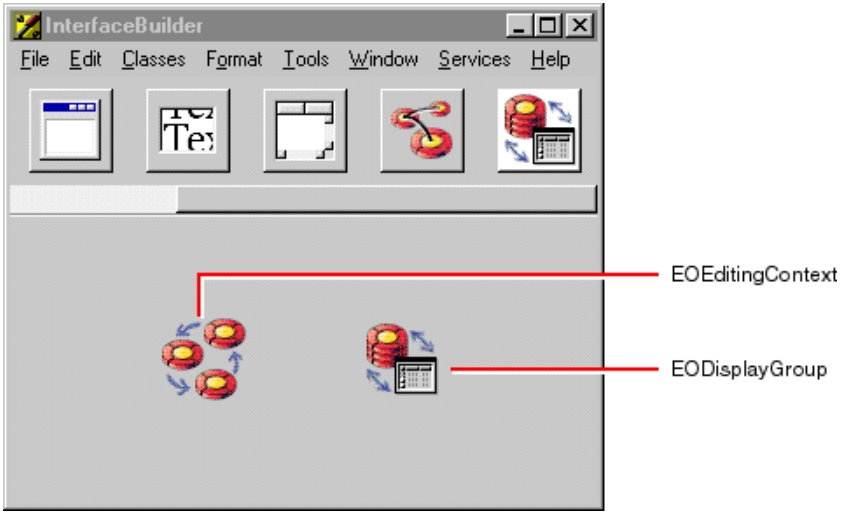
In Interface Builder you typically construct a user interface by dragging objects from a palette and dropping them into the window. Java Client WebObjects applications require that two special palettes be loaded into Interface Builder:

- **EOPalette.palette** includes two objects: EODisplayGroup and EOEditingContext.
- **EOJavaClientPalette.palette** has no visible objects but contains the code that creates Swing objects equivalent to the Yellow Box objects on the standard palettes.

If these palettes are not loaded, you must load them.

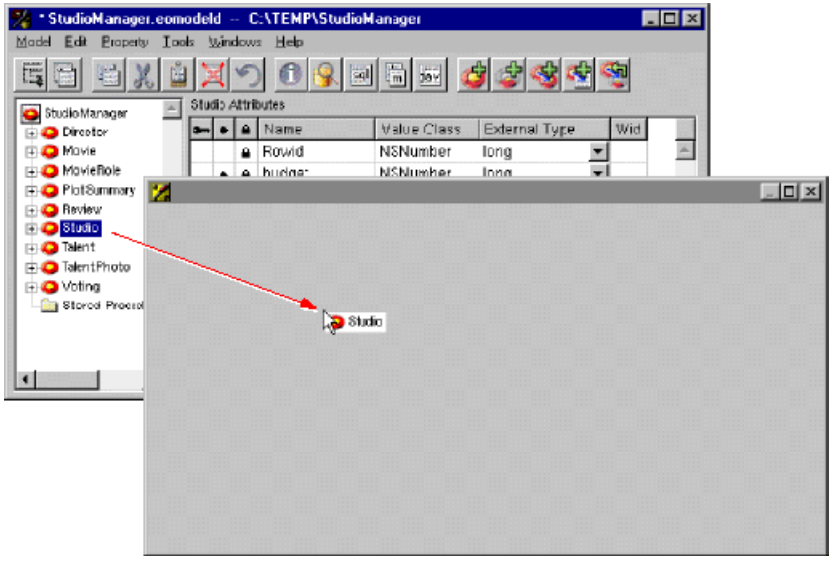
2. **Load the required palettes.**

   In Interface Builder, choose Tools ▸ Palettes ▸ Open.
   In the Open Palette panel, navigate to **NEXT_ROOT/Developer/Palettes.**
   Double-click **EOPalette.palette**.
   Perform the same sequence of steps, but this time load **EOJavaClientPalette.palette**.



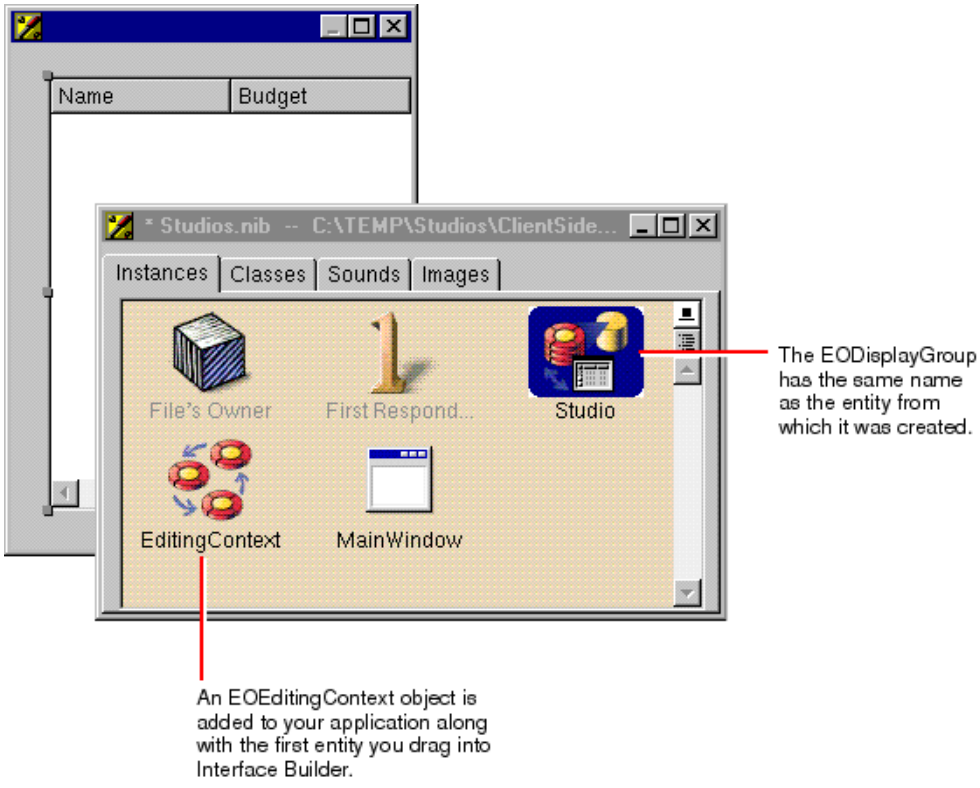
You'll be dragging objects off of the palette later. For now, however, you can construct a basic interface for a Java Client WebObjects application by simply dragging icons from EOModeler into Interface Builder.

3. **Drag the Studio entity from EOModeler into the window.**



The following figure shows the results of dragging an entity into your window. In the nib file window, there's a new EODisplayGroup that's named "Studio" after the entity you dragged in. Note that the nib file window also includes an EOEditingContext object. An EOEditingContext object is added to your application along with the first entity you drag into Interface Builder. Because a document typically only needs one EOEditingContext, this object is only added once.

Related Concepts: What are EODisplayGroups and EOEditingContexts?



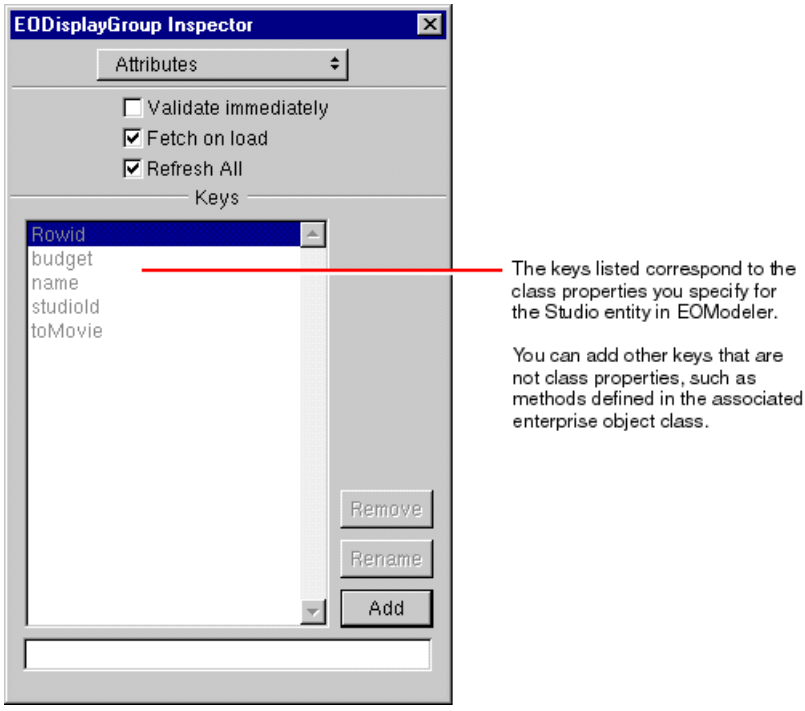
An entity EODisplayGroup has keys that correspond to the properties in its associated entity. You can examine these keys in the EODisplayGroup Inspector.

4. Examine the EODisplayGroup in the Inspector.

Select the Studio EODisplayGroup in the nib file window.

Choose Tools ▶ Inspector.

Make sure that the “Fetch on load” checkbox is checked.



The "Fetch on load" option is important because it allows data to be fetched from the database when you start your application.

The interface that was created when you dragged an entity into the window is already a functional (if simple) application. You can test it.

5. Test the interface.

On Mac OS X Server, choose Document ▶ Test Interface. Click  in the menu bar to exit the test.

On Windows NT, choose File Test Interface. Choose File Exit to exit the test.

Note that because the "Fetch on load" option was enabled for the Studio EODisplayGroup in the Inspector, the data is automatically fetched when you test your interface.

Formatting Currency Values and Dates

When an attribute is defined in your model as having the internal type Number, a currency formatter is automatically added to any control with which the attribute is associated. Likewise, when an attribute has an NSGregorianDate type, date formatters are added to controls associated with it.

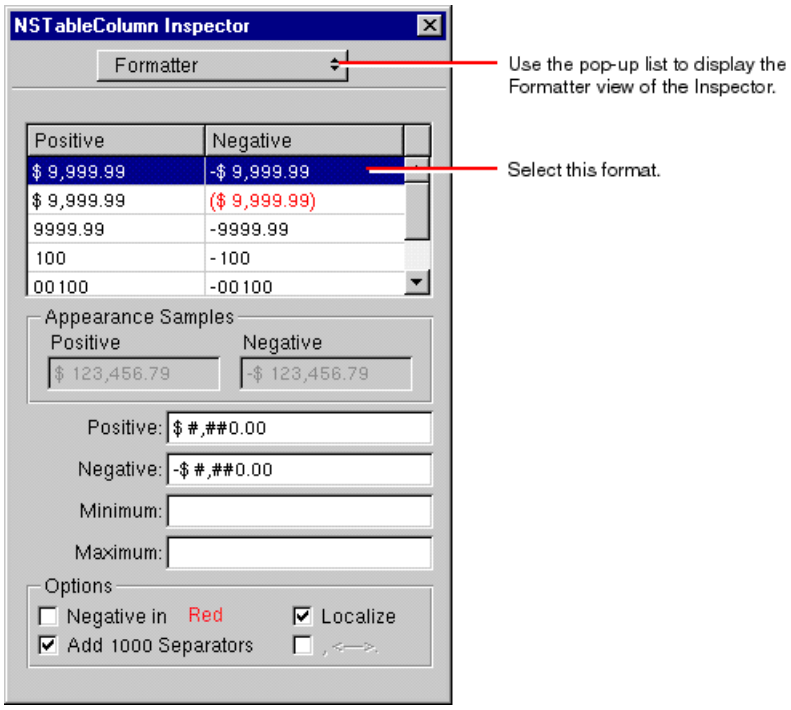
Not all adaptors map, say, the **budget** attribute to the Number data type. In that case, you won't automatically get currency formatting in the column. You can fix this problem either by setting **budget**'s internal type to be Number in the model, or you can add a currency formatter to the column (as described in "Writing Derived Methods" on page 83).

Once a control has a formatter, you can use the Inspector to change it.

6. Set field formatting.

Select the **Budget** column head in the table view, and display the Formatter view of the NSTableColumn Inspector.

Change the format to a standard currency format.



Do not set the format to show negative values in red. The JFC currently does not implement colored text.

Adding Action Methods

You can add basic behavior to your application, such as giving it the ability to add, delete, and save objects, without writing a line of code. This is possible because the EODisplayGroup, EOEditingContext, and EOInterfaceController objects in Interface Builder have predefined action methods that you can use to trigger operations in your application. An action method is a method that's invoked when the user clicks a button or another control object.

7. Add action methods.

Add three buttons to your window and label them “Add,” “Remove,” and “Save.”

These buttons will be used to insert new studios, delete existing studios, and save changes.

Control-drag from the Add button to the Studio EODisplayGroup.

In the Inspector, select Outlets from the pop-up list at the top of the left column.

Select **target** in the left column.

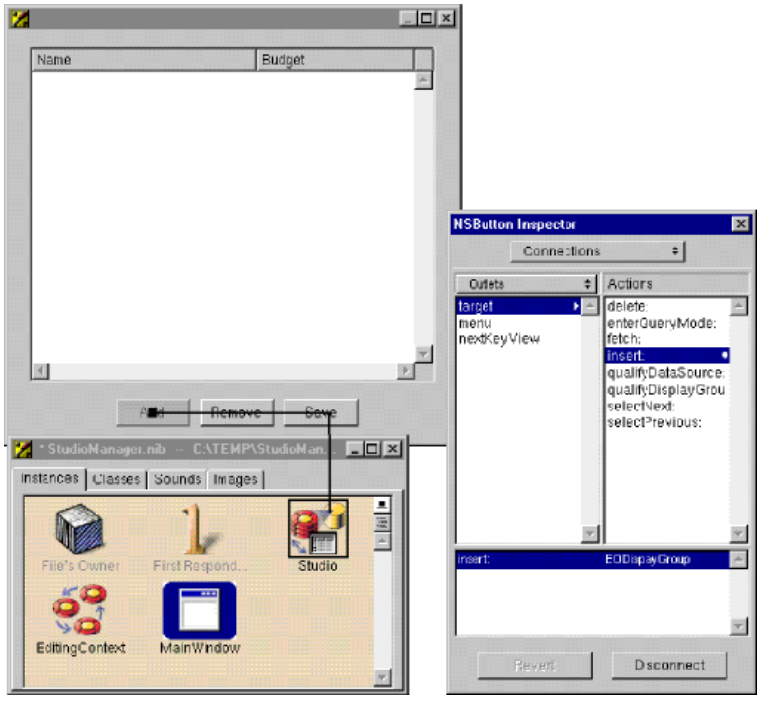
Double-click **insert:** in the right column.

Using the same process, connect the Remove button to the **delete:** method.

To connect the Save button, control-drag from the button to the File’s Owner object in the nib file window.

In the Inspector, select **target** in the left column.

Double-click **save:** in the right column.



The File's Owner icon represents the object that "owns" the nib file, or the nib file's root object. In a Java Client WebObjects application, this object is an instance of a custom subclass of EOInterfaceController that is automatically created for you (**StudioManager.java**, in this case). EOInterfaceController defines the **save** method and implements it to commit changes to the database.

Note: The EOEditingContext object in the nib file ("EditingContext") also defines a method—**saveChanges**—that also commits changes to the database. However, EOInterfaceController's method is preferable because it catches exceptions that might arise from this operation.

Building and Testing Your Application

You have now created a Java Client application—a fairly trivial one, to be sure, but still one with all the essential ingredients. Now and then it is a good idea to build and test your application to catch any problems. Interface Builder gives you a way to test your user interface even before any code is compiled.

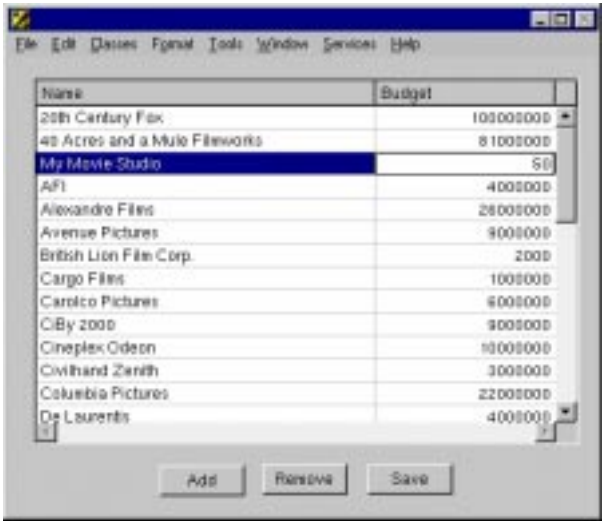
However, to gauge the complete picture, you still should build your application and test it using a browser or whatever other tool is intended for deployment.

Testing the Interface

With your current interface you can test-run your application in Interface Builder and try inserting and deleting some Studio objects.

1. **Test your interface.**

   Choose Document Test Interface (File Test Interface on Windows NT).

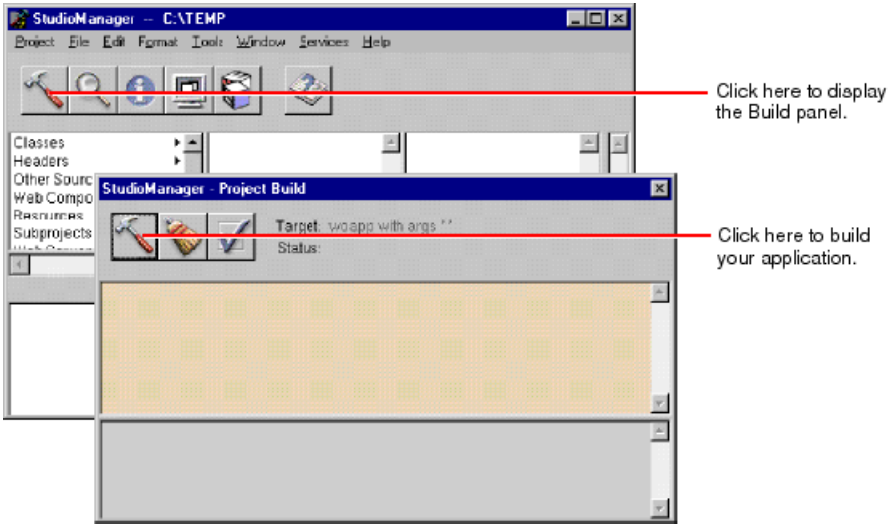


You will find that you cannot save your changes to the database. The save fails because the File's Owner object (an instance of a custom subclass of EOInterfaceController) is created on the client side when the application is started. No Yellow Box object corresponding to the File's Owner is available during testing. If you had any custom code, Interface Builder would also have no way to test it. To test the save function or any custom code, you must build the project and then run the application.

Building the Application

You build a Java Client project using Project Builder.

2. **Use Project Builder to build the application.**



If there are any coding or linking errors the Project Build panel displays them; click an error message in the upper part of the panel to go to the site of the error in the code editor.

Related Concepts: Debugging Java Client WebObjects Applications

Running a Java Client Application

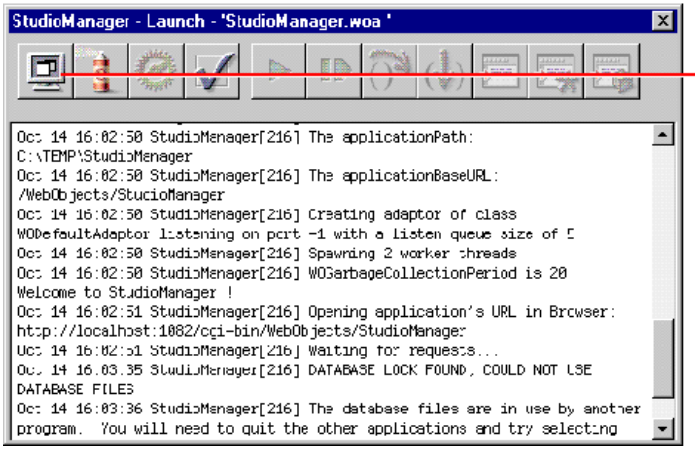
A Java Client application is really two applications; one application is on the server and the other is on the client, and they must be running concurrently. You start the server application as you do any WebObjects application in one of the following ways:

- Using Project Builder (during development and testing phases)
- From the command line
- Using the Monitor application (the preferred deployment mechanism)

For the procedures for the last two alternatives, check *Serving WebObjects*. You can launch the server application from Project Builder using the Launch panel.

1. **Launch the application with Project Builder.**

   Click the Launch button on the main window.
   Click the Launch button on the Launch panel



After starting the server application, start the client application; there are several ways to do this:

- **Using the Java interpreter (java)**: To start the client as a stand-alone Java application outside a browser, use the **java** interpreter. The syntax for using java to start a Java Client application is:

  ```
  java [-classpath classpath]
  com.apple.client.eoapplication.EOApplication
  -applicationURLurl
  [-page pageName]
  ```

  You might want to create a script file to make this command automatic and hidden.

  It might not be necessary to specify the `-classpath` option, but if the interpreter cannot find classes, you must either modify your CLASSPATH environment variable or add the `-classpath` option to the command. The `-applicationURL` option specifies the application's URL that you would also use in a browser and the `-page` option specifies the name of the page that contains the WOJavaClientApplet component. If *pageName* is not specified, "Main" is assumed.

  Please note that "com.apple.client.eoapplication.EOApplication" is the name of the class that contains the static main function that is usually

used to start up a Java Client WebObjects application. If you have a different main function you must specify the name of the class that implements it instead.

- **Using Microsoft Internet Explorer browsers**: To use your Java Client application in this browser, you must use version 4.0 or higher. To view the debugging output, launch Internet Explorer, choose Internet Options from the View menu, enable both Java Logging and Java Console in the Advanced options display, restart Internet Explorer, and select View ▸ Java Console. You should use Sun's Java Plug-in with Internet Explorer because there are bugs in the browser's Java implementation such as known problems with combo boxes. In addition, if you start a new applet in a browser that has run another applet, the new applet freezes because the browser's Java virtual machine is not restarted. You will need to restart the browser every time you launch your application; quit the browser and then launch the client. This procedure is not necessary if you use Sun's Java Plug-in. If you wish, WebObjects can automatically launch the browser for you.

- **Using Netscape browsers**: To run your Java Client application with a Netscape browser, you currently have to use Sun's Java Plug-in. If you wish, WebObjects can automatically launch the browser for you.

- **Using appletviewer**: The JDK's **appletviewer** tool is very useful during development because it minimizes your start-up time by removing the need to launch a browser. It also lets you view the debugging output inside the shell where you run **appletviewer**. To use the tool, copy the URL of the server application (displayed in the console output) and paste it a shell window as the argument, for example:

  ```
  appletviewer http://<host>:1234/WebObjects/MyApp
  ```

  If you are running **appletviewer** on the same machine as the WebObjects application, *<host>* is "localhost"; otherwise it is the host name of the machine on which the application is running.

There are a few considerations to keep in mind when running a Java Client WebObjects application:

- You can specify `-WOAutoOpenInBrowser NO` on the command line to avoid auto-launching a browser when you start up your server application.

- The CLASSPATH environment variable must be correctly set so your application can find all necessary Java classes. If you're using **appletviewer** or the Java interpreter, you need to include the directory containing your client-side classes. The following CLASSPATH works for a development environment where the client application is running on the server.

  On Mac OS X Server, type the following into the Terminal window from which you execute **appletviewer** or the Java interpreter.

  ```
  setenv CLASSPATH "/System/Library/Java:/System/Library/F
  rameworks/JavaVM.framework/Classes/swingall.jar:<Direct
  oryContainingStudioManager>/StudioManager/StudioManager
  .woa/WebServerResources/Java"
  ```

  On Windows NT, type the following into the Bourne Shell from which you execute **appletviewer** or the Java interpreter..

  ```
  export CLASSPATH="C:\Apple\Library\JDK\lib\swingall.jar;
  C:\Apple\Library\Java;C:<DirectoryContainingStudioManag
  er>\StudioManager\StudioManager.woa\WebServerResources\
  Java"
  ```

- If you run the application in a Microsoft Internet Explorer or Netscape browser, you may have to use Sun's Java Plug-in. These browsers currently do not implement the AWT specification exactly or have bugs that prevent Java Client applications from working correctly. In particular, Microsoft Internet Explorer does not reset the Java virtual machine which can cause the application to freeze. To use the plug-in, open the Web component containing your application's WOJavaClientApplet in WebObjectsBuilder and set the **useJavaPlugin** binding to YES. The first time you start an application using the plug-in, the browser will ask you to download the plug-in (the concrete behavior depends on the browser). Afterwards, the plug-in is loaded automatically. Please refer to Sun's documentation at **http://java.sun.com/products** for more information.

What if It Doesn't Work?

What if you test-run the application in Interface Builder, or if you build and run it, and it doesn't work?

- If no data appears in the table view, look in the Interface Builder Inspector to make sure that you have "Fetch on load" enabled for the Studio EODisplayGroup.

- If the buttons don't have the desired effect, check to see that they're connected to the appropriate action method in the appropriate object.

- If you get database errors when you try to add and delete studios or save changes, make sure that your model is properly specified. In particular, check that all of your entities have primary keys. Finally, choose Check Consistency from the Model menu in EOModeler to confirm that there are no problems in your model.

Optional Exercise

Enterprise Objects Framework provides additional action methods that you can use in connections: **fetch** (EODisplayGroup) and **refetch** (EOEditingContext). Try adding controls (such as buttons or menu items) to the application and connecting them to some of these action methods.

Until now you have still not written a single line of code. However, because of the built-in features of Enterprise Objects Framework, all of the following have been provided for you:

- Automatic primary key generation when you insert a new object

  As described in the section"Assigning Primary Keys" on page 45, every row in a database is uniquely identified by its primary key value. When you create a new object in your application and save it to the database, you're adding a new row to a database table, and this row needs a primary key (that is, it needs to have a unique value for the primary key attribute you set in EOModeler). Enterprise Objects Framework handles generating this unique value for you.

- Formatting of money and dates

- Coordinating the user interface with your data

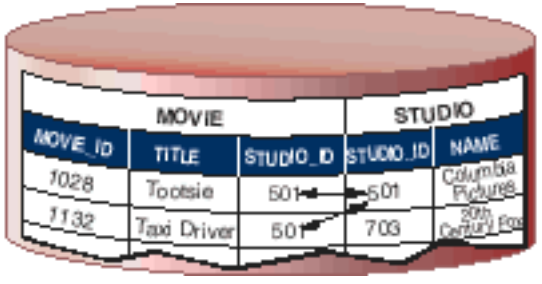
  Enterprise Objects Framework keeps all parts of an application synchronized with the current view of the data. For example, if you have two windows in an application that are displaying the same data and you change the values in one window, the other will automatically be updated to reflect the changes.

Adding Relationships

Creating an application that adds and modifies studios is just the first stage of the StudioManager application. Now you can enhance the application to display all of the movies owned by a selected studio.

The Studio, Movie, and Talent entities are not especially interesting when considered separately. Their real significance only becomes apparent in their relationships to each other. Every Movie has one corresponding Studio. One Studio can have many Movies. A particular actor (Talent) can star in several movies.

Relational databases model not just individual entities, but entities' relationships to one another. For example, a Movie entity has a corresponding Studio entity. This is modeled in the database by both the Movie entity and the Studio entity having a **studioID** attribute. In Movie, **studioID** is a foreign key, while in Studio it's a primary key. A foreign key correlates with the primary key of another table in order to model a relationship a source table (Movie) has to a destination table (Studio). In the following diagram, notice that the value in the **STUDIO_ID** column for both movies is "501". This matches the value in the **STUDIO_ID** column of the Columbia Pictures movie studio. In other words, the movies "Tootsie" and "Taxi Driver" both belong to Columbia Pictures.



This plays out in your running application as follows: Suppose you fetch a Movie object. Enterprise Objects Framework takes the value for the movie's **studioID** attribute and looks up the studio with the corresponding primary key.

For your application to take advantage of such database-defined relationships, your model must specify the corresponding relationships. When you created the model using the wizard, EOModeler created relationships between your selection of entities based on matching primary and foreign keys; it assigned

relationship names of the form "to*DestinationEntity*". You might have reason now to examine these relationships, add new ones, delete generated ones, or modify things such as whether a relationship is to-one or a to-many.

Note: Your model may already have some relationships in it, based on information EOModeler read from the database. For the purposes of this tutorial, you can just ignore these relationships.

You need to ensure that the following relationships are specified:

From the Studio (source) entity:

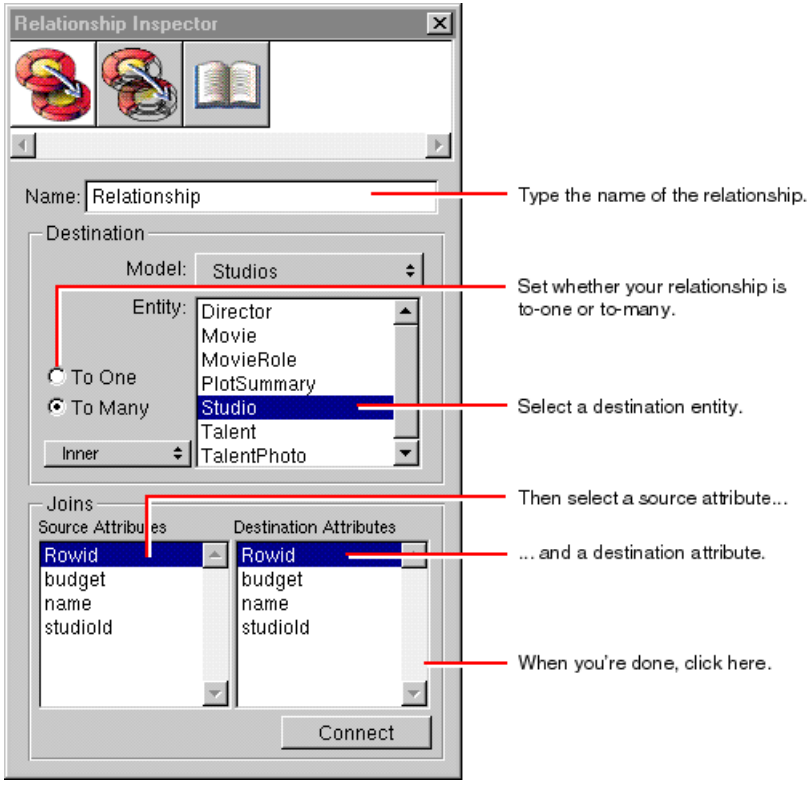
- Form a *to-many* relationship to the Movie (destination) entity.
- The source attribute is **studioID**. The destination attribute is **studioID**.
- Name the relationship **movies**.

From the Movie (source) entity:

- Form a *to-one* relationship to the Studio (destination) entity.
- The source attribute is **studioID**. The destination attribute is **studioID**.
- Name the relationship **studio**.

1. **Create a relationship.**

   Display the attributes view for the entity you want to use as the source of the relationship.
   Choose Property Add Relationship.
   In the Relationship Inspector, enter the name of the relationship.
   Select whether the relationship is to-one or to-many.
   Select a destination entity.
   Select a source attribute.
   Select a destination attribute.
   Connect them.



Adding Movies to the Application

The relationships you specified in EOModeler now come into play in your application. In EOModeler you added a to-many relationship from Studio to Movie, because a Studio can have many Movies. You can now use this relationship to display the movies for the selected studio.

In this type of configuration, called *master-detail*, the master table holds records for the source of the relationship, while the detail table holds records for the destination. As individual records in the master table are selected, the contents of the detail table change to show the records that correspond to the selection in the master. In the StudioManager application, Studio is the master table and Movie is the detail table.

Creating a Master-Detail Interface

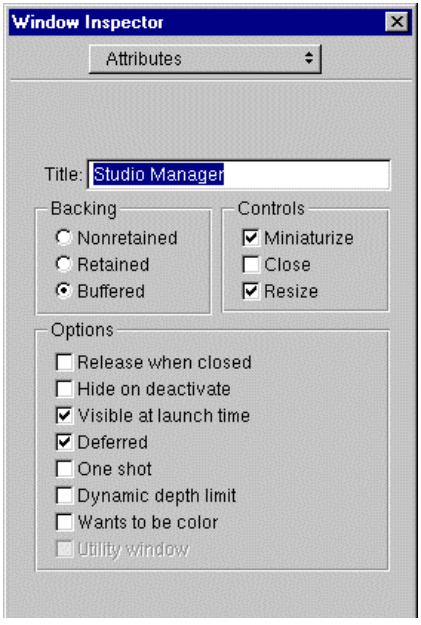
Starting with this exercise, you will create the final user interface of the StudioManager application. This means that you should start off by removing all objects added earlier to your nib file; also, give your application window a title.

2. Prepare the nib file.

In Interface Builder, delete the table view from the window.
Delete the Studio EODisplayGroup and the EditingContext from the nib file window.
Save the nib file.

3. Set the window title.

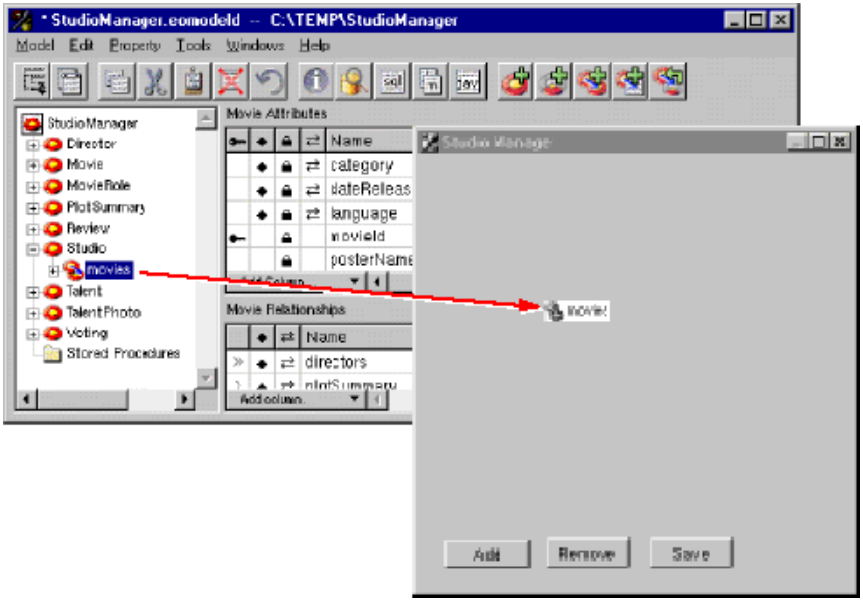
Select the window by clicking its title bar.
Choose Inspector from the Tools menu.
Enter “Studio Manager” in the TItle field of the Attributes display.



You can create a master-detail interface by simply dragging a relationship from EOModeler onto your window.

4. Create a master-detail interface.

Drag Studio's **movies** relationship from EOModeler onto the window in Interface Builder.
Rearrange and resize the tables so that they are next to each other.
Reconnect the Add and Remove buttons to the new Studio EODisplayGroup
Reconnect the Save button to File's Owner (the interface controller).
(See "Adding Action Methods" on page 55 for these procedures.)

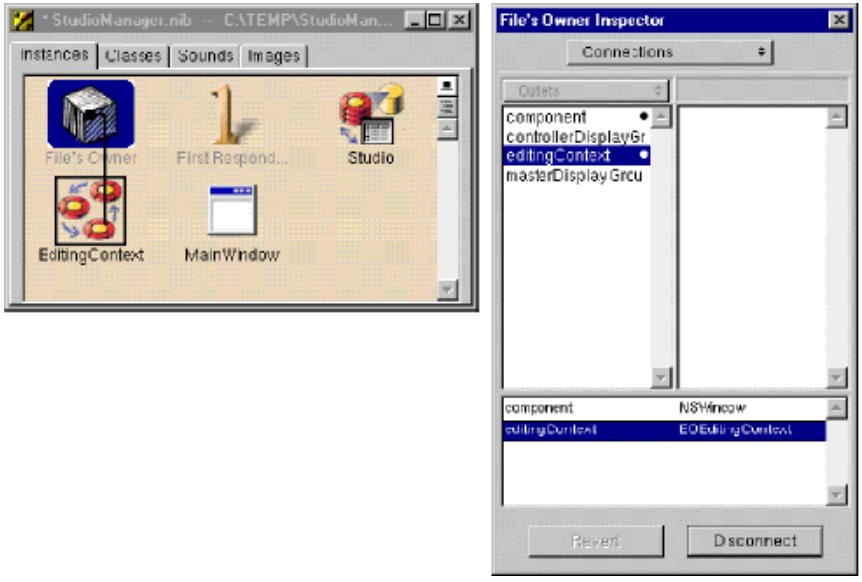


This operation creates a master-detail interface. Columns are automatically added for all of the attributes marked as class properties; you can delete any columns you don't want.

For a Java Client application, the interface controller—represented by the File's Owner icon in the nib file window—is the "controller" object in the Model-View-Controller design scheme. The interface controller comes already connected to its "view" through the component outlet. But you must connect it to its "model" object, the editing context.

5. **Connect the interface controller to its editing context and display group.**

   Control-drag from File's Owner to the EditingContext icon in the nib file window.
   In the Connections inspector, select the **editingContext** outlet.
   Click Connect.

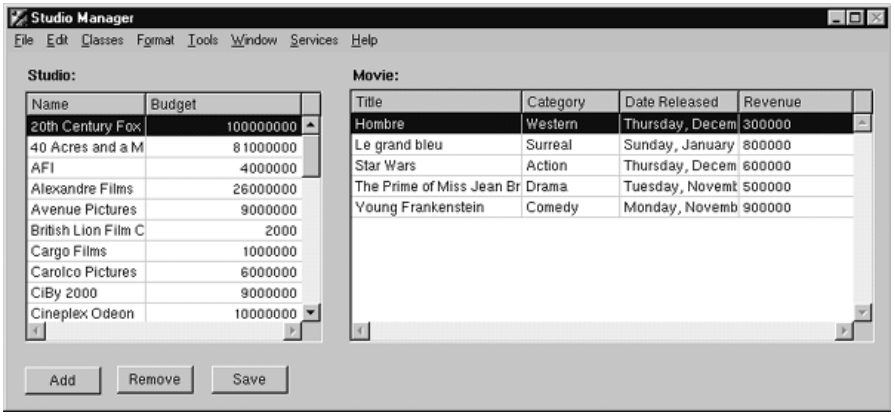


   Control-drag from File's Owner to the Studio icon in the nib file window.
   In the Connections inspector, select the **displayGroup** outlet.
   Click Connect.

In the following figure, you can see the master-detail interface in action. Notice that the table views have been rearranged, and that titles have been added above the Studios and Movies tables.

6. Test your interface.

Choose File Test Interface.

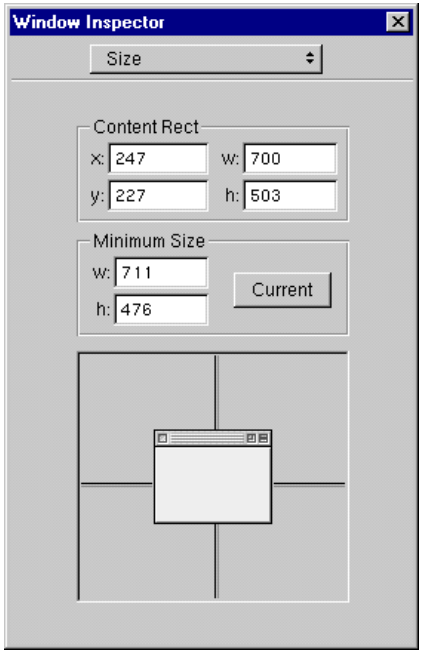


7. Use Project Builder to build the application.

To see the effects of your changes, you must compile and run the application. However, before you run the application, there is one last step to perform. The applet providing the running environment for your Java Client application is set to a default size in the WOJavaClientApplet bindings in **Main.wod**. This size could be too small to accommodate your user interface (or too large for it).

8. **Learn the size of the window.**

   Open **StudioManager.nib.**
   Choose Inspector from the Tools menu.
   Select the Size inspector.
   Write down the **w** and **h** parameters.



For example's sake, let's assume the window is 503 pixels high and 700 pixels wide. Transfer these numbers to the WOJavaClientApplet.

9. **Set the WOJavaClientApplet size bindings.**

   Open **Main.wod.**
   Enter the dimensions of the window in the height and width bindings.
   Save the file.

```
Applet: WOJavaClientApplet {
    height = 503; // change this
    width = 700; // and this
    interfaceControllerClassName =
“studiomanager.client.StudioManager”;
    useJavaPlugin = NO;
}
```

Now you are ready to test the application.

10. Run and test the application.

Choose Tools Launcher Run to launch the server side of the application.

Start up the client side (see “Running a Java Client Application” on page 59).

When you select a studio in the Studios table view, the display changes in the Movies table view to show the selected studio’s movies.

Transferring Movies Between Studios

One of the primary functions of the StudioManager application is to allow one studio to purchase movies from another. To make this possible, you’ll now add a pop-up list to the user interface.

The pop-up list displays a list of all of the studio titles. When you select a new studio in the pop-up list, you cause that studio to purchase the movie that’s selected in the table view.

1. Add a pop-up list.

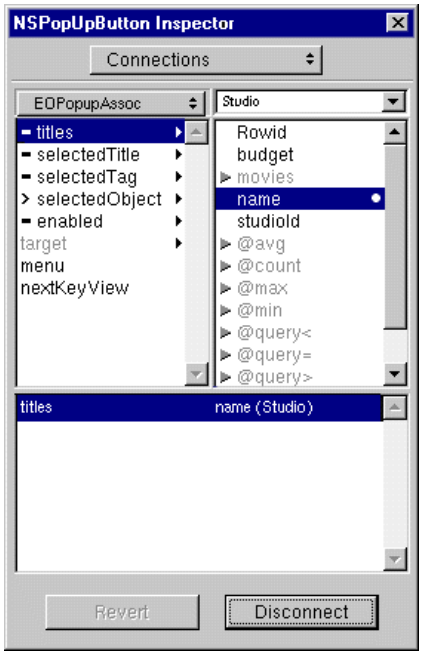
Drag a pop-up list (labeled “Item” on the Views palette) into the window.

Control-drag from the pop-up list to the Studio EODisplayGroup.

In the Inspector, select EOPopupAssoc from the pop-up list at the top of the left column.

Select **titles** in the left column. The **titles** aspect is bound to the class key whose values you want to display in the pop-up list.

Select **name** in the right column (since you want to display Studio names in the pop-up list).



Put the pop-up list directly below the Studio table view and leave some space between it and the row of buttons. Later you will be adding fields between the pop-up list and the buttons. For a guide, see the figure associated with step 2, "Test your interface and try out the new pop-up list."

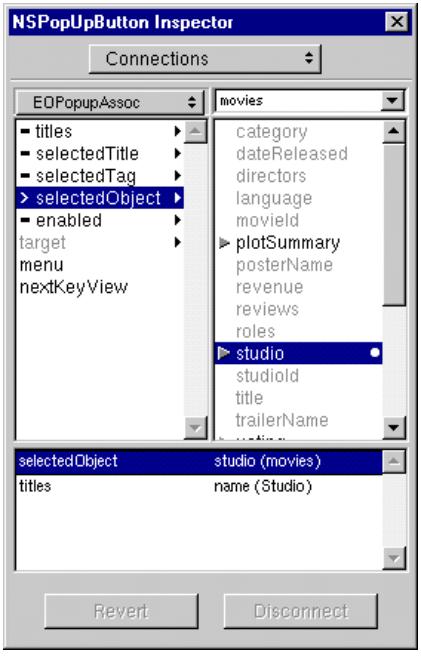
Now you have to add another binding to the EOPopupAssociation so that when you change the selected studio title, it sets the corresponding **studio** relationship property in the selected Movie object.

Control-drag from the pop-up list to the movies EODisplayGroup.

In the Inspector, select EOPopupAssoc from the pop-up list at the top of the left column.

Select **selectedObject** in the left column.

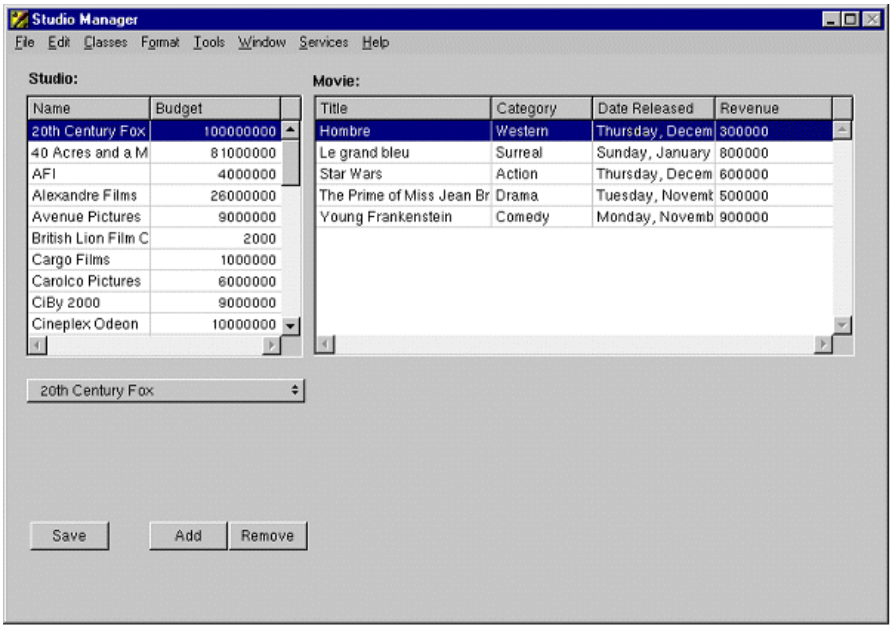
Select **studio** in the right column.



The **selectedObject** aspect is bound to the relationship property (in this example, Movie's **studio** property) that corresponds to the object bound to the **titles** aspect (Studio).

2. **Test your interface and try out the new pop-up list.**

   Choose File Test Interface.



3. **Build and test-run the application.**

   (See “Building and Testing Your Application” for details.)

You can now test the behavior of the pop-up list. For example, suppose you want to transfer the movie “Alien” from the 20th Century Fox studio to MGM. First select 20th Century Fox to display its movies. Then select “Alien” in the list of movies. Finally, use the pop-up list to change the selected studio from 20th Century Fox to MGM. This has the effect of removing “Alien” from 20th Century Fox’s **movies** relationship array and adding it to the **movies** relationship array of MGM. It also sets the “Alien” Movie object’s **studio** relationship property to point to the new studio, MGM. When you use the pop-up list to transfer a movie, you’ll notice that the movie disappears from the original studio’s movie list and reappears in the movie list of the new studio.

These changes aren’t committed to the database until you click Save. At that time Enterprise Objects Framework translates the changes you made in the object graph into the appropriate database changes. For example, it sets the

foreign key **studioID** in the transferred Movie object to have the same value as the **studioID** primary key of its new studio.

Note that Enterprise Objects Framework manages all of this for you without requiring you to write any code.

Related Concepts: What is an Association?

Putting the Finishing Touches on Your Model

You are almost ready to add custom behavior to your enterprise objects. But first you need to put a few finishing touches on your model.

In"Adding Relationships" on page 64, you added relationships between the Studio and Movie entities. Now you need to verify or add a few additional relationships to your model. You might find that the relationship already exists, but just the name needs to be changed:

From the Movie (source) entity:

- Form a *to-many* relationship to the MovieRole (destination) entity.
- The source attribute is **movieID**. The destination attribute is **movieID**.
- Name the relationship **roles**.

- Form a *to-one* relationship to the PlotSummary (destination) entity.
- The source attribute is **movieID**. The destination attribute is **movieID**.
- Name the relationship **plotSummary**.

From the Talent (source) entity:

- Form a *to-many* relationship to the MovieRole (destination) entity.
- The source attribute is **talentID**. The destination attribute is **talentID**.
- Name the relationship **roles**.

- Form a *to-one* relationship to the TalentPhoto (destination) entity.
- The source attribute is **talentID**. The destination attribute is **talentID**.
- Name the relationship **photo**.

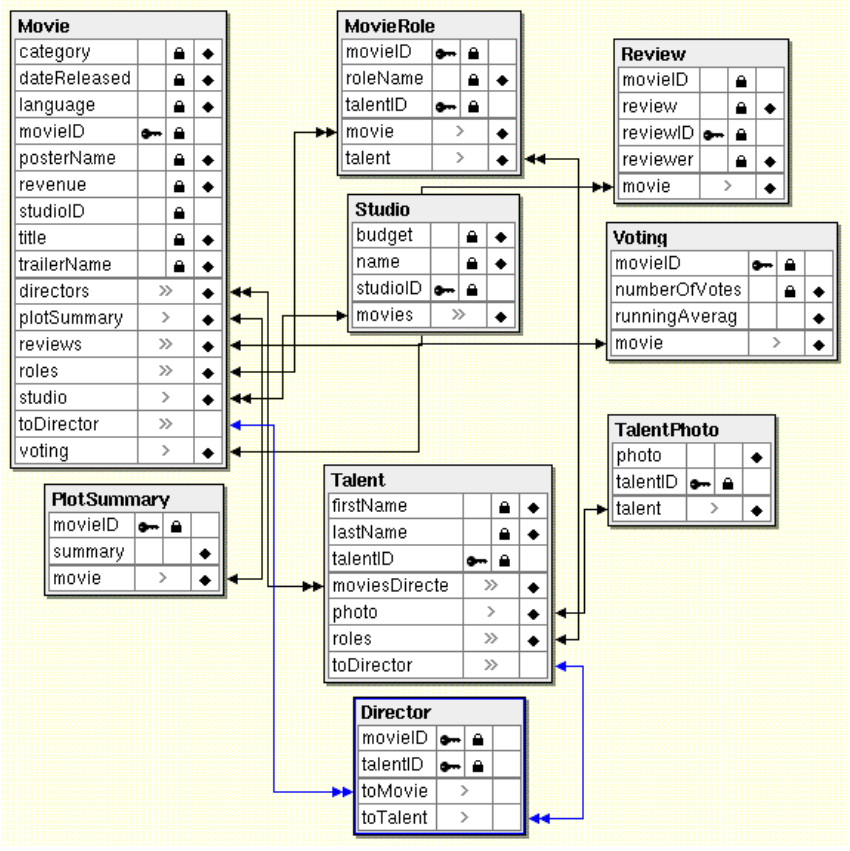
From the MovieRole (source) entity:

- Form a *to-one* relationship to the Movie (destination) entity.

- The source attribute is **movieID**. The destination attribute is **movieID**.
- Name the relationship **movie**.

- Form a *to-one* relationship to the Talent (destination) entity.
- The source attribute is **talentID**. The destination attribute is **talentID**.
- Name the relationship **talent**.

At this point your model is complete. There might be other relationships in your model, but the above relationships are the most important for our example project. Looking at your model using the Diagram View (select the model icon and choose Tools Diagram View) gives you an overview of the entities in the model and their relationships to other entities.



Adding Behavior to Your Enterprise Objects

As the preceding sections illustrate, you can go quite far in a Java Client application without writing any code. However, the real power of such an application or any Enterprise Objects Framework application lies in the enterprise objects you create. The behavior (business logic) you add to your objects is what brings your stored data to life.

Specifying Custom Enterprise Object Classes

If you create the model earlier with the help of the wizard, and choose the "Create Custom Enterprise objects" option, EOModeler derives both entity name and class name from the name of the associated database table. Otherwise, EOModeler maps entities to the EOGenericRecord class, which can be thought of as the default enterprise object class.

The EOGenericRecord class is sufficient when all you want the entity to do is get and set properties. However, when you want to add custom behavior to a class (for example, to assign default values when you create new objects or to perform validation), you need to implement a custom enterprise object class. This class includes the default behavior provided in EOGenericRecord as well as the custom behavior you implement.

1. **Specify custom enterprise object classes for the server and the client.**

   In the Model Editor, select the model (StudioManager).

   If the Client-Side Class Name column is not visible, select Client-Side Class Name from the Add Column pull-down list at the bottom of the window.

   Select the Studio entity in the table.

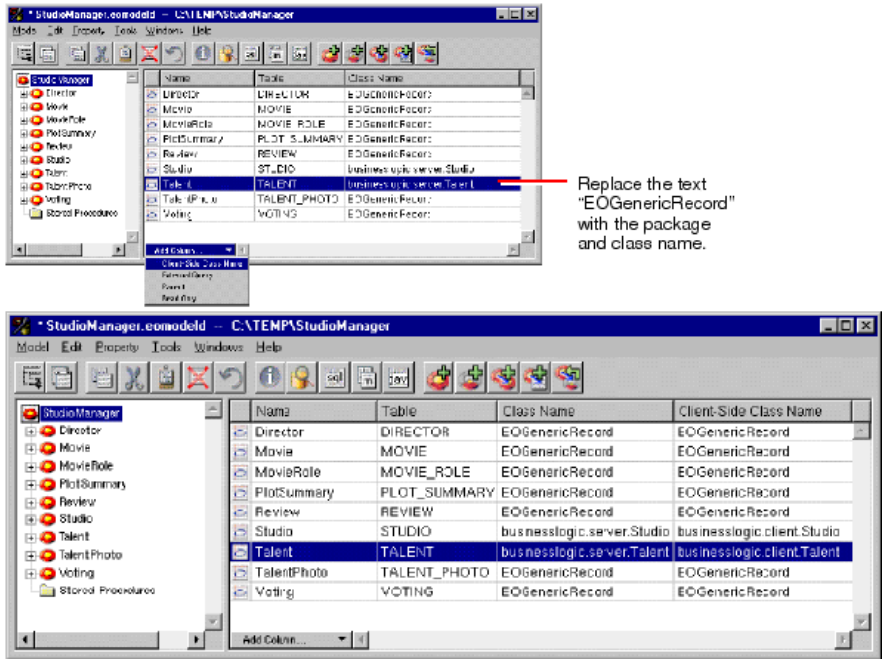
   Double-click the Studio cell under Class Name.

   Type "businesslogic.server.Studio" in the cell ("businesslogic.server" is the package name).

   Double-click the adjoining cell under the Client-Side Class Name column.

   Type "businesslogic.client.Studio" in this cell ("businesslogic.client" is the package name).

   Repeat the above steps for the Talent entity (append "Talent" to the package names).



For the StudioManager application, ensure that there are custom classes (with their package prefixes) corresponding to the appropriate entity; these classes should be named **businesslogic.server.Studio** and **businesslogic.server.Talent** under Class Name and **businesslogic.client.Studio** and **businesslogic.client.Talent** under Client-Side Class Name. Movie doesn't need to be a custom class since it doesn't have any specialized behavior. By convention, the names of classes (minus the package prefix) are based on the name of the corresponding entity and the initial letter of the name is capitalized.

There is no requirement that you create matching server and client classes. You can implement a class only on the server or the client, whichever suits your needs; the unimplemented class assumes the default behavior of EOGenericRecord.

Once you specify a custom class for an entity in EOModeler, you can generate source files for that entity.

Related Concepts: When Do You Use a Custom Enterprise Object Class?

Generating Source Files

To begin creating your custom classes, generate source files for the Studio and Talent entities. You'll use these source files as a basis for adding custom behavior to your enterprise objects. Generating source files in a Java Client application typically produces "skeletal" **.java** files for the associated class. These files are put in the **ClientSideJava.subproj** subproject.

Note: To generate source files for an entity, you must have replaced the text "EOGenericRecord" in the Class Name and Client-Side Class Name fields with a package name concatenated with a class name.

1. **Generate source files.**

   In the Model Editor, select the entity for which you want to generate source files.

   Choose Property ▸ Generate Client Java File.

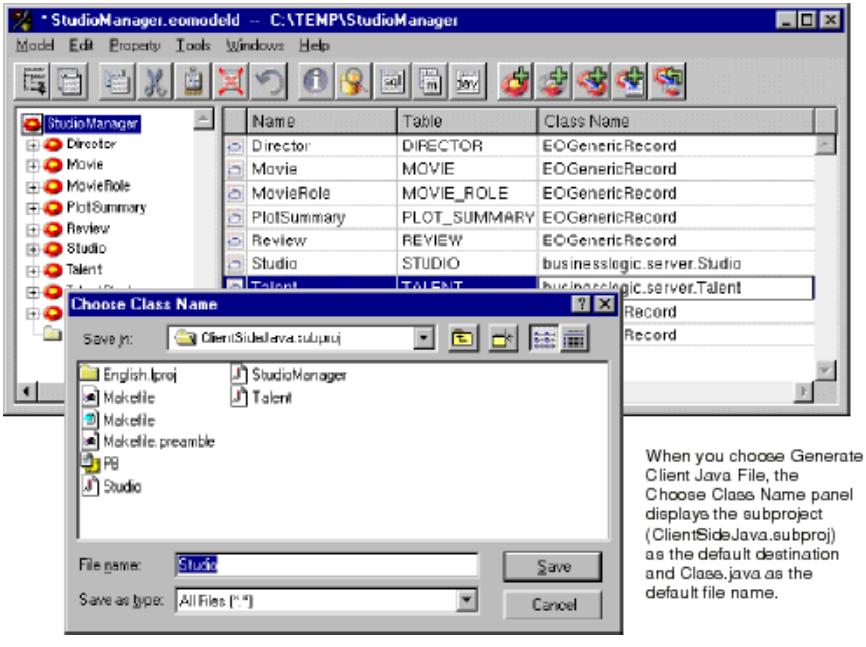
   In the Choose Class Name panel verify the file name and location (**ClientSideJava.subproj**) and click Save.

   Click OK when you're asked if you want to insert the files in the subproject.

   For the same entity, choose Property ▸ Generate Java FIle.

   In the Choose Class Name panel verify the file name and location (main project) and click Save.

   Click OK when you're asked if you want to insert the files in the main project.



When you choose Generate Client Java File, the Choose Class Name panel displays the subproject (ClientSideJava.subproj) as the default destination and Class.java as the default file name.

When Project Builder generates a class file (such as **Studio.java),** it strips off the package prefix and inserts a package declaration near the top of the file. The class file also includes the necessary import declarations as well as the instance variables and accessor methods derived from the properties of the Studio entity.

Studio.java (ClientSideJava.subproj)

```
package businesslogic.client;

import com.apple.client.foundation.*;
import com.apple.client.eocontrol.*;
import java.util.*;
import java.math.BigDecimal;

public class Studio extends EOGenericRecord {

    public static final String BudgetKey = "budget";
    public static final String NameKey = "name";
    public static final String MoviesKey = "movies";

    public Studio(EOEditingContext context, EOClassDescription
        classDesc, EOGlobalID gid) {
        super(context, classDesc, gid);
    }

    public String name() {
        return (String)storedValueForKey(NameKey);
    }

    public void setName(String value) {
        takeStoredValueForKey(value,NameKey);
    }

    public Number budget() {
        return (Number)storedValueForKey(BudgetKey);
    }

    public void setBudget(Number value) {
        takeStoredValueForKey(value,BudgetKey);
    }

    public NSArray movies() {
        return (NSArray)storedValueForKey(MoviesKey);
    }
```

```
    public void setMovies(NSMutableArray value) {
        takeStoredValueForKey(value,MoviesKey);
    }

    public void addToMovies(EOEnterpriseObject object) {
        NSMutableArray movies;
        movies = (NSMutableArray)storedValueForKey(MoviesKey);
        willChange();
        movies.addObject(object);
    }

    public void removeFromMovies(EOEnterpriseObject object) {
        NSMutableArray movies;
        movies = (NSMutableArray)storedValueForKey(MoviesKey);
        willChange();
        movies.removeObject(object);
    }


    public void buyAllMoviesStarringTalent(Talent talent) {
         invokeRemoteMethod
          ("clientSideRequestBuyAllMoviesStarringTalent",
          new Object[] {talent});
    }
}
```

Implementing Custom Behavior for Your Classes

The user interface you designed in Interface Builder already allows you to insert and delete Studio objects. However, it doesn't do any additional processing when these operations take place. For example, what if you want to assign default values to newly created objects? And how can you prevent users from inserting objects that contain invalid data? You can add methods to your enterprise objects to handle such issues.

Related Concepts: Adding Behavior to Enterprise Objects

Distributing Business Logic in Java Client Applications

The value of Java Client applications, of course, lies in their ability to distribute processing duties among objects on the server and objects on the client. Primarily for security and performance reasons, you can have only

objects on the server performing some tasks and only objects on the client performing others.

For example, sometimes you want only objects behind the firewalls and other security mechanisms of the server to have access to sensitive information, such as account numbers. On the other hand, processing tasks such as calculation of balances should be performed by objects on the client, thereby improving application performance by eliminating the need for a cycle of the request-response loop.

There are no hard and fast rules for how to distribute object behavior. An enterprise object on the client can have the same set of methods and instance variables as its counterpart on the server, or what it has can be a subset (or superset) of the other object's methods and instance variables. The best way to distribute business logic among objects depends on the particular nature of your application.

Writing Derived Methods

One kind of behavior you might want to add to your enterprise object class is the ability to perform computations based on the values of class properties. For example, studios have movies, and the total revenue of the movies times 1.5 constitutes the studio's portfolio value. To calculate a studio's portfolio value, you could have a method in **Studio.java** like the following:

Studio.java

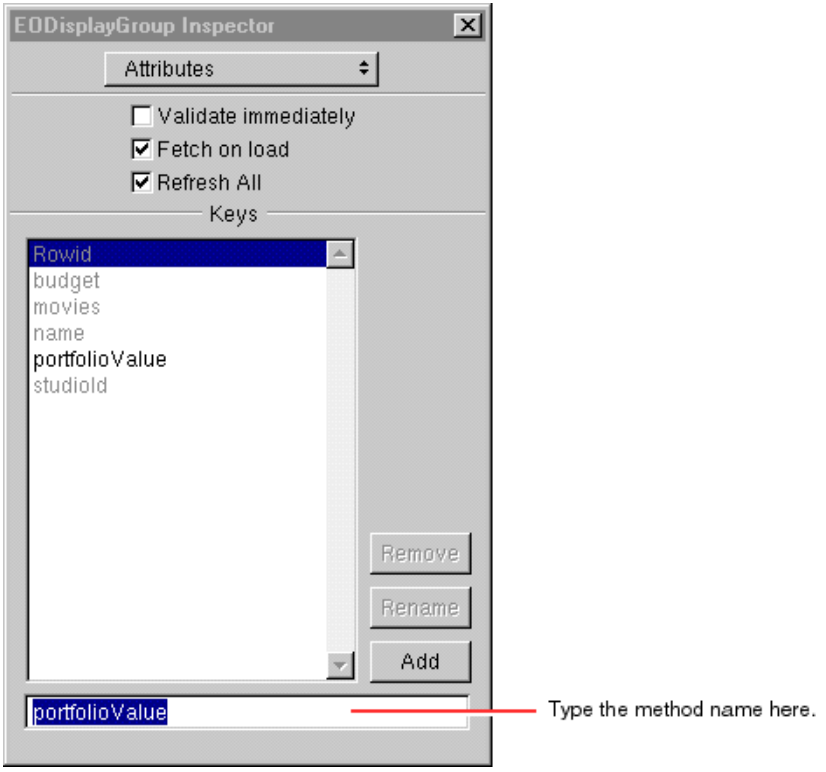
```
public Number portfolioValue() {
    int i,count;
    double total;
    NSArray revenues;

    total = 0.0;
    revenues = (NSArray)(movies().valueForKey("revenue"));

    count = revenues.count();
    for (i=0; i<count; i++) {
        total +=
           ((Number)(revenues.objectAtIndex(i))).doubleValue();
    }
    return new BigDecimal(total * 1.5);
}
```

You can display the results of this method in the user interface by forming an association between a control and the method. That way, whenever a new studio is selected or when a selected studio's movie revenues change, its portfolio value is dynamically recalculated and displayed.

1. **Add the code above to the client-side Studio.java file.**

2. **Add a method as a display-group property.**

   Display the Attributes view of the Inspector for the Studio EODisplayGroup.
   Add the name of the method (**portfolioValue**) you want to use in an association.
   Click Add.



Once you've added the method as a class key, you can use it in associations. But before you do this, add the necessary user-interface control.

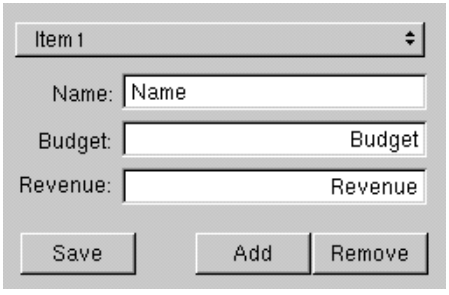
3. Add text fields to the user interface.

Drag three text fields from the Views palette.

Make them the same size and align them in a column.

Add labels (as shown at right) to each text field.

Justify the fields' contents (as shown).



Now make an association between the Revenue text field and the **portfolioValue** method.

4. Associate a method with a user interface control.

Control-drag from the Revenue text field to the Studio EODisplayGroup.

In the Connections Inspector, choose EOControlAssoc from the pop-up list at the top of the left column.

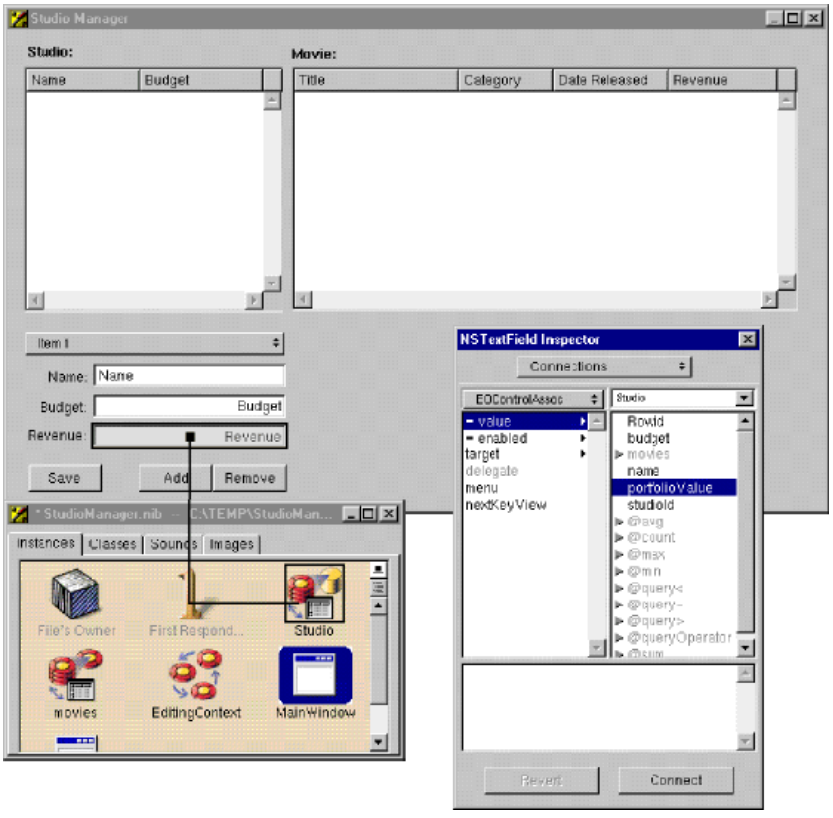
Select **value** in the left column.

In the right column select the method (**portfolioValue**) you want to associate with the control.

Double-click **portfolioValue** to connect.

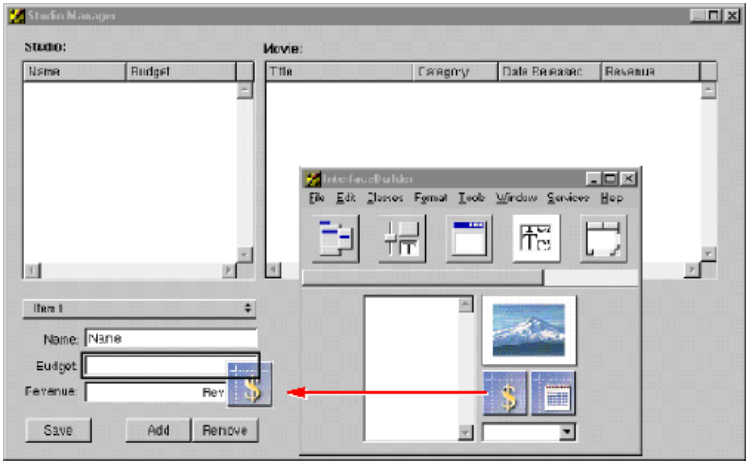
Repeat the above steps, connecting the Name field to Studio's **name** attribute and the

Budget field to the **budget** attribute.



You now need to add a formatter to the Revenue and Budget fields. The formatter isn't added automatically, because the field has no way of knowing that it's going to be used to display currency values—it's just connected to a property.

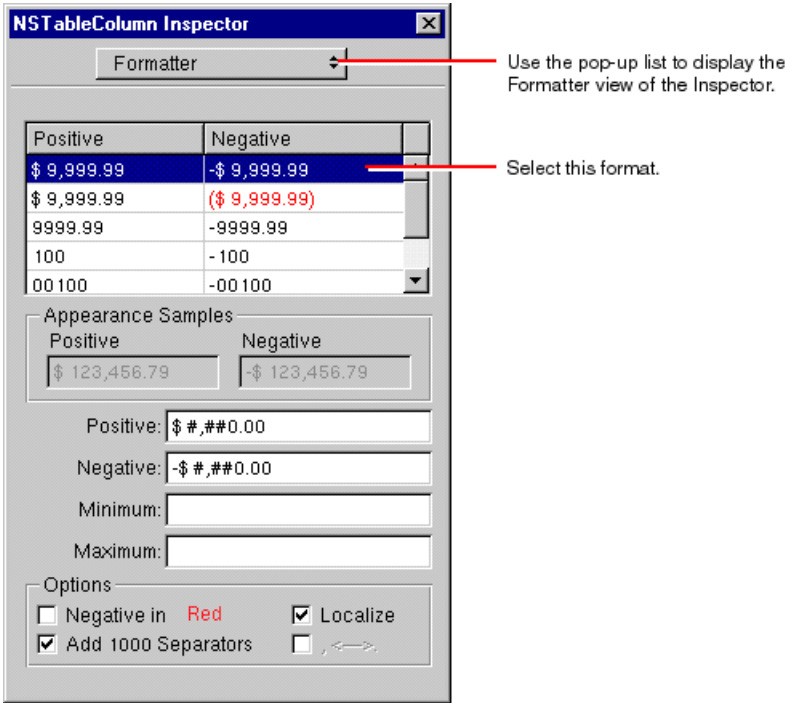
5. **From the DataViews palette, drag the currency formatter into the new text field.**



Once you've added the formatter, you can use the Inspector to change the format.

6. **Set the format.**

Select the text field, and display the Formatter view of the NSTextField Inspector. Change the format as shown.



7. **Build and test the application on the client.**

   (See "Building and Testing Your Application" for details.)

Performing Validation

Another element you'll likely want to add to your enterprise object classes is validation. For example, suppose that when a studio buys a new movie, you want to check to make sure that acquiring the movie won't cause the studio to exceed its budget. You could implement a method in the Studio class like the following:

Studio.java (server and client)

```
public void validateBudget(Number budget) throws
    EOValidation.Exception {
    if (budget.intValue() < 100) {
        throw new EOValidation.Exception
           ("A budget cannot be less than $100");
    }
}
```

Now when a studio buys more movies than it can afford, a panel displaying the message "A budget cannot be less than $100" appears when the user attempts to save the changes to the database.

Validation methods must be of the form **validate*Attribute***. The **validateBudget** method is invoked by the **validateValueForKey** method, which is part of the EOValidation interface that uses the EOClassDescription class to provide default implementations of validation methods. These methods are invoked automatically by framework components such as EODisplayGroup and EOEditingContext. They are:

- validateValueForKey
- validateForSave
- validateForDelete
- validateForInsert
- validateForUpdate

For more discussion of this topic, see the chapter "Designing Enterprise Objects" in the *Enterprise Objects Framework Developer's Guide* and the NSObject Additions class specification in the *Enterprise Objects Framework Reference*.

Providing Default Values for Newly Inserted Objects

When new objects are created in your application and inserted into the database, it's common to assign default values to some of their properties. For example, you might decide to assign newly created Studio objects a default budget (the budget is the amount a studio is allowed to spend on new movies).

To assign default values to newly created enterprise objects, use the method **awakeFromInsertion**. This method is automatically invoked right after your enterprise object class creates a new object and inserts it into an EOEditingContext.

The following implementation of **awakeFromInsertion** in the Studio class sets the default value of the **budget** property to be one million dollars:

Studio.java (server and client)

```
public void awakeFromInsertion(EOEditingContext ec) {
    super.awakeFromInsertion(ec);
    if (budget() == null)
        setBudget(new BigDecimal("1000000"));
}
```

When a user clicks the Add Studio button in the StudioManager application, a new record is inserted, with "$1,000,000.00" already displayed as a value in the **budget** column.

Invoking Server Methods Remotely

In a Java Client application you may want some methods to execute only on the server. This is particularly the case when security is an issue, but performance can be a reason as well (as when the method consumes a lot of system resources). Objects on the client side of a Java Client application can use two methods to invoke a server method:

- **invokeRemoteMethod**. An enterprise object on the client side can use this method to invoke a method in the corresponding enterprise object on the server. The arguments are the name of the method to invoke and an array of arguments. Before the method is invoked on the server, the current state of the client-side editing context is "pushed" to the server to ensure that the method executes in an identical context. (Note that EODistributedObjectStore has a version of this method that includes a flag as an argument; setting this flag to **false** prevents the client from pushing its editing-context state to the server.)

- **invokeRemoteMethodWithKeyPath**. You can send a message to *any* object on the server with this method, which is defined in EODistributedObjectStore. For more on this method, see the specification for this EODistribution class.

In our StudioManager example, let's say that you want to give studios the ability to buy all of the movies that star a specified actor, but you consider this a sensitive computation. You can implement a method such as the following in **Studio.java**:

Studio.java (client)

```
public void buyAllMoviesStarringTalent(Talent talent) {

invokeRemoteMethod("clientSideRequestBuyAllMoviesStarringTalent",
    new Object[] {talent});
}
```

The method begins with "clientSideRequest"; this is not accidental. The EODistributionContext object on the server-side EODistribution layer will reject a remote invocation unless it has this prefix *or* its delegate implements the proper delegation methods (see the reference documentation for EODistributionContext or EODistributedObjectStore for more information).

The following is the invoked method, which is implemented in the server's **Studio.java**:

Studio.java (server)

```
public void clientSideRequestBuyAllMoviesStarringTalent(Talent
talent) {
    int i, count;
    NSArray talentMovies;
    EOEnterpriseObject movie, studio;

    talentMovies = talent.moviesStarredIn();
    count = talentMovies.count();
    for (i = 0; i < count; i++) {
        movie =
           (EOEnterpriseObject)(talentMovies.objectAtIndex(i));
        if (!(movies().containsObject(movie))) {
           studio =
               (EOEnterpriseObject)(movie.valueForKey("studio"));
           if (studio != null)
               studio.
```

```
            removeObjectFromBothSidesOfRelationshipWithKey
            (movie,"movies");
         addObjectToBothSidesOfRelationshipWithKey
            (movie,"movies");
      }
   }
}
```

This method invokes the **moviesStarredIn** method:

Talent.java (server)

```
public NSArray moviesStarredIn() {
       int i, count;
       NSArray movies;
       NSMutableArray moviesStarredIn;
       EOEnterpriseObject movie;

       moviesStarredIn = new NSMutableArray();
       movies = (NSArray)(roles().valueForKey("movie"));

       count = movies.count();
       for (i = 0; i < count; i++) {
          movie = (EOEnterpriseObject)(movies.objectAtIndex(i));
          if (!(moviesStarredIn.containsObject(movie))) {
              moviesStarredIn.addObject(movie);
          }
       }
       return moviesStarredIn;
}
```

You can associate the **buyAllMoviesStarringTalent** method with a user interface control. But first you need to add to your user interface a table view that lists all actors (talent).

8. Add a new table view to your user interface.

Drag the Talent entity from your model into the nib file window in Interface Builder.

Drag a table view from the Palette onto your window.

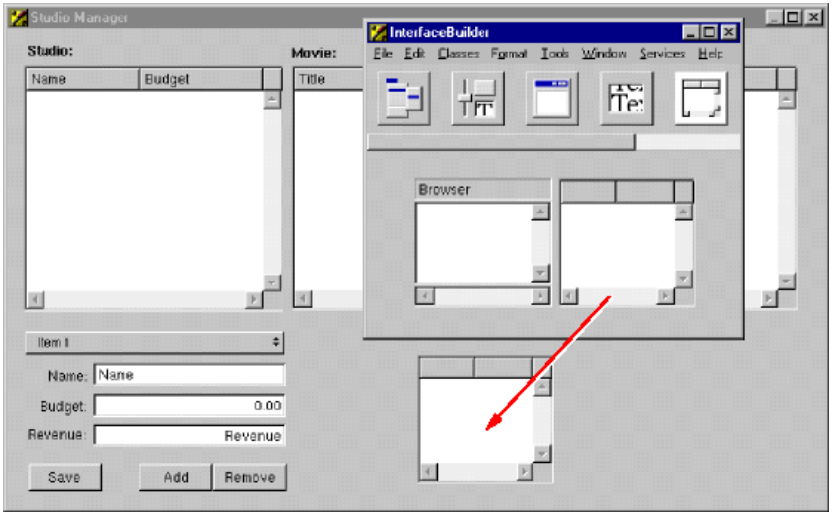
Control-drag from each table view column to the Talent EODisplayGroup.

Using the **value** aspect of the EOColumnAssoc, connect the table view columns to the

firstName and **lastName** class keys, respectively.

9. Add a button to the window.

Drag a button into the window.
Place it below the Revenue field.
Resize it.
Give it the title “Buy Movies Starring Selected Talent”.



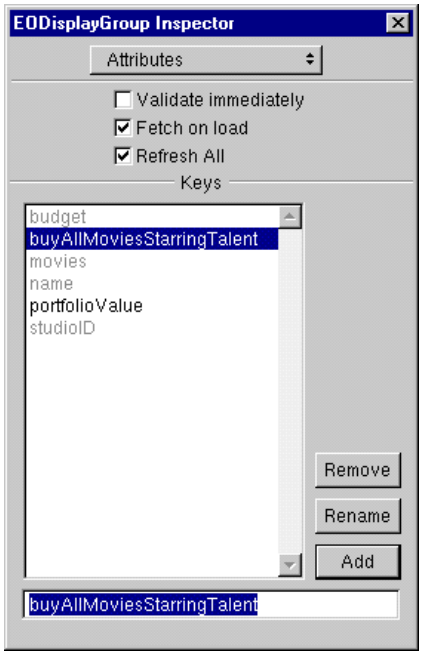
Now that you’ve added the table view, connected it to the **firstName** and **lastName** properties of the Talent EODisplayGroup, and added a Buy button to the window, you’re ready to use an EOActionAssociation to connect the button to the **buyAllMoviesStarringTalent** method.

10. Associate a method with a user interface control.

Display the Attributes view of the Inspector for the Studio EODisplayGroup.

In the text field type the name of the method (**buyAllMoviesStarringTalent**) you want to use in an association.

Click Add.



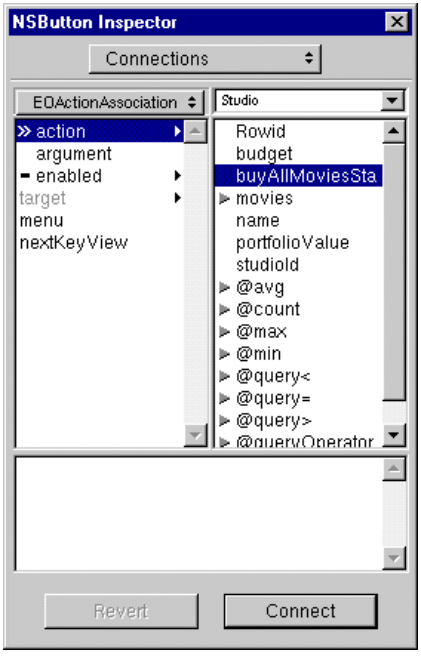
You can now use the **buyAllMoviesStarringTalent** method in associations.

Control-drag from the “Buy Movies Starring Selected Talent” button to the Studio EODisplayGroup.

In the Connections Inspector, choose EOActionAssociation from the pop-up list at the top of the left column.

Select **action** in the left column, and the method you want to connect to (**buyAllMoviesStarringTalent**) in the right column.

Click Connect.

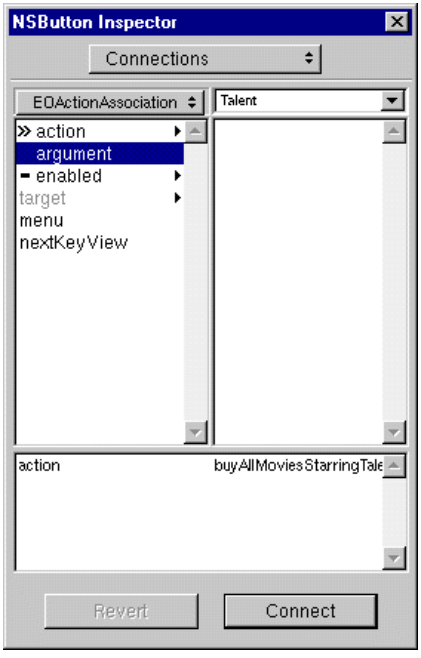


Because the **buyAllMoviesStarringTalent** method takes a Talent object as an argument, you also need to make a connection from the Buy button to the Talent EODisplayGroup.

Control-drag from the “Buy Movies Starring Selected Talent” button to the Talent EODisplayGroup.

In the Inspector, select **argument** in the left column. The **argument** aspect takes the destination of the connection (Talent) as an argument, which will be supplied to the **buyAllMoviesStarringTalent** method.

Click Connect.



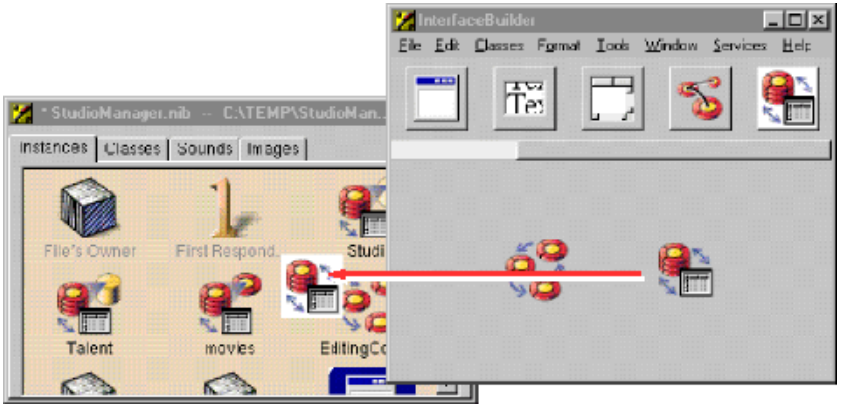
Once you finish connecting the button, you can use it to purchase all of the movies starring the selected actor for the selected studio.

Controlling the User Interface

In Java Client applications you can give the interface controller (implemented in this project in **StudioManager.java** on the client) a *controller display group*. By creating associations between the controller display group and aspects of user-interface objects, you can use the interface controller to manage various facets of the user interface. In the following steps, you add a method as a property of the controller display group and bind this method to the **enabled** aspect of the Revenue field through an EOControlAssociation; since this method simply returns **false**, the field is disabled.

1. **Add a display group to the nib file.**

   Drag a display group from the EOPalette to the nib file window.
   Double click the title of the display group to select it.
   Give the display group the name “Controller”.



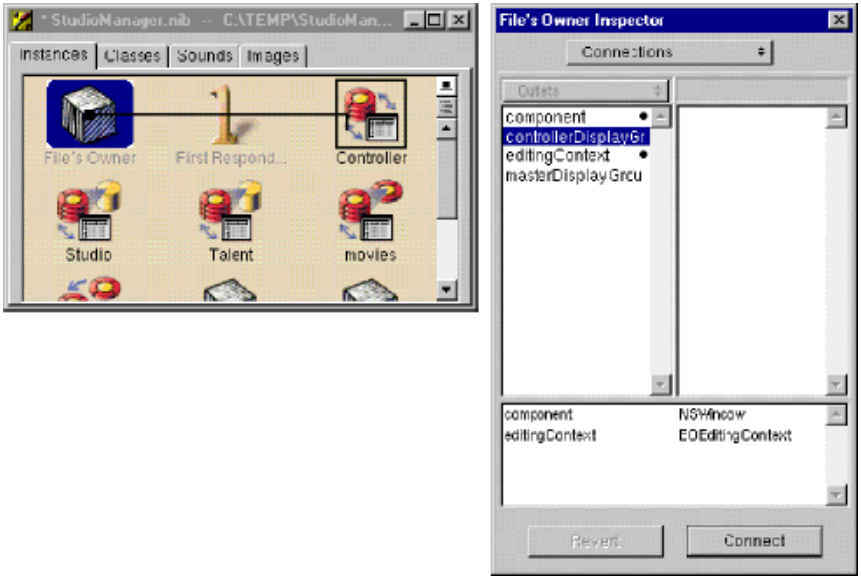
As mentioned earlier, the owner of the nib file (File’s Owner) is an instance of the custom EOInterfaceController automatically created by Project Builder. EOIntefaceController has a **controllerDisplayGroup** outlet; in the following step, connect the interface controller to this outlet.

2. **Connect the interface controller to its display group.**

   Control-drag from File's Owner to the Controller icon.

   In the Connections inspector, select **controllerDisplayGroup**.

   Click Connect.



Next add the **neverEnabled** method as a property of the controller display group.

3. Add a property to the controller display group.

Select the Controller display group in the nib file.
In the Attributes inspector, enter “neverEnabled” in the field.
Click Add.

Now hook up the field to the display group using an EOControlAssocation to bind its **enabled** aspect to the **neverEnabled** method.

4. **Connect the field's enabled aspect to the display group property.**

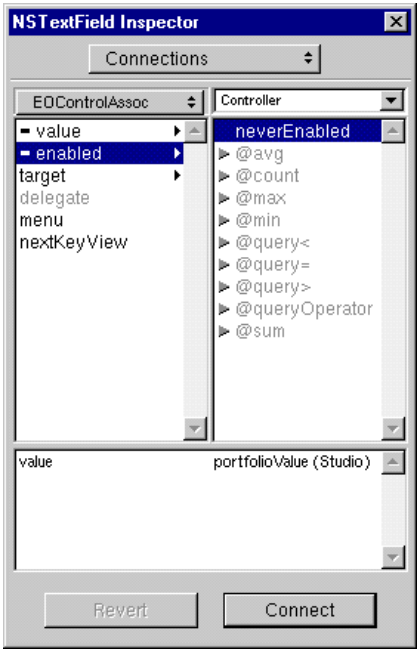
   Control-drag from the Revenue field to the Controller display group.

   In the Connections inspector, select EOControlAssoc from the pop-up list at the top of the left column.

   Select **enabled** in the left column.

   Select **neverEnabled** in the right column.

   Click OK.



5. **Implement the neverEnabled method.**

Now that the interface controller, the controller display group, and the Revenue field are interconnected via their outlets and associations, you can implement the method bound to the **enabled** aspect (in **StudioManager.Java** on the client).

```
 public boolean neverEnabled() {
        return false;
}
```

6. **Build, run, and test the application.**

Build the project and test the application. The Revenue field has a gray background and cannot be written into.

Chapter 3

Advanced Tasks

Debugging Java Client WebObjects Applications

It can be difficult to debug Java Client WebObjects applications because these applications have a client side and a server side. Each side runs in a totally different process and in a different virtual machine (VM), so you can't debug the one side by running a debugger for the other side.

Debugging Server Code

To debug the server side of a Java Client application use the standard debugging features of Project Builder. Open the launch panel for your application, specify necessary launch options, and start the debugger by clicking the debug button (the spray-can icon) in the launch panel. You can use the Launch panel to perform debugging tasks in all your server side classes.

See the documentation for Project Builder for details on its debugging features.

Debugging Client Code

Project Builder currently provides no support for debugging the client side of a Java Client application. Instead, use the Java debugger **jdb** (included with the JDK) in a shell window.

Before you can debug client code, compile your Java classes with the `-g` flag specified. To do this, either "make debug" your project or enter `OTHER_JAVAC_FLAGS=-g` as a build argument in Project Builder's Build Options panel.

Once your code has compiled, start up the client application with **appletviewer** or with the **java** interpreter (see "Running a Java Client Application" in the tutorial) with the `-debug` flag . These tools then print a "password" that you can use later to attach **jdb** to your client application. To attach **jdb**, open another shell and enter the following command:

```
jdb -password password
```

Please refer to the **jdb** documentation for information on setting breakpoints and performing other debugging tasks. As with running an application, your CLASSPATH environment variable has to specify the location of all Java classes used in your application.

If you don't want to attach to a running client application, you can start up **jdb** using **appletviewer** or the interpreter through a class name, for example:

```
jdb sun.applet.AppletViewer URL
```

```
jdb com.apple.client.eointerface.EOApplication URL
```

The advantage of starting up **jdb** like this is that you can set breakpoints before your application is executed; **jdb** stops before it executes the main function of the given class.

Note: Use the `-WOAutoOpenInBrowser NO` flag when starting up your server application to prevent the client application from automatically launching in your default browser.

Customizing Your Project With Wizards

Project Builder includes several features, including wizards, that you can—and should—use to add web components, client-side subprojects, interface-controller subclasses, and client-side interface files to Java Client applications. This is especially true with interface (nib) files; never create a client-side nib file using Interface Builder (as, for instance, by choosing the New Database Interface command from the Document menu).

Adding Client-side Subprojects

You can add more than one client-side subproject to your project, especially if you want to use a framework. The subprojects containing EOInterfaceController subclasses and their nib files have to have a special project type: EOJavaClientSubproject.

To add a subproject of this type

1. Open your project in Project Builder.
2. Choose New Subproject from the Project menu.
3. In the New Subproject panel, type a name for your subproject
4. Make sure that the pop-up list displays the project type EOJavaClientSubproject.
5. Click OK.

This procedure adds only the subproject; it does not add an interface-controller subclass, a nib file, or any other files (except makefiles). It also does not add **EOJavaClient.framework** to the root project's list of frameworks.

Adding Interface Controller Subclasses and Nib Files

To add an EOInterfaceController subclass with a new interface file to your client-side subproject

1. Select the Interfaces bucket in your EOJavaClientSubproject subproject
2. Chose New In Project from the File menu.
3. In the New File panel, enter the name for the new EOInterfaceController subclass and its interface file.
4. Click OK.

   The WebObjects Java Client Interface Wizard then appears, asking you to choose templates and other options for the interface.
5. Select the options that you want for the new interface file.
6. Follow the subsequent instructions until completion.

After finishing the wizard, ProjectBuilder will add two files to your client-side subproject: a source (**.java**) file for the EOInterfaceController subclass and the nib file that is owned by the interface controller.

Note: When you create a Java Client project, the EOInterfaceController subclass and its interface file by default have the same name as your application. If you rename these files, you must make adjustments elsewhere in your project, as described in "Manual Adjustments to Java Client Projects."

Adding Web Components (with Interface Controllers)

You can use Project Builder to add a web component containing a WOJavaClientApplet component with a binding to an EOInterfaceController subclass in your project. To create such a web component:

1. Select the Web Components bucket in your root (main) project.
2. Choose New In Project from the File menu.

3. In the New File panel, enter the name of the new web component.

   This also become the name of the EOInterfaceController subclass and its interface file to be used in this web component.

4. In the first screen of the WebObjects Component Wizard, select Java Client for assistance and select a language (usually Java). Click Next. (The Java Client option is disabled if your project does not contain a EOJavaClientSubproject subproject.

5. If you have multiple EOJavaClientSubproject subprojects, the next wizard screen ask you to pick the one you want to associate with the component. Select a subproject and click Next.

6. In the final wizard screen, select the template and associated options to use for the user interface. Follow the wizard's instructions, which vary depending on which options you choose.

When you complete these steps Project Builder adds the following files to your main project:

- A web component (**.wo**) containing a **.html** and a **.wod** file
- An **.api** file for the component in Resources
- A "skeletal" implementation file for the web component in Classes

In addition, Project Builder adds two files to the client-side subproject you chose in the wizard:

- A implementation (**.java**) file for the new EOInterfaceController subclass
- A nib file owned by the EOInterfaceController subclass

Project Builder presents the wizard with the Java Client assistance option *only* if your project has at least one subproject of type EOJavaClientSubproject.

Manual Adjustments to Java Client Projects

You should always use the Java Client wizards if you can because the files that they generate have characteristics that are important for Java Client applications. These files have various dependencies and assumptions, which you must know about if you decide to create them manually.

- If you create a subproject of type EOJavaClientSubproject by hand, make sure that **EOJavaClient.framework** is added to the frameworks of the main

project. Otherwise the compiler might not find all the Java Client classes required by the interface controllers.

- The file's owner class of an Java Client interface (nib) file must be the EOInterfaceController subclass that uses it. It is also very important that the package name of the file's owner class is identical to the package name of the interface controller. So if you change the package of the interface controller, you have to open the interface file in Interface Builder and change the name of the EOInterfaceController subclass used for the file's owner.

- The “interfaceControllerClassName” binding of WOJavaClientApplet used in web components has to be the complete class name of an interface controller in a EOJavaClientSubproject, including the full package prefix. If you change the package of the interface controller, you have to change the value of the “interfaceControllerClassName” binding.

- If you change the size of a window in a nib file which is later placed in a WOJavaClientApplet (because the WOJavaClientApplet uses the corresponding EOInterfaceController subclass), you have to modify the size bindings of the WOJavaClientApplet so that the window contents still fit into it.

- You might want to add additional bindings to a WOJavaClientApplet. This component takes standard java.applet bindings plus some special Java Client ones. See “The Ingredients of a Java Client Project” in the tutorial for more information or refer to the WOJavaClientApplet directory in the WebObjects Java Client examples for a complete list of bindings.

- If you use Sun's Java Plug-in, you must set the value of the “useJavaPlugin” binding of all WOJavaClientApplets to YES.

A typical **.wod** file for a web component using a WOJavaClientApplet looks like this:

```
Applet: WOJavaClientApplet {
   height = 567;
   width = 695;
   interfaceControllerClassName = "movie.client.Movie";
   useJavaPlugin = NO;
}
```

Chapter 4

Enterprise Objects Framework Concepts

Note to Oracle Users

The Oracle login panel is designed to work with SQL*Net v2, which gets the host machine name from the file **tnsnames.ora**. If you're using SQL*Net v1, you must explicitly supply the host machine name along with the server ID in the Server ID field by using a string of the following format:

T:hostMachine:serverID

For example, if you are using a host machine called "tahoe" and your database's server ID is "eof," you can connect the database by typing the following in the Server ID field:

T:tahoe:eof

What is an Enterprise Object?

An enterprise object is like any other object, in that it couples data with the methods for operating on that data. However, an enterprise object class has certain characteristics that distinguish it from other classes:

- It has properties that map to stored data; an enterprise object instance typically corresponds to a single row or record in a database.
- It knows how to interact with other parts of the Framework to give and receive values for its properties.

The ingredients that make up an enterprise object are its class definition and the data values from the database row or record with which the object is instantiated. An enterprise object also has a corresponding model that defines the mapping between the class' object model and the database schema.

What is a Model?

One of the fundamental features of Enterprise Objects Framework is that it maps the data in relational databases to objects. The correspondence between an enterprise object class and stored data is established and maintained by using a *model*. A model defines, in entity-relationship terms, the mapping between enterprise object classes and a database.

The following table describes the database-to-object mapping provided in a model:

Database Element	Model Object	Object Mapping
Data Dictionary	EOModel	—
Table	EOEntity	Enterprise object class
Column	EOAttribute	Enterprise object class instance variable
		(class property)
Row	—	Enterprise object instance

In addition to storing a mapping between the database schema and enterprise objects, a model file stores information needed to connect to the database server. This connection information includes the name of an adaptor to load so that Enterprise Objects Framework can communicate with the database.

What are EODisplayGroups and EOEditingContexts?

EODisplayGroup

EODisplayGroups transport values between an enterprise object and a user interface object. You also need an EODatabaseDataSource, which acts on behalf of the EODisplayGroup to fetch enterprise objects from the database. In combination, EODisplayGroup and EODatabaseDataSource coordinate the flow of data between the user interface and the database. The EODisplayGroup that's created when you drag an entity from EOModeler into Interface Builder is actually a compound object that consists of both an EODisplayGroup and an EODatabaseDataSource.

EOEditingContext

When you drag an entity into the nib file window from your model, an EOEditingContext object is added to your application along with the EODisplayGroup that's created from the entity. An EOEditingContext manages the graph of enterprise objects in your application. The EOEditingContext is responsible for ensuring that all parts of your application stay in sync. When an enterprise object changes, the EOEditingContext broadcasts a notification so that other parts of the application (such as the user

interface) can update themselves accordingly. The EOEditingContext also manages undo, and is the object through which you save changes to the database. For more information, see the EOEditingContext class specification in the *Enterprise Objects Framework Reference*.

What is an Association?

In the previous exercise, when you made a connection from the pop-up list to an EODisplayGroup, you formed an *association*. Associations were also involved when you created a table view by dragging an entity from EOModeler into Interface Builder—the associations were formed for you as a by-product of dragging in the entity.

EODisplayGroups use associations (EOAssociations) to mediate between enterprise objects and the user interface. An association ties a single user interface object, such as a table column, to a key (a named property) in an enterprise object or objects managed by the EODisplayGroup.

Associations keep the user interface synchronized with enterprise object values. When an object changes, its display in the user interface updates to reflect the change. Likewise, when the user edits the user interface, the values in the object are updated accordingly.

Associations can have multiple *aspects*. For example, in the preceding exercise you selected the **titles** aspect for the EOPopupAssociation to display all of the class keys whose values you could choose to display in the pop-up list. EOPopupAssociation also has several other aspects: **selectedTitle**, **selectedTag**, **selectedObject**, and **enabled**.

Enterprise Objects Framework includes associations for different types of user interface objects, such as table columns, text fields, pop-up lists, and so on. Each association has multiple aspects.

For a complete discussion of this subject and a listing of all possible associations, see the EOAssociation class and subclass specifications in the *Enterprise Objects Framework Reference*.

When Do You Use a Custom Enterprise Object Class?

Enterprise Objects Framework provides a "default" enterprise object class, EOGenericRecord. An EOGenericRecord can take on values for any properties defined in your application's model, but it implements no custom behavior. EOGenericRecord objects can hold simple values as well as refer to other enterprise objects through relationships defined in the model.

The criterion for deciding whether to make your enterprise objects custom classes or to simply use the EOGenericRecord class is *behavior*. One of the main reasons to use the Enterprise Objects Framework is to associate behavior with your persistent data. Behavior is implemented as methods that "do something" (as opposed to merely setting or returning the value for a property). Since the Framework itself handles most of the behavior related to persistent storage, you can focus on the behavior specific to your application.

Because the Studio and Talent classes need to have specialized behavior (for example, to perform validation when you attempt to save changes to the database), they need to be custom classes.

Adding Behavior to Enterprise Objects

Some of the more common ways to add behavior to your enterprise object classes are:

- Performing computations based on the values of class properties. For example, from an Employee's salary property, you might calculate a bonus.
- Managing the creation and insertion of objects (for example, assigning default values to newly created objects, creating related objects as the by-product of inserting a new object, appropriately setting relationships for new objects, and so on)
- Performing validation when a particular operation (such as save or delete) takes place
- Adding sophisticated business logic

For a more complete discussion of this subject, see the chapter "Designing Enterprise Objects" in the *Enterprise Objects Framework Developer's Guide*.

Chapter 5

Glossary

Several of the terms listed here apply to relational databases and entity-relationship modeling. Others apply strictly to Java Client applications.

adaptor

A mechanism that connects your application to a particular database server. For each type of server you use, you need a separate adaptor. Enterprise Objects Framework provides adaptors for Informix, Oracle, and Sybase servers, and for any server that is ODBC compliant.

attribute

In Entity-Relationship modeling, an identifiable characteristic of an entity. For example, **lastName** can be an attribute of an **Employee** entity. An attribute typically corresponds to a column in a database table. See *flattened attribute*, *entity*, and *relationship*.

class property

An instance variable in an enterprise object that meets two criteria: it's based on an attribute in your model, and it can be fetched from the database. "Class property" can either refer to an attribute or a relationship. In EOModeler you can specify class properties for server enterprise-object classes and class properties for client classes.

column

In a relational database, the dimension of a table that holds values for a particular attribute. For example, a table that contains employee records might have a column titled "LAST_NAME" that contains the values for each employee's last name. See *attribute*.

compound primary key

In a database table, the group of columns whose values, taken in combination, are guaranteed to uniquely identify each row. See *primary key*.

data dictionary

In relational databases, the system tables that describe the organization of data in a particular database.

database server

A data storage and retrieval system. Database servers typically run on a dedicated computer and are accessed by client applications over a network.

enterprise object

An Objective-C or Java object that conforms to the key-value coding protocol, whose properties (instance data) can map to stored data. An enterprise object brings together stored data with the methods for operating on that data. See *key-value coding* and *property*.

entity

In Entity-Relationship modeling, a distinguishable object about which data is kept. For example, you can have an Employee entity with attributes such as lastName, firstName, address, and so on. An entity typically corresponds to a table in a relational database; an entity's attributes in turn correspond to a table's columns. See *attribute* and *table*.

Entity-Relationship modeling

A discipline for examining and representing the components and interrelationships in a database system. Also known as E-R modeling, this discipline factors a database system into entities, attributes, and relationships.

fetch

In Enterprise Objects Framework applications, to retrieve data from the database server into the client application, usually into enterprise objects.

flattened attribute

A special kind of attribute that you add from one entity to another by traversing a relationship. For example, employees work for departments; you can add an attribute (such as departmentName) from the Department entity to the Employee entity as a flattened attribute. A flattened attribute is normally implemented by joining the tables corresponding to the source and destination entities whenever the attribute's data is fetched. See *relationship* and *attribute*.

foreign key

An attribute in an entity that gives it access to rows in another entity. This attribute must be the primary key of the related entity. For example, an Employee entity can contain the foreign key deptID, which matches the primary key in the entity Department. You can then use deptID as the source attribute in Employee and as the destination attribute in Department to form a relationship between the entities. See *key*, *primary key*, and *relationship*.

generic record

An instance of the EOGenericRecord default enterprise object class. A generic record has properties that map to stored data, but unlike a custom enterprise object, it adds no behavior to that data. Like custom enterprise objects, generic records conform to the key-value coding protocol; see *key-value coding*.

interface controller

An instance of a subclass of EOInterfaceController that is the owner of a nib file containing a "Java archive" describing a user interface made up of "Swing" (Java Foundation Classes) objects.

join

An operation that provides access to data from two tables at the same time, based on the values contained in related columns.

key-value coding

The mechanism that allows the properties in enterprise objects to be accessed by name (that is, as key-value pairs) by other parts of the Framework.

many-to-many relationship

A relationship in which each record in the source entity may correspond to more than one record in the destination entity, and each record in the destination may correspond to more than one record in the source. For example, an employee can work on many projects, and a project can be staffed by many employees. See *relationship*.

model

An EOModel object that defines, in Entity-Relationship terms, the mapping between enterprise object classes and the database schema. This definition is

typically stored in a file created with the EOModeler application. A model also includes the information needed to connect to a particular database server; see *connection dictionary.*

record

The set of values that describes a single instance of an entity; in a relational database, a record is equivalent to a row.

relational database

A database designed according to the relational model, which uses the discipline of Entity-Relationship modeling and the data design standards called normal forms.

relationship

A link between two entities that's based on attributes of the entities. For example, the Department and Employee entities can have a relationship based on the deptID attribute as a foreign key in Employee, and as the primary key in Department (note that though the join attribute deptID is the same for the source and destination entities in this example, it doesn't have to be). This relationship would make it possible to find the employees for a given department. See t*o-one*, *to-many*, *many-to-many*, *primary key,* and *foreign key.*

row

In a relational database, the dimension of a table that groups attributes into records.

table

A two-dimensional set of values corresponding to an entity. The columns of a table represent characteristics of the entity and the rows represent instances of the entity.

to-many relationship

A relationship in which each source record has zero to many corresponding destination records. For example, a department has many employees

to-one relationship

A relationship in which each source record has exactly one corresponding destination record. For example, each employee has one job title.