Lutris®

Enhydra™

Getting Started

The Lutris Enhydra Development Team: Jason Abbott, Louie Boczek, Dennis Chatham, Kyle Clark, Mark Diekhans, Dick Gemoets, Jay Gunter, Aidan Hosler, Wes Isberg, Andy John, Peter Johnson, Matthew Kalastro, Ray Kiuchi, Wendy Lin, Andrew Longsworth, Paul Mahar, John Marco, Shawn McMurdo, Paul Morgan, Christophe Ney, Scott Pirie, Kristen Pol, Wayne Stidolph, Josh Sugnet, Matt Schwartz, Mike Ward, David Young.

Documentation Team: Peter Darrah, Michael Maceri, C. Rand McKinney.

Printed in the U.S.A.
ENH-US040-30 1E0R400
0001020304-9 8 7 6 5 4 3 2 1

1200 Pacific Ave., Suite 300, Santa Cruz, CA 95060 • Phone 831·471·9753 • Fax 831·471·9754O

Contents

CHAPTER

1

Introduction

This book introduces the Enhydra Application Server and the Enhydra development environment. It provides an introductory overview of Enhydra and explains how to develop an application by using an example illustrating some of the key principles of Enhydra applications.

What You Should Already Know

This book assumes you have the following basic skills:

- A general understanding of the Internet, the World Wide Web (WWW), and Hypertext Markup Language (HTML).
- Good working knowledge of the Java programming language. Some knowledge of Java servlets is also helpful.
- Knowledge of basic Unix commands and the Unix **make** utility. This is not necessary if you are developing your application with the Kelp Toolset in an IDE such as JBuilder.
- A good understanding of relational databases; knowledge of SQL is helpful.

Document Conventions

Enhydra runs on a variety of operating systems. In general, file and directory paths used in this book are in Unix format, for example `/usr/local/bin`. To convert these paths into Windows format, simply change the forward slash (/) characters into backslash (\) characters; for example, `C:\usr\local\bin`.

This book uses the following typographical conventions:

- New terms being introduced are shown in *italics*.

- File names, directory names, and URLs are shown in fixed-pitch font, for example: `/usr/local/bin`
- Java packages, classes, methods, and other identifiers are shown in bold, for example, the **Session** object. Method names are suffixed with empty parentheses, even if the method takes parameters, for example **run( )**.
- Commands that you enter directly are shown on a separate line in fixed-pitch code typeface. For example:
  ```
  cd mydir
  ```
- Code examples are shown in fixed-pitch font. For example:
  ```
  System.out.println("something");
  ```
- Utility program names and options are shown in bold. For example: the **make** utility; the **-keep** option.

Where to Find Enhydra Information and Support

You can find a variety of information and support at the Enhydra web site, http://www.enhydra.org.

Software Downloads

You can download the latest version of Enhydra and other related software at: http://www.enhydra.org/software/downloads/index.html.

Online Documentation

A wide range of documentation is available at the Enhydra web site, at: http://www.enhydra.org/software/enhydra/documentation/index.html. The Enhydra installation also includes HTML documentation, as described in “Viewing Online Documentation” on page 22.

E-mail Lists

Lutris encourages you to join one or more of the following Enhydra E-mail lists:

- **Enhydra@enhydra.org** is the Enhydra mailing list for developer interaction.This list is monitored by the Enhydra project team, and is the ideal place to get answers to your questions from fellow Enhydra developers.
- **Enhydra-digest@enhydra.org** is a weekly digest of all mail sent to enhydra@enhydra.org.
- **EnhydraEnterprise@enhydra.org** is the Enhydra Enterprise mailing list, which is tailored for those who are developing and deploying Enhydra applications on a large scale. Here you can find answers to the more detailed Enhydra questions

such as those on Enterprise Java Beans (EJB) and the Common Object Request Broker Architecture (CORBA).

- **EnhydraEnterprise-digest@enhydra.org** is a weekly digest of EnhydraEnterprise@enhydra.org.
- **Enhydra-announce@enhydra.org** is the mailing list for receiving Enhydra announcements.

To join one or more of these lists, go to http://www.enhydra.org/community/mailingLists/index.html.

Mailing List Archives

You can search the combined Enhydra mailing list archives at: http://www.enhydra.org/community/mailingLists/index.html

Bug Reporting

You can report bugs in Enhydra, or request features in future releases at: http://www.enhydra.org/community/bugReports/index.html

Working Groups

Enhydra working groups bring together developers interested in creating new technologies for Enhydra and Enhydra applications. Each working group provides access to the current project source code and to the project E-mail list, which allows you to communicate with the project leaders and other developers. Go to http://www.enhydra.org/community/mailingLists/index.html to join one or more of these working groups.

Working Group	Focuses On...
Rocks	Defining Rocks, a XMLC-based application framework that provides a high-level interface to the Document Object Model DOM and Servlet 2.2 API. By abstracting these APIs and implementing common presentation logic, Rocks provides general functionality common in web applications.
DODS (Data Object Design Studio)	Performing Java data binding with DODS and using DODS to build Enhydra applications.
Enhydra Director	Defining the Enhydra Director architecture that enables Enhydra applications to distribute processing among application instances, either on the same or separate machines. Initially, Enhydra Director will support load balancing managers on popular web servers and operating systems.
Kelp	Enabling development of Enhydra applications using popular IDEs; currently JBuilder Foundation, Standard, Professional, or Enterprise. This work will include new functionality in the Enhydra Application Wizard, The Enhydra XMLC Wizard and XMLC Toolset, as well as sample JBuilder projects.

Working Group	Focuses On...
Internationalization	Resolving issues related to creating localized Enhydra applications.
EJB Containers	Integrating JOnAS Enterprise JavaBeans (EJB) containers with Enhydra. Applications that use EJB technology benefit from portability, scalability, and simplified development, deployment, and maintenance.
Web Containers	Creating a cutting-edge Servlet 2.2 container with full support for Rocks, JSP 2.0 and an embedded Web Server.
JMX	Using Sun's Java Management Extensions (JMX) API to enable both the Enhydra multiserver and Enhydra applications to use the Simple Network Management Protocol (SNMP), a hardware and software application protocol for managing applications in a network environment.
Architecture	Identifying the "big picture" for the components that make up Enhydra Enterprise, a full J2EE-compliant version of Enhydra. Includes identifying common infrastructure elements that other working groups depend on, defining specific requirements for those elements, and providing common implementation elements for the other groups.
Database and Transaction Manager (DBTM)	Creating the database connection management and transaction management components of Enhydra Enterprise. The DBTM is based on the JOnAS database connection pool management and JOnAS Transaction Manager.

Acknowledgements

As an open source product, Enhydra benefits from the contributions of many developers around the world. In particular, Lutris would like to thank the following people who have contributed information used in some form in this book: Robert Cadena, G. W. Estep, Rohan Oberoi, Dan Rosner, Peter Speck, and David Trisna.

CHAPTER

2

Overview

This chapter provides a high-level overview of Enhydra, Enhydra applications, and the tools used to create Enhydra applications.

What is Enhydra?

Enhydra is an application server for running robust and scalable multi-tier web applications, and a set of application development tools.

An *application server* usually operates between a web server and a database server, and provides dynamically-generated content for the web server to send to web browser clients.

As illustrated in Figure 2.1, Enhydra has three parts:

- **Multiserver**, that runs Enhydra applications either by itself or with a web server.
- **The application framework**, a collection of Java classes that provide the runtime infrastructure for Enhydra applications.
- **Enhydra tools** that you use to develop Enhydra applications.

An *`Enhydra application`* is a Java program that runs in Multiserver and uses the Enhydra application framework at runtime. Figure 2.1, “Enhydra application architecture” illustrates the basic elements of an Enhydra application within the context of the Enhydra architecture.

The next section describes Enhydra applications in more detail.

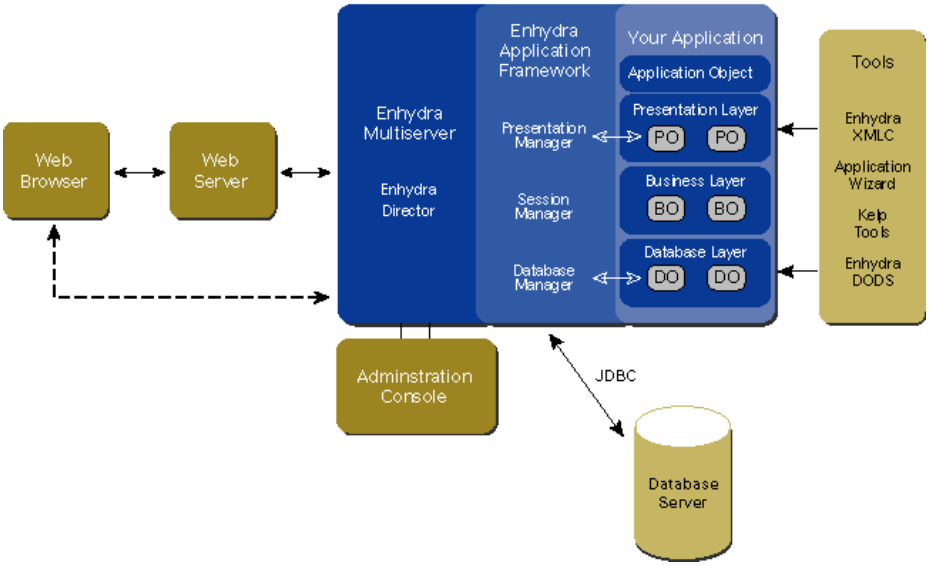


Figure 2.1 Enhydra application architecture

What's New in 3.0

Enhydra 3.0 includes a number of new features, including:

- Load balancing through Enhydra Director (see "Enhydra Director" on page 9 for more information).
- Support for the Java Servlet 2.2 API.
- Support for Java Server Pages (JSP) 1.1 API.
- Dynamic recompilation of XMLC classes.
- Numerous other improvements, including:
  - session maintenance using URL rewriting instead of cookies
  - Wireless Markup Language (WML) support
  - enhanced JBuilder integration
  - support for Oracle's JDeveloper IDE
  - an enhanced Multiserver Administration Console.

Anatomy of an Enhydra Application

An Enhydra application has, at a minimum:

- A single application object.
- One presentation object for each page to be dynamically generated.

These objects run in the context of the Enhydra application framework, as described in “The Enhydra Application Framework” on page 10.

Application Object

The application object is the central hub of an Enhydra application, and is a subclass of **com.lutris.appserver.server.StandardApplication**. It contains application-wide data, such as:

- The application's name.
- The application's status (running / stopped / dead).
- The name and location of the configuration file that initializes the application.
- The log channel to use for logging.
- References to the application's: session manager, database manager, and presentation manager, as described in “The Enhydra Application Framework” on page 10.

Properties

You can add properties (instance variables) to the application object to store information that needs to be accessible throughout the application. For example, if your application has a dozen pages that need to share a collection of customer data, you could make a vector containing the data a property of the application object so all the pages can easily access it.

Methods

Each application object has the following methods:

- The **startup( )** method starts the application; you can extend this to perform other startup functions, such as reading settings from the configuration file.
- The **requestPreprocessor( )** method initializes the **Session** data structure; you can extend this as needed; for example, to check for HTTP basic authorization.

In general, application objects do not deal with HTML, handle requests or otherwise talk to the network. The presentation objects perform these tasks.

Presentation Objects

A presentation object (PO) generates dynamic content for one or more pages in an Enhydra application.

When a browser requests a URL that ends in `.po`, Enhydra passes the request on to the corresponding presentation object. Enhydra then instantiates and calls the presentation object. For example, for the URL `http://www.foo.com/myapp/Xyz.po`, Enhydra will call the presentation object **Xyz**.

Note Enhydra only calls a presentation object for URLs with a `.po` suffix. The web server will generally serve a static file for other requests.

Presentation objects must implement the interface **com.lutris.appserver.server.httpPresentation.HttpPresentation**. This interface has one method, **run( )**, that Enhydra calls, passing it an HTTP request to handle. Presentation objects differ from Servlets in that they need handle only a single request at a time. No concurrency control is required.

Enhydra also provides a response object that a PO can use to write data in response (similar to a servlet's **service( )** method). POs usually handle GET requests (for example form submissions) and respond by writing HTML, but they may perform other functions, for example, read files sent by a POST request.

The Three Layers

For modularity and ease of maintenance, you should divide your application into three distinct parts or layers:

- **The presentation layer** contains presentation object (POs). POs handle how the application is presented to web browsers through HTML.
- **The business layer** contains business objects (BOs). BOs contains the application's business logic, including algorithms and specialized functions, but not data access nor display functions.
- **The data layer** contains data objects (DOs). DOs interface with the persistent data source, which is typically a relational database.

The Enhydra Application Framework only requires that you use an application object and presentation objects. The business and data classes you create are up to you. Dividing your application into these three layers minimizes maintenance cost because it isolates the application's data layer from the user interface. This, in turn, enables you to change the data layer without affecting the presentation layer.

An additional benefit of having a distinct data layer is that you can use the Data Object Design Studio (DODS) to create your data objects. DODS graphically creates data objects to populate the data layer, and creates both Java code and SQL code to create the corresponding tables in the database.

Multiserver

Multiserver is the runtime component of Enhydra. It provides the services that an Enhydra application uses to communicate with the web server, and perform all its other runtime functions.

To understand Enhydra Multiserver, you need to understand a little about *servlets*. A servlet is a Java class that dynamically extends the functionality of a web server. Normally, when a browser sends a request to a web server, the server simply finds the files identified by the requested URL and returns them to the browser. However, if the browser requests a page constructed by a servlet, the server sends the request information to a servlet, which constructs the response dynamically and returns it to the server.

The Java servlet API is a standard extension to Java and is part of the Java2 Enterprise Edition (J2EE). Some web servers support the servlet API directly, while others require an adjunct servlet runner, such as JServ for the Apache web server. Enhydra 3.0 supports the servlet API version 2.2.

Each Enhydra application runs as a single servlet, in contrast to a generic servlet application, which typically has one servlet for each dynamically-generated page. Enhydra Multiserver is a *servlet runner* that executes servlets such as Enhydra applications, either with a web server or by itself. Multiserver can run applications in a small-scale development environment on its own; for a production environment requiring greater performance, you can use Multiserver in conjunction with a web server.

Note Since an Enhydra application is a servlet, it can run in any standards-compliant servlet runner, not just in Multiserver.

Enhydra Multiserver has a custom class loader for each application (servlet). Because of this one-to-one correspondence between servlets and class loaders, you can install and start new applications without needing to stop the server. To update an existing application, you simply restart its class loader.

Enhydra Director

Enhydra Director is a new feature in Enhydra 3.0 that provides applications superior scalability by distributing the user load among several Enhydra Multiservers. The load balancing algorithm supports session affinity and weighted round-robin distribution with server failover. Session affinity means that a particular session instance will always access the same Multiserver instance. The weighted round-robin load balancing scheme takes into account the capacity of each Multiserver instance and the number of existing connections. Server failover ensures that if a Multiserver goes down, all application connections are automatically transferred to another Multiserver instance.

Director uses a new connection method, the *Enhydra Director connection method*, and a new set of web server extension modules. Enhydra Director works with Apache servers via the Apache Module interface, Netscape servers via Netscape Application Programming Interface (NSAPI), and Microsoft servers via the Internet

Server Application Programming Interface (ISAPI), and other web servers via the Common Gateway Interface (CGI).

For more information and installation instructions for Enhydra Director, see the online documentation installed with Enhydra.

The Administration Console

Enhydra Multiserver provides an administration console for managing applications through a web browser. The administration console enables you to:

- Start and stop applications.
- Add and remove applications from management.
- Modify an application's operational attributes and check its status.
- Trace the execution of an application to aid in debugging.

The administration console is described in more detail in “Using the Multiserver Administration Console” on page 85

The Enhydra Application Framework

The Enhydra application framework includes:

- The presentation manager
- The session manager
- The database manager

In general, the application framework includes all the classes in the **com.lutris.appserver.server.*** packages, that provide the infrastructure that Enhydra applications use at runtime. The general architecture of an Enhydra application in the context of the application framework is illustrated in Figure 2.1, “Enhydra application architecture”.

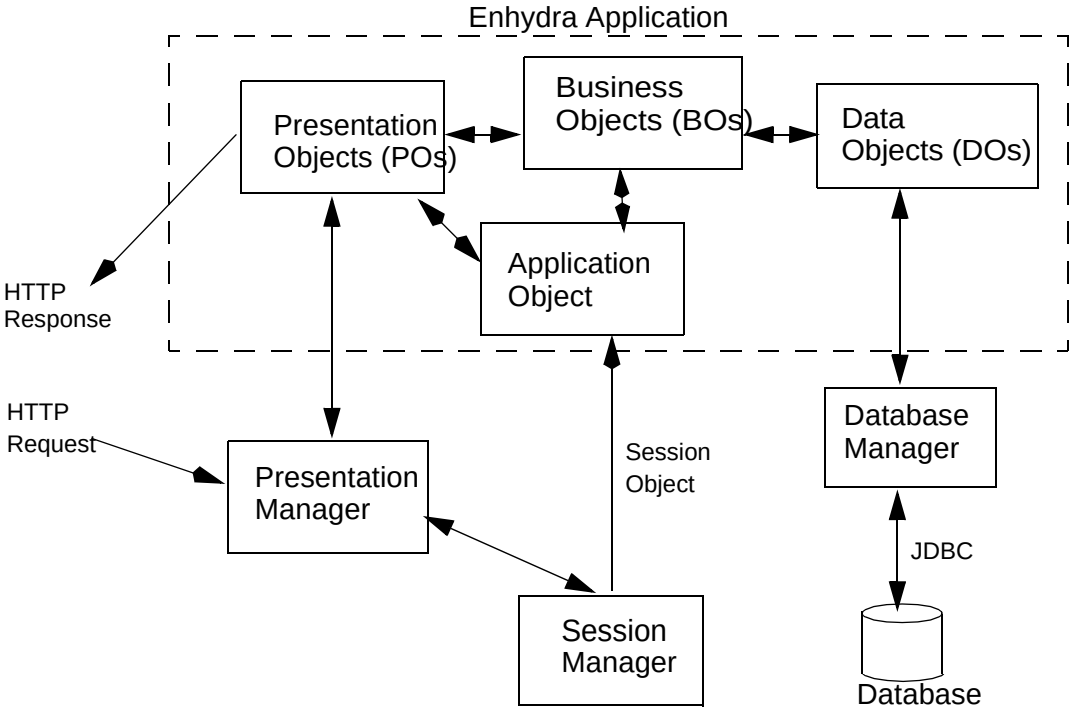


Figure 2.2 Enhydra Application and Enhydra Framework

Presentation Manager

The Enhydra presentation manager handles the loading and execution of the presentation objects in an Enhydra application. The presentation manager maps URLs to POs, and calls the PO's **run( )** method.

Each Enhydra application has one instance of a presentation manager. To increase performance, the presentation manager caches POs and associated files in memory as necessary. The presentation manager also provides the key that the session manager uses to locate a session; this key is either a cookie or a string appended to each URL in the application.

Each application has a presentation manager that is an instance of the class **com.lutris.appserver.server.httpPresentation.HttpPresentationManager**. The **com.lutris.appserver.server.httpPresentation** package contains classes and interfaces that the presentation manager and POs use.

Session Manager

The Enhydra session manager enables an application to maintain state throughout a session. A session is defined as a series of requests from the same user (browser client) during a specified time period. Enhydra provides a general implementation of session management that you can extend to create more sophisticated state models.

Enhydra maintains user state by creating a Session object for each user. When a user first makes a request to an application, the session manager creates a new Session object and assigns it a unique session ID. The session manager uses the session ID to retrieve the Session object for subsequent requests. Applications can add user-specific information to the Session object, and then access the Session object from the request object, as it is passed through the application.

If a user has been idle (has not issued a request to the application) for more than the period specified in the configuration file, the user's session becomes invalid, and the session manager releases the corresponding Session object. This makes it possible to implement security schemes that require users to log in before accessing the application. In such a scheme, the user enters an appropriate password and gains access to the rest of the application; however, once the user's session has been idle for more than the allowed time (for example, ten minutes), the application requires the user to log in again.

Each application has a session manager that is an instance of the class **com.lutris.appserver.server.session.StandardSessionManager**. When it is created, the session manager reads the maximum time that a session can persist, the maximum session idle time, and other related information from the application configuration file, *appName*`.conf`. The **com.lutris.appserver.server.session** package contains classes and interfaces that the session manager and the application use for session management.

Database Manager

The Enhydra database manager controls an application's pool of database connections. The database manager works with *logical databases*. A logical database is an abstraction that hides the differences between different database types. A logical database uses Java Database Connectivity (JDBC) to communicate with database servers such as Oracle, Sybase, and Informix.

The database manager is responsible for the state of a database connection, the SQL statements that are being executed, and the result sets that are in progress. Specifically, the database manager:

- Allocates and releases connections to the logical database.
- Allocates object IDs from the logical database.
- Creates queries and transactions.
- Maintains other database-related information.

Each application has a database manager that is an instance of the class **com.lutris.appserver.server.sql.StandardDatabaseManager**. When it is created, the database manager reads a configuration file that specifies the logical database to use, the actual database types to which it maps, and other related information.

The **com.lutris.appserver.server.sql** package contains the classes and interfaces that the database manager and data objects use.

Enhydra Tools

Enhydra includes the following tools to help you create applications:

- The Application Wizard
- The Extensible Markup Language Compiler (XMLC)
- The Data Object Design Studio (DODS).

Application Wizard

The application wizard (**newapp**) is a command-line tool that creates a basic framework for an Enhydra application. The application wizard allows you to create and run a new "stub" application in a matter of minutes, giving your development project a jump-start.

The application wizard is also incorporated into the Enhydra Kelp tools, so that you can perform the same operation from the JBuilder graphical IDE.

For an example of using the application wizard, see "Creating Your First Application" on page 25.

XMLC

The Extensible Markup Language Compiler (XMLC) creates a Java object that mirrors the structure of an XML document. Extensible Markup Language (XML), which is defined by the World Wide Web Consortium (W3C), is the universal format for structured documents and data on the Web. XMLC uses the document object model (DOM), a W3C standard interface to allow a programs to access and update the content and structure of XML documents.

Note Although XMLC works with XML documents, this book will focus on its use with HTML pages.

XMLC enables you to cleanly separate HTML templates in your application, which are typically created by page designers from Java code, which is typically created by programmers. This provides increased modularity and eases team development and application maintenance. Page designers can change the user interface of the application without requiring any code changes, and the programmers can change the "back-end" Java code without requiring any changes to the HTML.

XMLC is a command-line tool that generates a Java class file from an HTML input file. An application can use the Java class at runtime to change the content or attributes of any tags with ID or CLASS attributes. For an example of using XMLC, see “Tutorial: Building Enhydra Applications” on page 25.

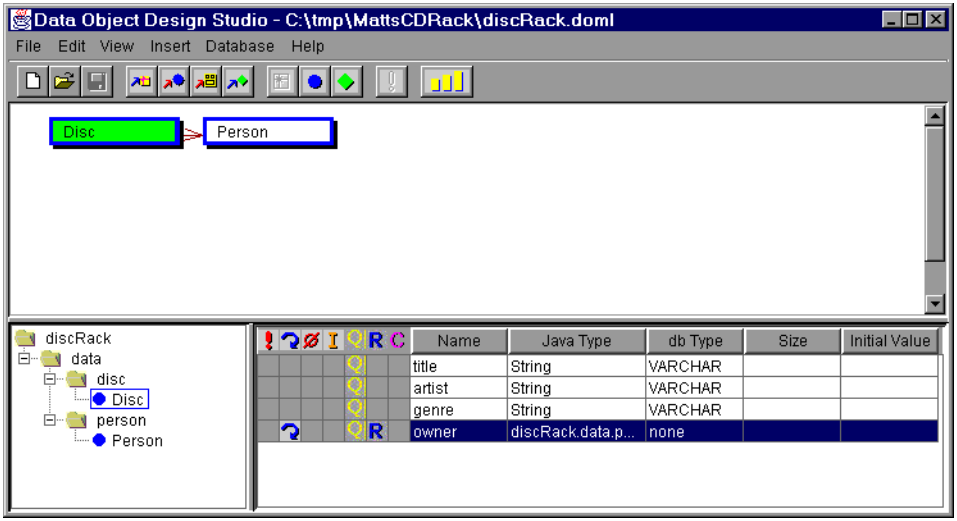
Dynamic Recompilation

XMLC dynamic recompilation is a new feature in Enhydra 3.0. It enables you to change HTML layouts at runtime without restarting an application.

With the new feature, you can make any changes to the static content of HTML pages, and the application will automatically pick up the changes. As long as you do not add or change any ID and CLASS attributes of tags in a page, you don’t have to rebuild and restart the application.

The Data Object Design Studio (DODS)

The Data Object Design Studio (DODS) is a graphical tool you can use to define your data model.



Data Objects (DOs) in the DODS data model correspond to tables in the database. Each DO has *attributes* that describe database columns and *reference attributes* that refer to other DOs. Reference attributes enable you to create a hierarchy of DOs, for example many-to-one or many-to-many relationships.

Once you have defined your data model, DODS generates all of the code to implement it:

- The SQL code to define the database tables
- The Java code to create the corresponding application data objects

For each data object, DODS generates a set of source files. For example, if your data model includes the definition of an entity named “thing,” then DODS would generate the following:

- A file containing the SQL CREATE TABLE command to construct a table in a relational database, named `thing.sql`.
- A Java source file defining a data object representing a row the table. This class provides a “set” and “get” method for each attribute, methods to handle caching, and is a subclass of the Enhydra framework class **GenericDO**. In this example, the class would be named **ThingDO**.
- A Java source file that defines a query class, which provides SQL query access to the database table. The query class will return a collection of **ThingDO** objects that represent the rows found in the table matching criteria passed from the application.
- Optionally, a Java source file that defines a business data object (BDO) class to handle interaction between the data object and the business layer. In this example, the class would be named **ThingBDO** by default.

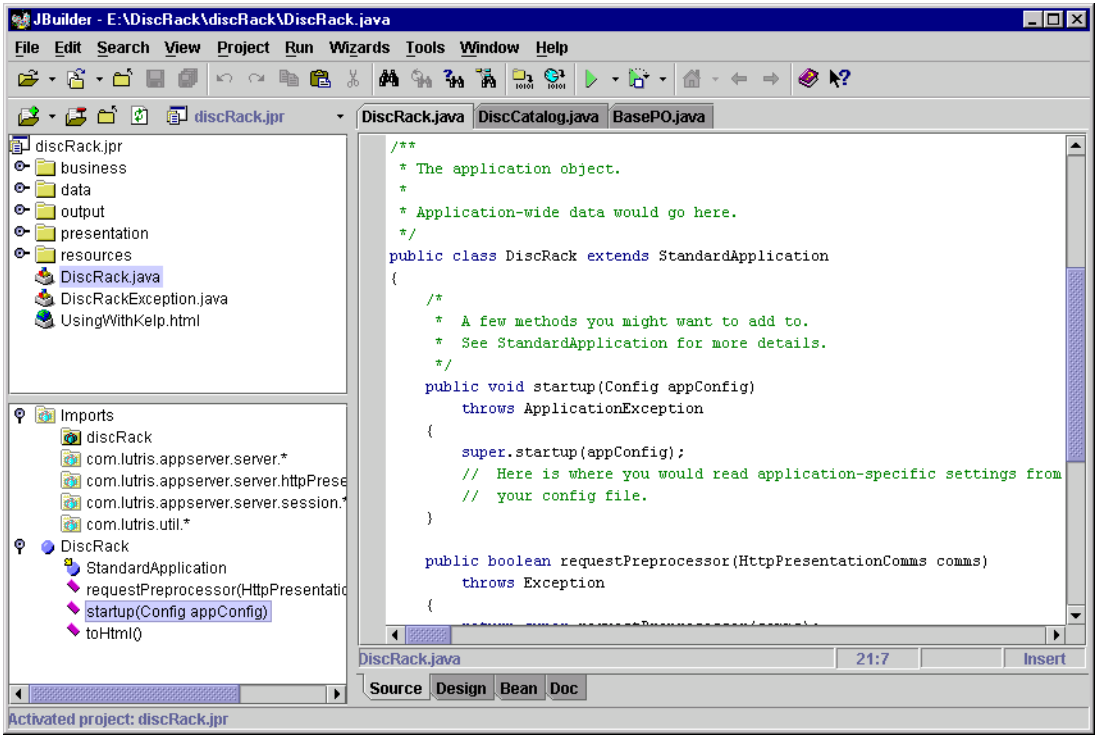
When DODS generates source code, it creates a directory structure that matches the package hierarchy you have designed. DODS creates the **make** files in each directory, and runs **make** on the generated source code. All that is left for you to do is create a database using the SQL files, and write the more interesting components of your application.

For an example of using DODS, see “Using DODS” on page 48.

JBuilder and the Kelp Tools

The Kelp tools enable you to develop Enhydra applications in Borland JBuilder Oracle JDeveloper interactive development environments (IDEs). You can use Kelp in place of Enhydra's shell scripts and **make** files.

The following is a general outline of Kelp, as features will vary depending on the IDE. See the documentation included with the Kelp tools for specific features supported by your Java IDE.



Enhydra Application Wizard

The Application Wizard generates an Enhydra application that you can develop, run and debug from within an IDE. The wizard lets you set the name, directory and package for your new application. It generates the files described in “Creating Your First Application” on page 25. It also generates a `Readme.html` file that lists the steps to build and run the new application.

XMLC Compiler Wizard

The compiler wizard lets you set XMLC options, select HTML files to compile and call the XMLC compiler from JBuilder.

The wizard also provides a mapping table that maps directories to package names. This is useful when you keep your HTML files in a directory that does not match the package name you want to use in the generated DOM classes. For example, the DiscRack sample project has HTML files in a `resources` directory that need to be compiled using the presentation package.

XMLC Property Pages

Property pages give you full control over how XMLC builds DOM classes from HTML files. The property pages allow you to customize class name generation and set XMLC option files for the entire project as well as for individual HTML files.

For example, the DiscRack sample includes three XMLC options files: one for the presentation package, and two more for packages that reside within the presentation package. You can use the XMLC Property Pages to associate each HTML file in the resource directories with the appropriate XMLC options file in the presentation directories.

Enhydra Sample Project

The Enhydra sample project is an IDE project file that lets you build, debug and run your application from within the IDE. The project also demonstrates how to perform several dynamic page generation tasks using XMLC. When you run the Enhydra sample project, the pages display the HTML tags and the Java methods required to perform each task.

CHAPTER

3

Installation

This chapter explains how to install Enhydra. You can install Enhydra on any system with a standard Java virtual machine. In particular, Enhydra supports:

- Windows NT 4.0, Windows 95, and Windows 98
- Solaris, Linux, and other Unix systems

For more detailed system requirements, see the top-level `index.html` file on the Enhydra CD.

Procedure

Follow these steps to install and configure Enhydra:

1. If you do not already have the Java Development Kit (JDK) 1.2.2, install Java2, as described in “Downloading and Installing Java 2”.
2. Insert the Enhydra CD, and load the `index.html` file in the top level of the CD into your web browser. This file contains general information, including system requirements and registration information. Under the heading “Install Enhydra,” click on the Enhydra link. This displays a page with installation instructions.
3. If you are using Windows, follow the instructions in “Installing on Windows” on page 20. If you are using Linux or a Unix system, follow the instruction in “Installing on Unix” on page 21.
4. Configure Enhydra, following the instructions in “Configuring Enhydra” on page 22.
5. Use your web browser to view the online documentation, as described in “Viewing Online Documentation” on page 22.
6. If you want to run the sample applications, follow the instructions in “Running the Sample Applications” on page 23

Downloading and Installing Java 2

Enhydra 3.0 requires the Java Development Kit (JDK) version 1.2.2, which is part of Java™ 2, from Sun Microsystems. If you have already installed the JDK 1.2.2 on your system, you can skip this section.

Enhydra is certified for use with JDKs other than the Sun JDK. For more information, see the release notes.

You can download Java 2 for Windows from http://java.sun.com/products/jdk/1.2. Select your platform, and follow the download and installation instructions. Reboot your system when the installation is complete.

Note JBuilder contains its own version of the JDK, so if you are going to use JBuilder, you can use its JDK instead of installing Java2 separately. See "Installing JBuilder" on page 24.

Installing Enhydra 3.0

When you install Enhydra, you will select an installation directory. The installation process will create a `lutris-enhydra3.0` sub-directory that contains the Enhydra executables and libraries. This book will refer to this `lutris-enhydra3.0` directory as the *`Enhydra root directory`*, or <*`enhydra_root`*>.

Installing on Windows

On Windows systems, Enhydra 3.0 includes an InstallShield™ program that automates the installation process. It is strongly recommended that you use the InstallShield program to ensure that Enhydra is installed and configured properly. Advanced users may choose to install Enhydra manually on Windows, but Lutris will not support such an installation.

Cygnus Tools

Enhydra requires a Unix shell environment and command emulator on Windows systems. The InstallShield program will install Cygnus Tools from Cygnus Solutions, containing Cygwin®, a Unix shell environment for Windows.

Although it is possible to use a Unix shell environment other than CygWin, Lutris strongly recommends using CygWin.

Procedure

Follow these steps:

1. If you are using Windows NT, log on as a user with Administrator privileges.
2. If you are upgrading from a previous version of Enhydra, delete the old files.

3 Exit all Windows programs.

4 Double-click on the file `LutrisEnhydra3.0.exe` in the `Enhydra/install/cuts` directory on the CD. This will launch the InstallShield program directly from the CD.

   On some systems, you may encounter problems executing the InstallShield program directly from the CD. If so, copy the file `Enhydra/install/cuts/LutrisEnhydra3.0.exe` from the CD to a temporary location on your hard drive. Then double-click on this copy of `LutrisEnhydra3.0.exe` to launch InstallShield.

5 Follow the InstallShield Wizard's instructions to install Enhydra on your system.

Important InstallShield will prompt you to enter an installation directory. The Cygnus Tools require that you install Enhydra on your C: drive

Installing on Unix

Following the instructions in the web pages on the CD, click on the link "Get Enhydra Installation Files." Then right-click on the link of your choice:

- Select `lutris-enhydra3.0.zip` for a zip archive
- Select `lutris-enhydra3.0.tar.gz` for a GNU tar archive
- Select `Lutris-Enhydra-3.0-1.i386.rpm` for Red Hat package manager

Choose "Save Link as...," then select a directory to save the archive on your hard drive.

If you are upgrading from a previous version of Enhydra, now delete the old version.

Red Hat Linux

To use the Red Hat Package Manager (`RPM`) file for Enhydra, log on as `root`, and enter the following command:

```
rpm -ivh Lutris-Enhydra-3.0-1.i386.rpm
```

This installs Enhydra in the `/usr/local/enhydra3.0` directory. To install Enhydra in a different location, enter the following command, replacing `dir` with the name of the desired directory:

```
rpm -ivh --prefix dir Lutris-Enhydra-3.0-1.i386.rpm
```

After the installation completes, check the `rpm_config.status` file to see if RPM was able to find the JDK and configure Enhydra properly.

Solaris and Other Unix Systems

The Enhydra distribution is packaged as a zip archive. Use **unzip** to extract the files from the archive, as follow:

```
unzip lutris-enhydra3.0.zip
```

Note Using anything other than **unzip** (for example Solaris **tar**) to extract the files may not set the file permissions properly. Using **unzip** is strongly recommended. If you do use **tar**, you must use GNU **tar**, available at `ftp.gnu.org`.

Configuring Enhydra

After installing Enhydra, you must configure it as follows:

1. On Windows, open a Cygwin shell window; enter the commands below in this window. On Unix, you may use any shell window.
2. Change to the Enhydra root directory, for example:

   ```
   cd /usr/local/lutris-enhydra3.0
   ```

3. Enter the following command, where *JDK_LOC* represents the directory where the JDK 1.2 is installed:

   ```
   ./configure JDK_LOC
   ```

   For example, if the JDK is installed in `/jdk1.2.2`, use the command:

   ```
   ./configure /jdk1.2.2
   ```

Windows In a Cygwin shell window, you must convert Windows paths to Unix-style paths. If the JDK is installed on your C drive, simply omit the drive in the path (for example, for `C:\jdk1.2.2`, use `/jdk1.2.2`). If installed on any other drive, you must include the drive letter preceded by two slashes, for example, if the JDK is installed in `E:\JDK1.2.2`, then use `//E/JDK1.2.2`.

Setting the PATH Environment Variable

To enable running the Enhydra tools from the command line, you must add `<`*enhydra_root*`>/bin`. to your PATH environment variable.

On Unix systems, simply set the PATH environment variable as you normally would, for example:

```
setenv PATH
```

On Windows systems, to change the path used by applications run from the Cygwin shell, edit the file `.bashrc` in the `enhydra` directory.

This will enable you to run XMLC, DODS, and the Multiserver Administration Console from the command line.

Viewing Online Documentation

In addition to the documentation on the Enhydra.org web site, the Enhydra installation includes HTML documentation. View the file `<`*enhydra_root*`>/doc/index.html` in your web browser. This file includes links to the release notes, the Frequently Asked Questions list, and other informative documents.

The `doc` directory includes the following documentation:

- Release notes, frequently asked questions (FAQ) list, and license information. For updated release notes, refer to the Enhydra.org web site.
- General information on the Enhydra Multiserver Administration Console, in the `admin` directory

- Enhydra developer documentation, in the `developer` directory
- Javadoc API reference documentation, in the `user-doc` directory
- XMLC documentation, including Javadoc for the Document Object Model, in the `xmlc` directory

Running the Sample Applications

Enhydra 3.0 includes some sample applications ("demos") that illustrate using Enhydra. The Windows InstallShield program installs the sample applications by default. In addition, the DiscRack application described in Chapter 5, "The DiscRack Example Application", is installed to `<enhydra_root>/examples/DiscRack`.

On Unix systems, you must install them manually as follows:

1. On the CD, go to the directory `Enhydra/install/cuts/demos`.
2. Decompress the `lutris-enhydra-demos3.0.tar.gz` or `lutris-enhydra-demos3.0.zip` file to a directory of you choice, for example, `/usr/local/lutris-enhydra-demos3.0`.

Follow these steps to run them:

1. Change to the directory containing the sample applications (`lutris-enhydra-demos3.0`) and run the start script:
   ```
   cd /usr/local/lutris-enhydra-demos3.0
   ./start
   ```
2. To access the sample applications, enter the following URL into your browser: `http://localhost:xxx`, where `xxxx` represents the port you specified for the samples during installation. For example, if you specified port 9050, enter `http://localhost:9050`.
3. You will see a page with links to several sample applications:

Sample	Description
DemoApp[1]	A very simple demonstration application.
Calculator[1]	Simulates a pocket calculator.
Chat Room[1]	Uses a special algorithm to achieve the effects of push technology using HTTP. Messages that users enter appear immediately to other users.
Golf Shop	An example of a classic Web shopping cart application used to support an on-line Golf Store. The user name is `enhydra` and the password is `lutris` or you may create your own account.

1. This application was constructed with Enhydra JDDI, a tool for rapid prototyping of simple applications.

Installing JBuilder

Borland JBuilder is a graphical IDE for Java development, as described in “JBuilder and the Kelp Tools” on page 15. Lutris Enhydra Professional Edition includes Borland JBuilder Foundation. To install JBuilder, follow the installation instructions in the `index.html` file in the `Jbuilder` directory on the CD.

Although Enhydra Standard Edition does not include Borland JBuilder, you can download JBuilder Foundation for free from the Borland web site: http://www.borland.com/jbuilder.

JBuilder Foundation includes a complete JDK that you can use with Enhydra. If you install JBuilder first, you can then configure Enhydra to use the JBuilder JDK.

Installing the Kelp Tools

The Kelp Tools for JBuilder and Oracle JDeveloper are included in both Enhydra Standard Edition and Enhydra Professional Edition. You should install Enhydra 3.0 and either JBuilder or JDeveloper before installing Kelp.

For detailed system requirements and installation instructions, see `Enhydra/kelp/kelpdoc.html`.

Windows

To install the Kelp tools on Windows:

- For JBuilder, double-click on `Enhydra/kelp/cuts/jb-kelp1.1.exe` to run the InstallShield program.
- For JDeveloper, double-click on `Enhydra/kelp/cuts/jdev-kelp1.1.exe` to run the InstallShield program.

The InstallShield program will install Kelp and make the Kelp tools accessible from the IDE menus.

Linux and Solaris

To install the Kelp tools on Linux or Solaris:

- For JBuilder, extract the zip file `Enhydra/kelp/cuts/jb-kelp1.1.zip` to your system.
- For JDeveloper, extract the zip file `Enhydra/kelp/cuts/jdev-kelp1.1.zip` to your system.

On Linux and Unix systems, you need to manually configure your tools for use with Kelp. The zip files include a `readme.html` file that includes instructions for setting up the Kelp tools and the sample project.

CHAPTER

4

Tutorial: Building Enhydra Applications

This chapter describes how to build a Enhydra application from the ground up, and provides important tips on Enhydra application development. It begins with using the application wizard to create a starting framework, and leads you through using XMLC to expand the application, then adding simple database access.

If you are already familiar with the basics of Enhydra, you may want to skip to the next chapter, “The DiscRack Example Application” on page 59, for a look at an application with more advanced features.

Note In this tutorial, you are often instructed to enter commands. On Unix platforms, you can enter the commands in any shell. On Windows, you must enter the commands in a Cygwin shell window.

Creating Your First Application

The *application wizard* (sometimes referred to as “newapp”) is a quick way to get up and running with Enhydra. It is a command-line tool that generates the basic Java files and directory structure for a new application.

First, create a directory to contain your new application, and name it anything you want, for example “myapps.” Then, open a shell window and make the new directory the current directory, for example:

```
cd myapps
```

Now, run the application wizard by typing `newapp` followed by the name of the application to create. In this case, call it simpleApp:

```
newapp simpleApp
```

The application wizard will create a new directory called `simpleApp`. This directory is sometimes referred to as the *application root* directory.

Make this the active directory:

```
cd simpleApp
```

Notice that the application wizard created the following in this directory:

- Two **make** files: `config.mk` and `Makefile`
- A `README` file that contains some simple instructions to build and run the application
- A `simpleApp` directory that contains all the source code for the application; the contents of this directory are explained later

Building the Application

Enter the **make** command to build the application:

```
make
```

This will create two sub-directories in the application root directory:

- The `classes` directory contains the application's class files.
- The `output` directory contains everything needed to run the application.

The top-level **make** file, `Makefile`, contains directives that tell **make** to recursively descend the application directory tree, following the directives of the **make** file in each sub-directory. It also has an `INCLUDE` directive that references `config.mk`, which in turn references `<`*`enhydra_root`*`>/lib/stdrules.mk`, a large **make** file shared by all Enhydra applications.

When you build the application, **make** compiles the files in the `simpleApp/simpleApp` source directory and creates a corresponding directory structure in the `classes` directory. It then combines those classes into a `jar` (Java archive) file and places the `jar` file into the `output` directory along with the configuration files needed to run the application.

To start the application, enter the following commands:

```
cd output
./start
```

Now, to access the application, enter the following URL in your browser's location field: `http://localhost:9000`.

You will see the following page in your browser:

You have just built and run your first Enhydra application!

Now hit **ctrl+C** in the shell window to stop the Enhydra process.

How It Works

The application created by the application wizard provides a simple example of how Enhydra works. Look at the file `myapps/simpleApp/simpleApp/presentation/Welcome.html`, which contains a few dozen HTML tags. Notice tags such as these:

```
<CENTER>
The current time is <span id="Time">1/1/00 00:00:00 Oh no!</span>.
</CENTER>
```

At runtime, Enhydra replaces the content of the SPAN tag with a date value. The text in there is just a placeholder; it will never appear at runtime. Note the period outside of the `SPAN` tag will not be replaced. Thus, the sentence will always end with a period.

Look also at the `Welcome.java` file in the same directory. In particular, notice these lines of code:

```
String now = new Date().toString();
WelcomeHTML welcome =
WelcomeHTML)comms.xmlcFactory.create(welcomeHTML.class);
welcome.setTextTime(now);
comms.response.writeHTML(welcome);
```

When you build the application, XMLC finds the `SPAN` tag in the HTML and recognizes the `ID` attribute with value “Time”. It creates a Java class called **WelcomeHTML** with a method **setTextTime( )** that the application uses to modify the text content of the `SPAN` tag. In general, XMLC will create a **setTextXXX( )** method for each `SPAN` tag with ID attribute value “XXX”.

Then, at runtime, the application replaces the original text content of the SPAN tag with a string representation of the current date. Then, the call to **writeHTML( )** writes the document out to the HTTP response, looking something like this:

```
...
<CENTER>
The current time is <SPAN>Mon Feb 28 10:42:34 PST 2000</SPAN>.
</CENTER>
...
```

For a more detailed explanation of XMLC, see "Using XMLC" on page 29.

The Directories and Files in SimpleApp

Let's take a closer look at the directories and files in the simpleApp directory:

- The simpleApp sub-directory contains the source code for the application. It is divided into three sub-directories for the business, data, and presentation layers. The business and data directories are empty in the new application. The presentation directory contains java, html, and media files.
- The classes sub-directory contains the application's compiled Java classes in their package hierarchy, and any associated media files such as .GIF files.
- The output sub-directory contains the application's configuration files, multiserver.conf and simpleApp.conf, and a lib directory with a simpleApp.jar file containing an archived package of everything in the classes directory.

The finished Enhydra application includes the jar file and the configuration file. The **make** program also copies the multiserver configuration file and the start script to the output directory to make it easier to run the application.

Configuration Files

The application configuration files contain critical information that determine how an Enhydra application runs:

- The Multiserver configuration file, multiserver.conf
- The application configuration file, which is named AppName.conf (for example simpleApp.conf) by default
- If the application is to be administered by the Multiserver Administration Console, the multiserverAdmin.conf file
- The start script, which is named start. This is not a configuration file; however, it specifies the all-important CLASSPATH that the application will use. If you do not specify a CLASSPATH in this file, the application will use the system CLASSPATH, which may not be correct.

By editing the configuration files, you can change the various settings contained in the files, as explained below.

Note The application wizard creates "input" versions of the two configuration files in the *app_root/appName* directory; for example, simpleApp/simpleApp. These are the files

that you should edit. Running **make** creates "runtime" versions of the files in the *app_root*/output directory, for example simpleApp/output. You should not edit these versions of the files, because they will be over-written each time you re-build the project.

The Multiserver configuration file contains information that the Multiserver uses to run the application, including:

- The name and location of the application configuration file
- The name of the log file and other log file information
- The TCP port on which the application will run

The application configuration file contains the following important application-specific information:

- The CLASSPATH that this application will use; the difference between this CLASSPATH and the one specified in the multiserver start script is that this one applies to the application's own class loader, while the latter is global in scope for all applications run by that Multiserver instance.
- The class name of the application object; for example:
  ```
  Server.AppClass = simpleApp.SimpleApp
  ```
- The prefix used to derive presentation object class names and paths from URLs; for example:
  ```
  Server.PresentationPrefix = "simpleApp/presentation"
  ```
- The maximum length (in minutes) of a user session and the session idle time; for example:
  ```
  SessionManager.Lifetime = 60
  SessionManager.MaxIdleTime = 2
  ```
- The default URL for the application; for example:
  ```
  Application.DefaultUrl = "Welcome.po"
  ```
  When a browser requests the application URL, this is the URL (typically a presentation object) that the Multiserver returns by default.

Using XMLC

XMLC, the Extensible Markup Language Compiler was introduced in Chapter 2, "Overview". It is a powerful tool that you can use to create applications that have a clean separation between the user interface and the back-end programming logic.

In general, XMLC can work with XML pages, but for practical reasons we are going to focus on how it works with HTML pages. XMLC parses an HTML file and creates a Java object that enables an application to change the HTML file's content at runtime, without regard for its formatting. The Java objects that XMLC creates have interfaces defined by the Document Object Model (DOM) standard from the World Wide Web Consortium (W3C).

For simplicity, the remainder of this chapter will refer to HTML documents, but in most cases the discussion can be extended to include XML documents in general.

Adding a Hit Counter

To get a feel for how XMLC works, you are going to extend your application to display a "hit counter" that shows the number of users who have accessed it.

Find the files `Welcome.html` and `Welcome.java` in the `presentation` directory.Add the following line of HTML to `Welcome.html` before the closing `</CENTER>` tag:

```
Number of hits on this page: <SPAN ID="HitCount">no count</SPAN>
```

The ID attribute tells XMLC to generate an object corresponding to the SPAN tag, so that it can replace the text "no count" at runtime.

Now, add the two lines of code indicated below to `Welcome.java` in the same directory.

```
import java.util.Date;
import com.lutris.xml.xmlc.*;
import com.lutris.appserver.server.httpPresentation.*;
public class Welcome implements HttpPresentation {
```

Add this:

```
  static int hitCount=0; // All Welcome PO's will share this.
  public void run(HttpPresentationComms comms)
    throws HttpPresentationException {
    String now = new Date().toString();
    WelcomeHTML welcome =
        (WelcomeHTML)comms.xmlcFactory.create(welcomeHTML.class);
    welcome.setTextTime(now);
    // Increment the count and write into the html.
```

Add this:

```
    welcome.setTextHitCount( String.valueOf( ++hitCount ) );
    comms.response.writeHTML(welcome);
  }
}
```

Build the application by running **make** from the top-level `simpleApp` directory. Then restart Enhydra:

```
cd /myapps/simpleApp
make
cd output
./start
```

Building the application with **make** runs XMLC on all the HTML files in the application, in this case, just `Welcome.html`.

Test the application by loading `http://localhost:9000` in your browser. You will see this:

Notice that the page now displays the number of times it has been accessed. Reload the page several times to verify that it works correctly.

The application is doing two things:

- Storing the hit count in **hitCount**, a static property of the **Welcome** presentation object
- Writing it to the web page with the **setTextHitCount( )** method

Recall that the presentation manager instantiates a presentation object for each request. So, the **Welcome** class is instantiated once per browser request. But because **hitCount** is a static property, it is shared by all **Welcome** objects, and its value gets incremented by each request.

In the same way that it added a **setTextTime** method for the `<SPAN ID="Time">` tag, XMLC creates a **setTextHitCount** method for the `<SPAN ID="HitCount">` tag. The application then uses the **setTextHitCount** method to write the value of **hitCount** into the page, within the corresponding SPAN tag.

Note XMLC creates the **WelcomeHTML** class, but by default it deletes the Java source file.

Understanding the DOM

HTML documents have a hierarchical or tree-like structure that can be modeled in an object-oriented language such as Java. The Worldwide Web Consortium's standard for the XML / HTML object model is called the Document Object Model (DOM).

Enhydra applications use the DOM to manipulate HTML content at runtime. For example, consider the following HTML:

```
<TABLE>
     <TR>
       <TD ID="cellOne">Shady Grove</TD>
       <TD ID="cellTwo">Aeolian</TD>
     </TR>
     <TR>
       <TD ID="cellThree">Over the River, Charlie</TD>
       <TD ID="cellFour">Dorian</TD>
     </TR>
</TABLE>
```

This HTML snippet has a TABLE tag that contains TR tags, that in turn contain TD tags containing text. This defines a tree-like hierarchy, as illustrated in Figure 4.1

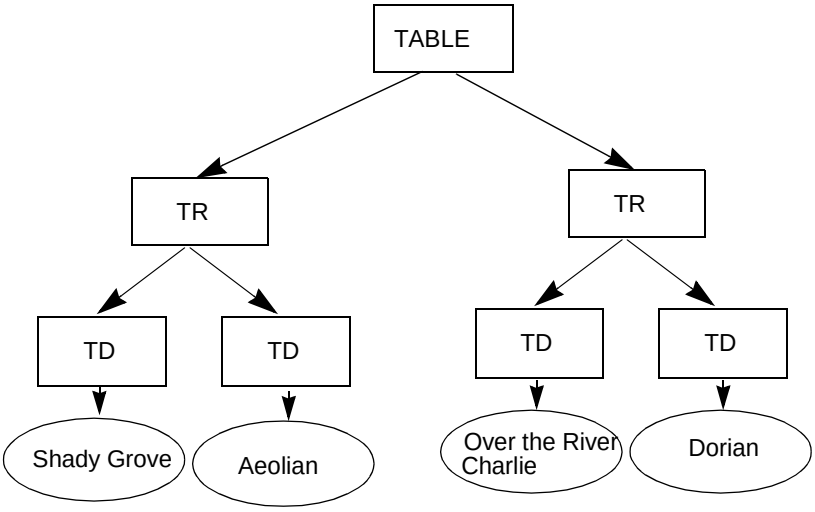


Figure 4.1 DOM tree of HTML

Each box or ellipse in the above figure is a *node* in the tree. The node above a node in the hierarchy is called is *parent*; the nodes below are called its *children*. Some nodes (like HTML tags) have *attributes*, for example a table cell has a background color attribute. The W3C defines packages and interfaces that mirror the object hierarchy of nodes in an HTML document. In addition, XMLC includes an API for changing attribute values.

You could use code like this to set the color of one of the table cells:

```
HTMLTableCellElement cellOne = theDocument.getElementCellOne();
cellOne.setBgColor("red");
```

The class **HTMLTableElement** and **setBgColor( )** came from the W3C packages, and **getElementCellOne( )** came from XMLC. This code illustrates one important thing XMLC does—create methods to access nodes in the DOM. XMLC generates the **getElementXXX( )** methods that return objects corresponding to tags with ID attributes. You could change the color of a table cell with the W3C classes alone, but

your code would have to traverse the DOM tree, so it would have been more laborious.

SPAN and DIV Tags

SPAN and DIV are HTML tags that you may not be familiar with. They are typically used to apply styles using Cascading Style Sheets. Outside of that, they are largely ignored by browsers. XMLC makes extensive use of them, however.

Use the SPAN tag to enclose a block of text that you want to replace at runtime. In general, a SPAN tag can enclose any text or *in-line* tag. An *in-line* tag is any tag that does not cause a line break in the layout, for example A (anchor) or B (bold) tags. Do not use SPAN tags to enclose other tags, such as TABLE or P (paragraph). Use the DIV tag to enclose *block* tags, such as TABLE, that do cause a line break in the HTML layout.

Using XMLC From the Command Line

Previously, you have run XMLC implicitly when you built the project with **make**. To run XMLC from the command line, you must have the enhydra.jar file on your CLASSPATH.This is something that the **make** files automatically do for you.

Set your CLASSPATH with one of the following commands:

Unix
```
export CLASSPATH=<enhyra_root>/lib/enhydra.jar:.
```

Windows
```
export CLASSPATH=<enhydra_root>/lib/enhydra.jar\;.
```

The basic command-line syntax of XMLC is:

```
XMLC -options file.html
```

where *options* is a set of command line options, and *file* is the name of the input file. There are several dozen command line options. In this section we introduce three immediately useful ones: **dump**, **class**, and **keep**.

The -dump Option

The **-dump** option makes XMLC display the DOM tree for a document. This is primarily useful as a learning tool; once you are familiar with XMLC you will rarely use it.

Create a new file called Simple.html in the simpleApp/simpleApp/presentation directory. Add the following HTML to it:

```
<HTML>
<HEAD><TITLE>Simple Enhydra Page</TITLE></HEAD>
<BODY>
<H1 ID="MyHeading">Ollie Says</H1>
The current time is <SPAN ID="Time">00:00:00</SPAN>.
</BODY>
</HTML>
```

Change to the presentation directory and enter this command:

```
xmlc -dump Simple.html
```

XMLC displays the following in the shell window:

```
DOM hierarchy:
  HTMLDocument:null DocumentType
    HTMLHtmlElement: HTML
      HTMLHeadElement: HEAD
        HTMLTitleElement: TITLE
          Text: XMLC Test
        HTMLBodyElement: BODY
          HTMLHeadingElement: H1: id='MyHeading'
            Text: Ollie Says
          Text: The current time is
          HTMLElement: SPAN: id='Time'
            Text: 00:00:00
          Text: .
```

Each line shows the DOM object name followed by a colon and then the corresponding HTML tag; if the tag has attributes they are listed following the tag in name/value pairs. For instance, **HTMLHeadingElement** is the DOM name for the H1 tag, and it has an id attribute with value "MyHeading." The level of indenting shows the object relationships, so for example, you can see that the first **HTMLHeadingElement** is the child of **HTMLBodyElement**.

The -class and -keep Options

By default, XMLC creates a class with the same name as the HTML file. So, for `Simple.html` it would create `Simple.java`. Use the **-class** option to specify a name for the class that XMLC creates.

By default, XMLC deletes the Java source file that it creates; leaving only the compiled class file. Use the **-keep** option to keep the Java source file. The source file is useful primarily for understanding how XMLC and the DOM API works.

To create a Java object named **SimpleHTML** for the HTML file `Simple.html`, and to keep the Java source, enter this command:

```
xmlc -keep -class simpleApp.presentation.SimpleHTML
Simple.html
```

XMLC generates two files: `SimpleHTML.java` and `SimpleHTML.class`.

Open `SimpleHTML.java`. At the beginning of the file, you will see two methods, **getElementMyHeading( )** and **getElementTime( )**. XMLC recognized the ID attributes of the heading and SPAN tags and generated these methods. XMLC also generated the method **setTextTime( )** for the SPAN tag. Peruse this file to get an idea of the object that XMLC creates for a very simple document.

You're done exploring how XMLC works for now. But keep your `Simple.html` file, because you are going to use it later in this tutorial.

Enhydra Programming

This section covers some more topics essential to Enhydra application development:

- Maintaining session state
- Adding a page to an application
- Populating an HTML table
- Adding a business object to the application

Maintaining Session State

Since HTTP is a stateless protocol, an application that needs to keep user-specific information across multiple requests must perform *session maintenance.* For an overview of how Enhydra performs session maintenance, see “Session Manager” on page 12.

Think of the user’s session as a container in which the application can store any information associated with a particular user. The class that you use as the container is **com.lutris.appserver.server.session.SessionData**; it is similar to a hash table in that it has a **set( )** method to which you pass a string key and an object to store, and a **get( )** method that returns the object, given the string key.

Enhydra matches a user to a particular **SessionData** object with a *session key*, a very long randomly-generated character string. When the Enhydra session manager first creates a **SessionData** object for a user, it generates a session key and stores it in its internal data structure. Enhydra also gives the session key to the client, either passed as a cookie or appended to the URL. The next time the client makes a request, the session manager uses the key to find that user’s **SessionData** object.

Generally, you don’t need to worry about the session key—Enhydra handles all those details for you “under the hood.” You do, however, need to keep track of the keyword strings that you use to get and set each object you want to save to the session.

To help you understand session maintenance, you are going to enhance your application so that the Welcome page displays the number of times a particular user has accessed it, in addition to the total “hits” on the page. For fun, you’ll also display the session key on the page.

Add these four lines of HTML just before the closing `</CENTER>` tag in `Welcome.html`:

```
<P>Number of hits from you:
<SPAN ID="PersonalHitCount">no count</SPAN>
<P>Session identifier:
<SPAN ID="SessionID">no count</SPAN>
```

Now add the following code to `Welcome.java`. Add this import statement:

```
import com.lutris.util.*;
```

Add this member property to the Welcome class (just after **hitCount**):

```
final String hits = "HITS";
```

The string “HITS” is the keyword that the application will use to save and recall the hit count information.

Now add the following code to the **run( )** method, just before the call to **comms.response.writeHTML( )**. You can find this code in the <*enhydra_root*>/doc/samples/SessionMaint.java file.

```
try {
  Integer personalHits = (Integer)comms.session.getSessionData().get(hits);
  if(personalHits == null) {
    personalHits = new Integer(1);
  } else {
    personalHits = new Integer(personalHits.intValue() + 1);
  }
  comms.session.getSessionData().set(hits, personalHits);
  // Save personalHits to the user’s session.

  welcome.setTextPersonalHitCount( personalHits.toString() );
  welcome.setTextSessionID( comms.session.getSessionKey() );
  // Shows the session key value used for session tracking.
} catch (KeywordValueException e) {
  comms.response.writeHTML("Session access error" + e.getMessage());
}
```

This code begins by calling **getSessionData( ).get(hits)** to get the value stored for the keyword string “HITS.” Because **SessionData** stores only generic **java.lagn.Objects**, you have to type cast it to **Integer**. If the object has not been previously stored in the session, the code creates a new **Integer** of value one. If it does exist, it is incremented.

The code then saves the **Integer** object back into the session with **setSessionData( ).set(hits, personalHits)** and writes the value into the web page with **getSessionKey( )**. Normally, you would not need to deal with the session key, but for curiosity’s sake this example shows you how to display it.

Rebuild and start the application, and access the page with your browser. The page now looks like this:

The page now displays the total number of hits as well as the number of hits from a particular client. Since you are running the application on your Localhost server, it is not accessible to any other clients, so these two numbers will always be the same. However, if it were running on a "real" server, you would see different numbers depending on how many times you had accessed the page versus the total number of hits. Notice also that the session ID string always stays the same.

Adding a New Page to the Application

Next, you are going to add a new page (HTML file and presentation object) to your application. You're going to use the little HTML file you created previously, `Simple.html`. In addition to learning how to add a page, you're also going to play around with the DOM a little bit to become more familiar with it.

First, you need to create a new presentation object. Copy the file `Welcome.java`, and call it `Simple.java`. Edit `Simple.java` and change the name of the class from **Welcome** to **Simple**, and then remove all the session-related code you added to Welcome. Change all the occurrences of **WelcomeHTML** to **SimpleHTML**; also change the **welcome** identifier to **simple**.

Now, you have a "stripped down" presentation object. The **run** method should look like this:

```
public void run(HttpPresentationComms comms) throws HttpPresentationException
{
  String now = new Date().toString();
  SimpleHTML simple = new SimpleHTML();
  simple.setTextTime(now);
```

```
  comms.response.writeHTML(simple);
}
```

Add the following statements at the top of the file, after the other **import** statements:

```
import org.w3c.dom.*;
import org.w3c.dom.html.*;
```

Now add the following lines just *before* the last statement in the **run( )** method:

```
HTMLHeadingElement heading = simple.getElementMyHeading();
heading.setAttribute( "align", "center" );
Text heading_text = (Text) heading.getFirstChild();
heading_text.setData( "Mr. Ollie Otter says:" );
```

This code does the following:

- Gets the **HTMLHeadingElement** object named "MyHeading," from the DOM
- Sets its `ALIGN` attribute to `CENTER`. This will center the heading on the page
- Gets the child object of the heading (a **Text** object)
- Sets a new value for the text, "Mr. Ollie Otter says:"

You could have done the same thing by putting a SPAN tag around the text in the heading, since then XMLC would have generated a **setTextMethod( )** that you could have then called in the code, but this example illustrates how to do it with the DOM.

Note This code performs some low-level DOM manipulation that you should normally not do in your application, because it violates the separation of presentation and business logic. It is only presented here to help explain the DOM.

Finally, edit `Makefile` in the `presentation` directory and add the new files to the `CLASSES` and `HTML_CLASSES` variables, to make them look like this:

```
CLASSES = \
  Redirect \
  Welcome \
  Simple
HTML_CLASSES = WelcomeHTML \
               SimpleHTML
```

The `HTML_CLASSES` variable is passed to the XMLC **-class** option to make it create the specified classes, just as you did on the command line. The Lutris convention is to create **fooHTML** class for a file `foo.html`. This convention is defined in `<enhydra_root>/lib/stdrules.mk`.

Save all the files and run **make** in the `presentation` directory to build the package.

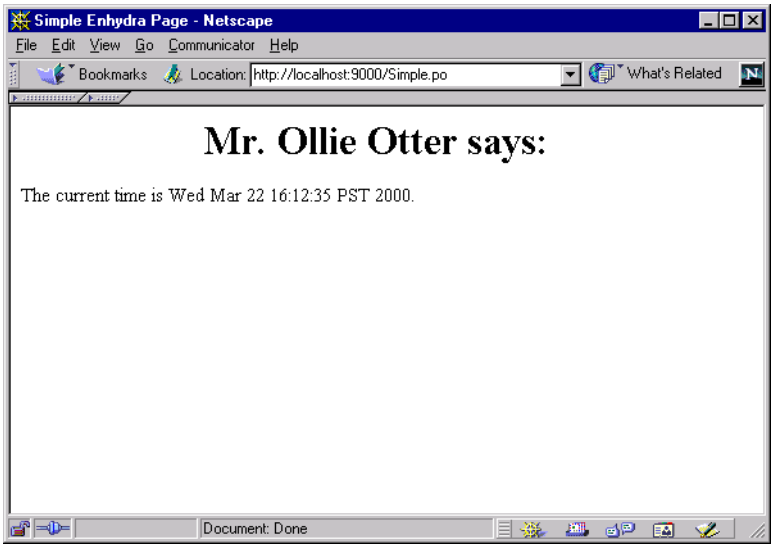
To create a link from the Welcome page to your new page, add the following HTML at the bottom of `Welcome.html`:

```
<A HREF="Simple.po">Go to Simple Page</A>
```

If you have not done so already, stop your Multiserver by hitting **ctrl-C** in the shell window, and build the application from the top level:

```
cd /myapps/simpleApp/simpleApp
make
```

Now access the application from your browser, as you did before. Click on the link "Go to simple page" to view the Simple PO. You will see the following rather uninteresting page:



Don't worry, though: you're going to make this page more interesting in the next section.

Populating a table

Another common task in web application development is how to populate an HTML table with dynamic data. This section discusses populating a table using a static String array as the data source; in a later section, you will modify the code to get data from a database.

Follow these steps to populate a table:

- In the `Simple.html` file, create an HTML table with a template row with an ID attribute
- In the corresponding presentation object, programmatically populate the table.
- Rebuild and run the application.

Create the table in HTML

Edit the file `presentation/Simple.html` in your simpleApp project. Add the HTML shown below just before the end of the BODY tag.

Note If you don't want to type in all this HTML, you can copy and paste it from `<enhydra_root>/doc/GettingStarted/samples/TableCode.html`.

```
<H2 align=center>Disc List</H2>
<TABLE border=3>
<TR>
```

```
<TH>Artist</TH> <TH>Title</TH> <TH>Genre</TH>
<TH>I Like This Disc</TH>
  </TR>
<TR id=TemplateRow>
<TD><SPAN id=Artist>Van Halen</SPAN></TD>
<TD><SPAN id=Title>Fair Warning</SPAN></TD>
<TD><SPAN id=Genre>Good Stuff</SPAN></TD>
<TD><SPAN id=LikeThisDisc>Yes</SPAN></TD>
</TR>
</TABLE>
```

This HTML contains a table with a single "template" row (`TR` tag). Notice that both this row and the `SPAN` tags enclosing the cell contents have ID attributes. This is called a template row, because it is used as a model from which you construct further rows of the table.

Programmatically Populate the Table

Now copy the file *<enhydra_root>*`/doc/GettingStarted/samples/TableCode.java`. into your application's presentation directory, and rename it `Simple.java`. If you like, you can save your old `Simple.java` to `Simple.sav` for future reference.

Now, look at your new `Simple.java`: In addition to the standard features of a presentation object, the first thing you'll notice in this code is a member property of that is an array of strings representing the content the application will use to populate the table. This array takes the place of a database result set for this example:

```
String[][] discList =
  { { "Felonious Monk Fish", "Deep Sea Blues", "Blues", "Yes" },
   { "Funky Urchin", "Lovely Spines", "Techno Pop", "Yes" },
   { "Stinky Pups", "Shark Attack", "Hardcore", "Not" } };
```

The next new section of code gets the document objects for the table elements:

```
HTMLTableRowElement templateRow = simple.getElementTemplateRow();
HTMLElement artistCellTemplate = simple.getElementArtist();
HTMLElement titleCellTemplate = simple.getElementTitle();
HTMLElement genreCellTemplate = simple.getElementGenre();
HTMLElement likeThisDisc = simple.getElementLikeThisDisc();
```

The next section of code removes the ID attributes from these objects. The reason for this is that the DOM requires that each ID in the document be unique; when you make a copy of the table row, you would otherwise have duplicate IDs. The **removeAttribute( )** method removes the attribute with the specified name:

```
templateRow.removeAttribute("id");
artistCellTemplate.removeAttribute("id");
titleCellTemplate.removeAttribute("id");
genreCellTemplate.removeAttribute("id");
likeThisDisc.removeAttribute("id");
```

Then, a call to **getParentNode( )** gets a reference to the table document object, which you'll be using later:

```
Node discTable = templateRow.getParentNode();
```

Next comes the heart of the code, a **for** loop that iterates through each "row" in the "result set," puts text in each cell in the table row, and then appends a copy (or *clone*) of the row to the table:

```
for (int numDiscs = 0; numDiscs < discList.length; numDiscs++) {
  simple.setTextArtist(discList[numDiscs][0]);
  simple.setTextTitle(discList[numDiscs][1]);
  simple.setTextGenre(discList[numDiscs][2]);
  simple.setTextLikeThisDisc(discList[numDiscs][3]);
  discTable.appendChild(templateRow.cloneNode(true));
}
```

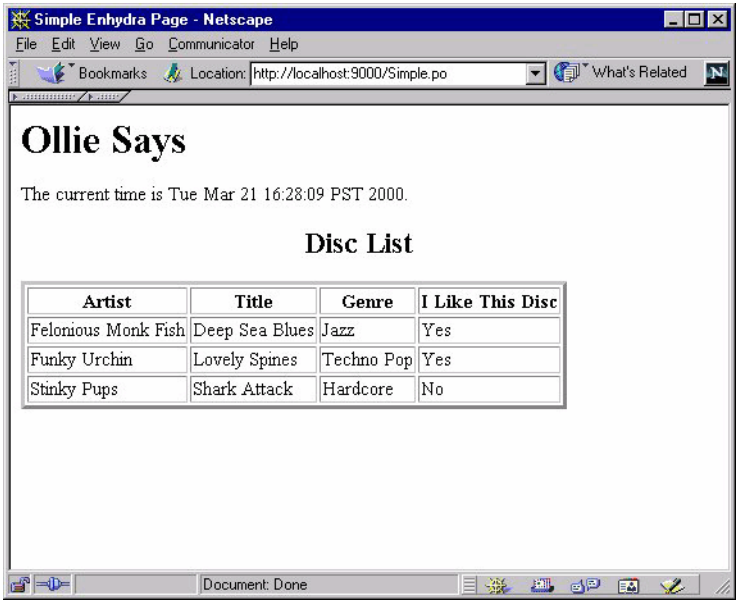
That last statement is crucial: The **cloneNode( )** method creates a copy of the **Node** object which calls it, in this case, **templateRow**. The boolean argument determines if it copies only the node itself or the node and all its children, and their children, and so on. In this example, the argument is **true**, because you want to copy the row and its child nodes (the table cells and the text inside them).

Finally, **removeChild( )** removes the template row from the table. This ensures that the "dummy data" in the template does not show up in the runtime page.

```
discTable.removeChild(templateRow);
```

Rebuild and Run the Application

Now rebuild the application, and load the page in your browser. You should see the following result:



Adding a Business Object

So far, your application has three objects: the **SimpleApp** application object, and two presentation objects, **Welcome** and **Simple**. Now, you are going to add a business object that you will use it in the following sections. This will not change what the application displays.

The business object represents a list of discs; this is not terribly useful, but it does suffice to illustrate a basic role of business objects as you proceed.

Create a new file called `SimpleDiscList.java` in the business directory, `simpleApp/simpleApp/business`. It's in your application's business package, so the first line in the file will be:

```
package simpleApp.business;
```

Now, add the following lines (cut and paste the array initializer from the **Simple** class, but be sure to add the underscore in front of the identifier):

```
public class SimpleDiscList {
  String[][] _discList =
    { { "Felonious Monk Fish", "Deep Sea Blues", "Blues", "Yes" },
      { "Funky Urchin", "Lovely Spines", "Techno Pop", "Yes" },
      { "Stinky Pups", "Shark Attack", "Hardcore", "Not" } };
  public SimpleDiscList() {
  }
  public String[][] getDiscList() {
      return _discList;
  }
}
```

To ensure that this file gets compiled when you build the project, edit the **make** file in the `business` directory, and add the name of the file to the CLASSES variable:

```
CLASSES =   \
SimpleDiscList
```

This **make** file is automatically included by the top-level **make** file, so that is all you have to do to add the file to the project. Make sure the file compiles by entering **make** in the `business` directory.

Now, back in the `presentation` directory, edit `Simple.java` as follows:

- Import the new class:
  ```
  import simpleApp.business.SimpleDiscList;
  ```
- Add these two lines to create an instance of your new business object, and call its **getDiscList( )** method. These lines take the place of the static array initializer in the previous section.
  ```
  SimpleDiscList sdl = new SimpleDiscList();
  String[][] discList = sdl.getDiscList();
  ```

Rebuild and test your application.

You won't see anything different, but you have extracted some functionality out of the presentation object into the new business object. This will come in handy in an upcoming section: when you replace the static array with a real database query, you

won't have to change your presentation class, because the business object provides a buffer between it and the data layer.

Connecting the Application to a Database

Enhydra uses Java Database Connectivity (JDBC), a standard Java API, to communicate with databases. Enhydra can connect to any JDBC-compliant database, such as Oracle, Sybase, Informix, Microsoft SQL Server, PostgreSQL, and InstantDB.

Before you can proceed to connect the application to a database, you are going to take a brief detour to lay some groundwork. In particular, you are going to:

- Create the database table used by the application
- Establish and test the JDBC connection to your database
- Configure Enhydra's database manager to connect to your database through JDBC

Creating a Database Table

The remainder of this section requires the existence of a specific table in your data, so you need to create that table before proceeding. Most databases provide a tool for directly executing SQL statements. For example, Oracle supplies SQL*Plus. Use this tool to execute the provided SQL file and create the table in your database. The SQL file is in `<enhydra_root>/doc/GettingStarted/samples/tutorial_create.sql`

Here is the command for Oracle SQL*Plus:

```
SQL> @<enhydra_root>/doc/GettingStarted/samples/tutorial_create.sql
```

This SQL file contains a `CREATE TABLE` statement to create a simple table, `LE_TUTORIAL_DISCS`, and some `INSERT` statements to populate it with data. You are going to use this table in the forthcoming sections of the tutorial.

If you are using a database other than Oracle, refer to your database documentation for instructions on how to execute a SQL file or create tables.

Establishing a JDBC Connection

Before you can create a database application, you need to establish a JDBC connection from your system to the database server, which may be running on a different system.

This section shows you how to write a simple stand-alone program to establish a JDBC connection to database server. Starting with a stand-alone program enables you to isolate any problems that occur.

This example uses an Oracle database. If you are using a different database, refer to your database's documentation for more specific information

Note If you have already configured JDBC on your system, you can skip this section.

1. Configure your database to “listen” on a specific TCP port. Most database servers listen on a certain port by default (for example, Oracle uses port 1521). See your database's documentation for details.
2. Install the JDBC driver for your database. This consists of a `zip` or `jar` file containing the database-specific JDBC classes, for example the Oracle JDBC driver is `classes111.zip`. See your database's documentation for details.

Windows The Cygnus tools require the JDBC driver to be on the C drive, so make sure the JDBC driver library is on the C drive.

3. Determine the connection string for your database. The connection string contains:
   - The name of the system running the database server
   - The TCP port on which the database server is listening
   - The name of the database instance

   Every database has its own format. For example, the connection string for the Oracle driver is
   `jdbc:oracle:thin:@db_host:xxxx:db_inst`

   where `db_host` is the name of the database server, `xxxx` is the port number (1521, by default), and `db_inst` is the name of the database instance. You will also need your database username and password. Use these values as illustrated in the call to **DriverManager.getConnection( )** in the code below

   Use the following simple program to test your JDBC connection:

```
import java.sql.*;
public class JDBCTest {
  public static void main( String[] args ){
    Connection con  = null;
    Statement  stmt = null;
    ResultSet  rs   = null;
// Load the driver, get a connection, create statement, run query, and print.
    try {
      Class.forName("oracle.jdbc.driver.OracleDriver");
      con = DriverManager.getConnection(
        "jdbc:oracle:thin:@your_computer:1521:your_sid" ,"name", "pass" );
      stmt = con.createStatement();
      rs = stmt.executeQuery("SELECT * FROM LE_TUTORIAL_DISCS");
      rs.next();
      System.out.println("Title = " + rs.getString("title") +
        " -- Artist = " + rs.getString("artist"));
    }
    catch(ClassNotFoundException e) {
      System.err.println("Couldn't load the driver: "+ e.getMessage());
    }
    catch(SQLException e) {
      System.err.println("SQLException caught: "+ e.getMessage());
    }
  }
}
```

To use this program:

1. Create a directory with a name such as `/tmp/jdbcTest`.

   ```
   cd /tmp
   mkdir jdbcTest
   cd jdbcTest
   ```

2. Copy the above code from *<enhydra_root>*`/doc/GettingStarted/samples/JDBCTest.java` into the new directory.

3. Edit the file to put in your connect string, username and password. These appear in the call to **getConnection( )**, the second statement in the **try** block.

4. Compile the file:
   ```
   javac JDBCTest.java
   ```

5. Set your `CLASSPATH` so the JVM can find your JDBC driver and the test program. For an Enhydra application, this would normally be performed in an application's start script; however, since this is a stand-alone test application, you'll just do it in the shell window. On Unix, enter this command:
   ```
   export CLASSPATH=<path_to_your_JDBC_driver>:.
   ```
   On Windows, enter this command:
   ```
   export CLASSPATH=<path_to_your_JDBC_driver>\;.
   ```

6. Run the program:
   ```
   java JDBCTest
   ```

If you have populated the table as instructed previously, you will see the following in the shell window:

```
Title = Rockin Apps -- Artist = Enhydra Orchestra
```

If there was an error, you will see some exception messages in the shell window that should help you isolate the problem. Also, refer to your database's JDBC documentation or the Enhydra mailing list.

Configuring the Application to use JDBC

To make the JDBC classes available to your Enhydra application, you must put the JDBC driver in the `CLASSPATH` system variable. To do this, set `CLASSPATH` in the application's start script. Note there are two copies of the start script: one in the application's source directory and one in the `output` directory. As with `application.conf`, building the application overwrites the one in the `output` directory.

Note Enhydra has its own class loader, so if you put the JDBC driver in the `CLASSPATH` by specifying it in the application's configuration file, the driver will not work.

So, edit the file `simpleApp/simpleApp/start` and add these lines just before the last line in the file:

```
CLASSPATH="JDBC_LIB"
export CLASSPATH
```

where *`JDBC_LIB`* is the JDBC driver library (generally a `jar` or `zip` file), including the file path. For example,

```
CLASSPATH="/myapps/lib/classes111.zip"
```

Be careful not to put any blank spaces in this line, since they will prevent it from working properly.

Configuring the Database Manager

Now that you have verified your JDBC connection, you need to provide the database connection parameters to your application. You do this in the application's configuration file, `appName.conf`. In this example, the file is `simpleApp.conf`. The **make** utility copies this file to the `output` directory after every build, so be sure to edit the file in `simpleApp/simpleApp`, not the version in `simpleApp/output`.

Open `simpleApp/simpleApp/simpleApp.conf` in a text editor. Add the following lines to the bottom of the file:

```
#-------------------------------------------------------------
#                   Database Manager Configuration
#-------------------------------------------------------------------
DatabaseManager.Databases[] = "id"
DatabaseManager.DefaultDatabase = "id"
DatabaseManager.Debug = "false"
DatabaseManager.DB.id.ClassType = "Oracle"
DatabaseManager.DB.id.JdbcDriver = "oracle.jdbc.driver.OracleDriver"
DatabaseManager.DB.id.Connection.Url = "jdbc:oracle:thin:@db_svr:1521:db_inst"
DatabaseManager.DB.id.Connection.User = "User"
DatabaseManager.DB.id.Connection.Password = "Password"
DatabaseManager.DB.id.Connection.MaxPreparedStatements = 10
DatabaseManager.DB.id.Connection.MaxPoolSize = 30
DatabaseManager.DB.id.Connection.AllocationTimeout = 10000
DatabaseManager.DB.id.Connection.Logging = false
DatabaseManager.DB.id.ObjectId.CacheSize = 20
DatabaseManager.DB.id.ObjectId.MinValue = 1
```

Note This is an example of the configuration file for Oracle; for information on configuring for other databases, see Appendix A, "Database Configuration" on page 79.

Change all of the items shown in bold to match your connection parameters as follows:

- The `id` identifier is used in the configuration file to identify the database connection. You can use any string here, as long as it is consistent throughout the file.
- The connection string is **`jdbc:oracle:thin:@db_svr:1521:db_inst`**. This is the format of the Oracle connection string, where `db_svr` is the database server, 1521 is the TCP port on which the server is listening, and `db_inst` is the name of your database instance.
- Set **`User`** to your database user name
- Set **`Password`** to your database password

Make sure there is a carriage return at the end of the file; this is required for the file to work properly.

After you edit the configuration file, run **make** to propagate the changes to the file in the output directory.

Adding Data Access

Now that you have laid the groundwork, you are ready to add data access to simpleApp. This section describes how to add a simple data object with embedded SQL that replaces the static array used in “Populating a table” on page 39. The next section, “Using DODS”, describes how to build a “real” data layer for the application using DODS.

First, you are going to create a data object that queries the database. Copy the file SimpleDiscQuery.java from <*enhydra_root*>/doc/GettingStarted/samples to your data layer, that is, the simpleApp/simpleApp/data directory. Remember to edit the **make** file in the data directory to add the file to the CLASSES variable, just as you did in “Adding a Business Object” on page 42.

Take a look at SimpleDiscQuery.java. In particular, notice the import statement right at the top:

```
import java.sql.*;
```

This tells you right away that this class is going to use JDBC. In addition to the constructor, there is only one other method, **query( )**, where the object performs most of its real work.

The constructor has essentially one statement:

```
c = Enhydra.getDatabaseManager().allocateConnection();
```

This statement tells the Enhydra database manager to allocate a database connection. Then, the **query( )** method calls **executeQuery** on the connection to execute the SQL query statement:

```
resultSet = connection.executeQuery("SELECT * FROM
LE_TUTORIAL_DISCS");
```

The remainder of the code in **query( )** iterates through the result set returned by the SELECT statement, and returns it in the form of a Vector of Vectors. Although each row is known to contain only four elements (since there are four columns in the table), the number of rows is unknown in general, which is why the method returns a Vector.

However, you will recall that the presentation object **Simple** expects the data to be in the form of a two-dimensional array of Strings. So, the **SimpleDiscList** needs to perform some conversion.

Edit the file for the business object you created previously, SimpleDiscList.java.

Now, find the file <*enhydra_root*>/doc/GettingStarted/samples/SimpleDiscList.java. You can simply replace your old SimpleDiscList.java with this file, or if you prefer, you can make the changes manually:

1 Add two import statements at the top:

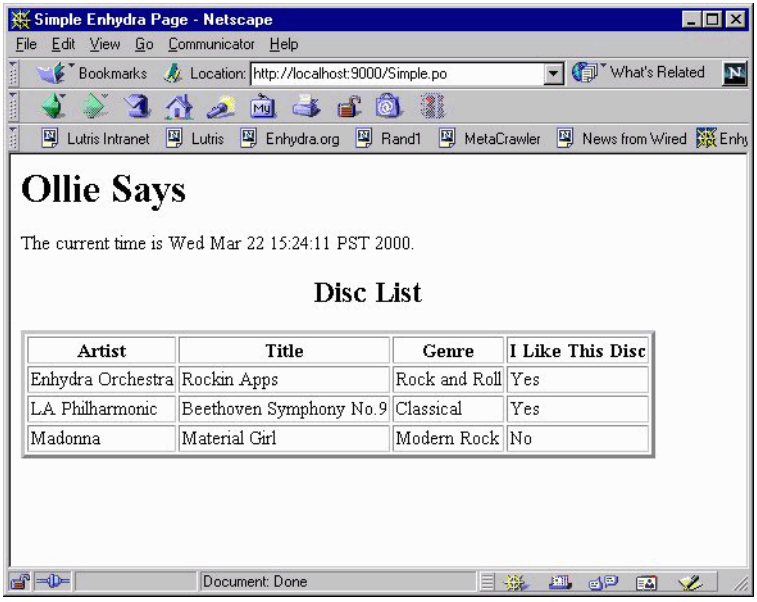
```
import simpleApp.data.SimpleDiscQuery;
import java.util.*;
```

2 Add a member variable corresponding to the data object:
```
SimpleDiscQuery _sdq;
```

3 Replace the body of the **getDiscList( )** method with the code in the new file. It converts the Vector of Vectors returned by **query( )** to a two-dimensional String array that the presentation object expects.

Notice that you did not have to change the presentation object at all. The data object provides a buffer between the PO and the data object.

Now, build the application from the top level. When you get it to compile, try running it. If everything is in place, you will see the following:



Notice the discs displayed in the table have the values from the database, not the static array. You've created your first database query page!

Using DODS

The Data Object Design Studio (DODS) is an graphical object-relational mapping tool that generates SQL code to create tables in a database and the corresponding application code to access the tables. DODS creates code that is specific to different databases, so you don't have to learn the nuances of each database. DODS also handles common issues such as transactions and data integrity.

DODS is most useful when you are creating a database from scratch, or when you are free to modify the database schema, as explained in "Loading the Schema" on page 54.

You are not required to use DODS in developing a Enhydra database application, but it does significantly simplify the process.

Introduction

This section introduces DODS and explains how to use it to create the database schema and associated data objects for the DiscRack application (see Chapter 5, “The DiscRack Example Application”). This example uses an Oracle database.

Start DODS by entering the following command:

```
dods
```

Note If DODS does not run when you enter this command, you need to set your PATH. See “Setting the PATH Environment Variable” on page 22.

You will see the DODS graphical interface. Open the DODS file for the DiscRack project by choosing File | Open, and selecting *<enhydra_root>*/examples/DiscRack/discRack.doml.

DODS will show the DiscRack data model:

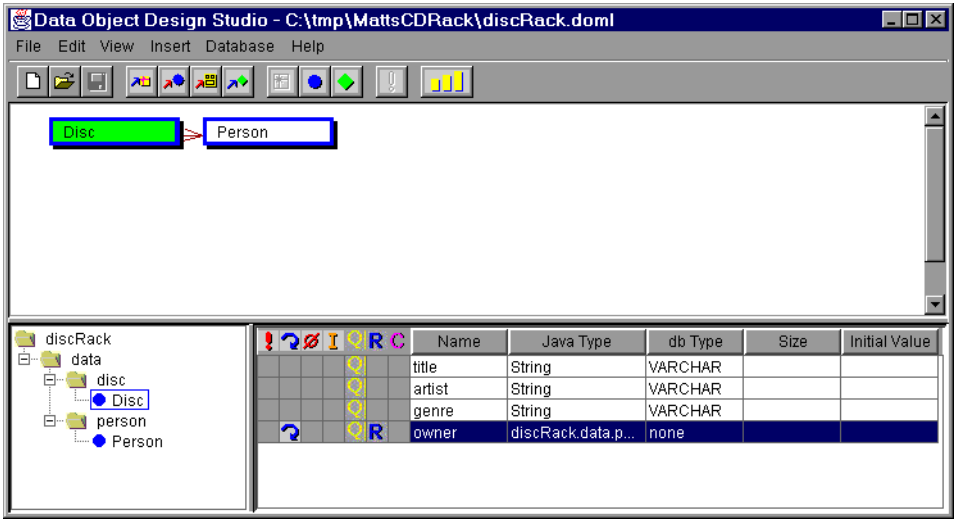


Figure 4.2 DODS with the discRack.doml loaded.

The DODS window has three sub-panels:

- At the top, the *object panel* shows the entity-relationship model. Each data object (corresponding to a table in the database) is represented by a box with a name inside. Relationships between data objects are shown by red lines between the boxes.
- In the lower left, the *package panel* displays the package hierarchy in the current data model, with a directory for each object / table in the data model, and a blue dot signifying objects in each package.
- In the lower right, the *attribute panel* shows the attributes (properties or columns) of the selected data object. Each attribute is represented by a row in the grid, with various information, such as the data type or nullability of the attribute, indicated in columns of the grid.

The schema of the discRack database is pictured in Figure 4.2. DODS shows the features common to both the database schema and the object model. For example, the **disc** data object has **title** and **artist** fields, that are the properties (members) of the Java class as well as the columns of the corresponding database table.

DODS creates Java code for object operations and SQL code for database operations, for example the one-to-many relationship between person and discs.

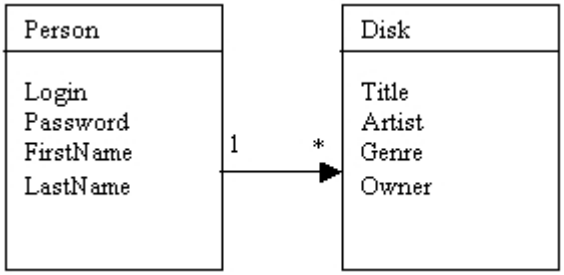


Figure 4.3 Disc Rack object-model/schema

Creating Data Objects

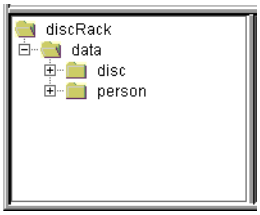
Now you are going to create the data layer for the discRack application from scratch.

1. Create a new data model by choosing File | New. DODS will close any open data model file and start a new one containing a single package, named “root” by default.
2. Select your database type. Using the Database menu, select your database type. For example, if you are using Oracle, choose Database | Oracle. This will ensure that the SQL statements that DODS generates have the correct syntax for your database.
3. Now create the package hierarchy.

   Begin by changing the name of the root package. In the lower left panel, select the `root` folder then choose Edit | Package. Enter the name of this application in the dialog box, “simpleApp,” then hit Enter.

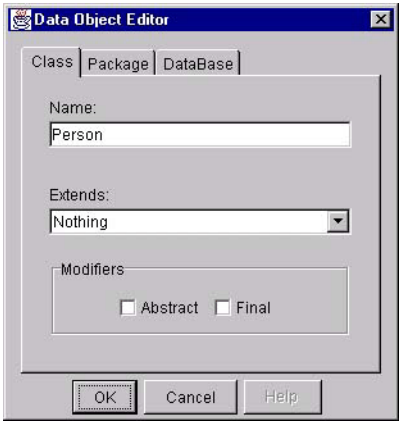
   Add the data package: select the discRack folder, choose Insert | Package, type “data,” then hit Enter.

   Similarly, create `disc` and `person` directories as subdirectories of the `data` directory. Recall that package names in Java begin with a lowercase letter. The package panel (in the lower left of the window) should now look like this:



4 Create the Person data object.

Select the person package in the lower left pane then choose Insert | Data Object. You will see the Data Object Editor dialog box.

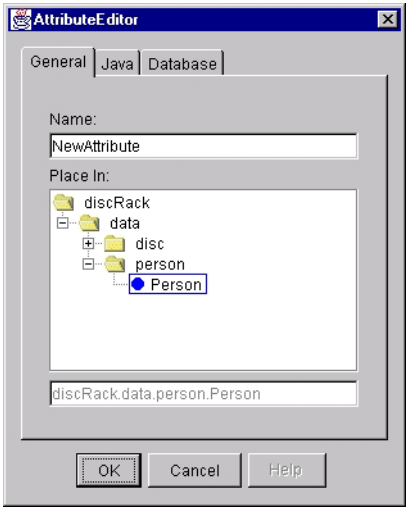


Change the defaults as follows:

- In the Class tab, change the name to “Person”
- In the Package tab, select the person package
- In the DataBase tab, change the "db Table Name" to “Person”

Leave the rest of the defaults and click OK. This will create both a Person table in the database and a Person object in the Java application. Note the class is uppercase “Person” while the package name is lowercase “person.”

Now add the login attribute to the Person data object. Select the Person data object (indicated by a blue dot) and click Insert | Attribute. The Attribute Editor window comes up:



Change the defaults as follows:

- In the General tab, change the name to login.
- In the Java tab, note the default Java data type is string, corresponding to the SQL VARCHAR data type. Leave this value.
- In the Database tab, select the “Can be queried” check box.

Leave the rest of the defaults and click OK.

Notice that the attributes appear in the attribute panel in the lower right of the DODS window. The fields show the different options you can set in the Attribute Editor; for example, the yellow "Q" means that the data object can be queried. Move the cursor over the other symbols to see what they represent.

Repeat this procedure to add three more attributes to the person data object: “password,” “firstname,” and “lastname.” You may want to use the Add Attribute button on the toolbar.

5 Create the Disc data object similarly to the procedure in the previous step. Add the following attributes: title, artist, and genre.

Adding the owner attribute is a little more complicated because there is a one to-many relationship between Person and Disc based on the owner attribute. Select the Disc data object and choose Insert | Attribute. Change the name to Owner. Click on the Java tab and then on the Java Type list box. Scroll down and select "discRack.data.person.PersonDO."

Click on Database tab. Select "Can be queried" and "Referenced DO must exist." Choose OK to close the Attribute Editor.

You will notice a red arrow in the interface between the Disc object and Person object showing that Disc uses Person. Notice the attribute pane displays icons indicating that the owner attribute has an object reference and a referential constraint.

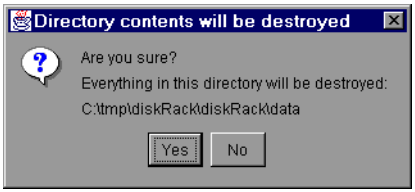
6 Save and build the data model.

Save the data model file. DODS stores the data model specification in a doml file, with an XML-like syntax. Select File | Save and choose a location for the doml file. Save it in the top level project directory.

Now that you have defined all the data objects, DODS is ready to generate and compile the code. The "Build All" command causes DODS to overwrite the existing data directory with all the new information from the data model, so if you want to save your old data layer code, copy it to another directory.

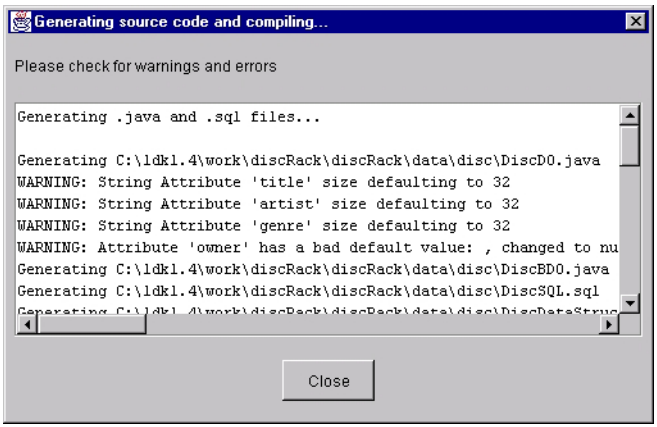
Now, the big moment you have been waiting for… select File | Build All. The "Destroy and Create New DODS Directory" dialog box appears. Select the data

directory of your simpleApp project. Choose the Re/Create Directory button. You will see the following warning dialog box:



Choose Yes, and the build will begin. DODS will align the directory structure in the GUI with the directory structure in the project, generate Java and SQL files, and then run **make** to build the data layer.

As it proceeds, DODS displays messages in a dialog box:



If the build is successful, you will see the message “DODS BUILD COMPLETE.” If the build fails, look at the messages in the output screen. Verify that the paths of the files are correct. You can also search the doml file for clues, since it is easy to read.

DODS generates the following files in the `data` directory:

- The `disc` and `person` directories, which contain the Java code for the disc and person data objects, respectively, and an SQL file defining the corresponding database table.
- The `create_tables.sql` and `drop_tables.sql` files, which contain standard SQL statements to create and remove, the disc and person tables from a database, respectively.
- Two **make** files: `Makefile`, and `config.mk`.
- The `classes` directory, which is initially empty.

Each data object directory contains Java source files to create four classes. For example, the person data object contain **personDO**, **personQuery**,

personDataStruct, and **personBDO**. The data object and the query classes are the most commonly used classes.

DODS also generates **make** files for the data layer. This allows you to compile the data layer independently or along with the entire project. The empty classes directory is only used if you compile the data layer separately.

Loading the Schema

The next step in the process is to run the SQL script that DODS generated to create the tables in the database. Figure 4.4 illustrates the complete schema generated by DODS:

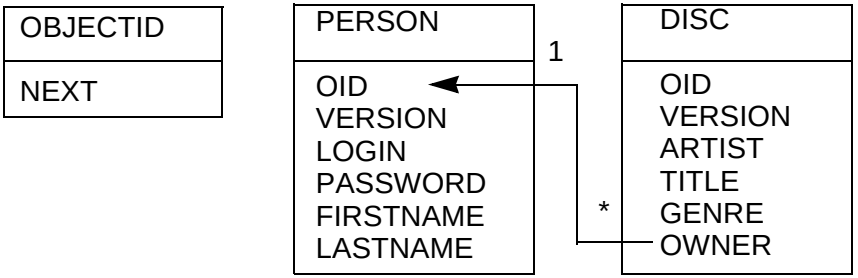


Figure 4.4 Disc Rack database schema generated by DODS

Notice there are some differences from the original database schema:

- The DISC and PERSON tables have two additional fields, OID and VERSION;
- There is a third table, OBJECTID, that contains one column, NEXT, with a single row.

The OID column is the primary key for each table created by DODS. The application code generated by DODS ensures that every row has a value of OID that is unique within the database. Whenever a new row is added to a table, the application generates a unique Object ID to put in the OID column; it uses the OBJECTID table to keep track of the next Object ID to be assigned.

DODS application code uses the VERSION column in each table to ensure that the data that an application is updating is accurate. Because many users can be accessing the database simultaneously, a record can change between the time an application retrieves it when it attempts to change the record.

Every time an application updates a row, it increments the VERSION column in the database. The application qualifies updates on both the VERSION and OID columns—if it finds that there are no rows that have the expected values, then it knows that another process has changed the row it is trying to update, and it throws an exception. You can catch the exception in your application code to handle such situations appropriately.

Procedure

To load the SQL scripts that DODS creates, follow these steps:

1 Edit the `create_tables.sql` file in the *<DiscRack_root>*`/discRack/data` directory.

The file contains the SQL CREATE TABLE commands to create the PERSON, DISC, and OBJECTID tables:

```
create table person
(
   login VARCHAR2(32) DEFAULT '' NOT NULL   ,
   password VARCHAR2(32) DEFAULT '' NOT NULL   ,
   firstname VARCHAR2(32) DEFAULT '' NOT NULL   ,
   lastname VARCHAR2(32) DEFAULT '' NOT NULL   ,
   oid DECIMAL(19,0) NOT NULL PRIMARY KEY,
   version INT NOT NULL
);

create table Disc
(
   title VARCHAR2(32) DEFAULT '' NOT NULL   ,
   artist VARCHAR2(32) DEFAULT '' NOT NULL   ,
   genre VARCHAR2(32) DEFAULT '' NOT NULL   ,
   owner DECIMAL(19,0)  NOT NULL  REFERENCES person ( oid )
     ON DELETE CASCADE,
   isLiked INTEGER DEFAULT 0 NOT NULL   ,
   oid DECIMAL(19,0) NOT NULL PRIMARY KEY,
   version INT NOT NULL
);

create table objectid
(
   next DECIMAL(19,0) NOT NULL
);
```

Note DODS may generate SQL files that are not fully compatible with your database server. You may have to edit the file manually to remove extraneous text that may be causing errors when reading the file. For example, for Oracle, you may have to remove extra blank lines.

You can configure many things about the SQL that DODS generates in the configuration file *<enhydra_root>*`/dods/dods.conf`. For example, by default DODS generates C-style comments, but you can change the style of comments if your database requires a different format.

2 Load the tables into the database. For example, the command for SQL*Plus is:
`SQL> @<`*enhydra_root*`>/examples/DiscRack/discRack/data/create_tables.sql`

3 Add some dummy data to the database for testing purposes. The file `<enhydra_root>/doc/GettingStarted/samples/tutorial_insert.sql` contains some

sample data, including one person and several discs. Here is the command for SQL*Plus:

```
SQL> @/<enhydra_root>/doc/GettingStarted/samples/
tutorial_insert.sql
```

Using the DODS Data Objects

Now all you need to do is modify the business object, **SimpleDiscList**, to use the DODS data objects instead of the simplified one you created previously. Replace your old `SimpleDiscList.java` with the file `<enhydra_root>/doc/GettingStarted/samples/SimpleDiscList.java`.

The main difference between the old and new objects is in the **getDiscList( )** method; here is the heart of it:

```
...
try {
  DiscDO[] discArray;
  DiscQuery dquery = new DiscQuery();
  discArray = dquery.getDOArray();
  String result[][] = new String[4][discArray.length];
  for(int i=0; i< discArray.length; i++) {
    result[0][i] = (String)discArray[i].getTitle();
    result[1][i] = (String)discArray[i].getArtist();
    result[2][i] = (String)discArray[i].getGenre();
    result[3][i] = discArray[i].getIsLiked() ? "Yes" : "No";
  }
}
return result;
...
```

This code uses the **DiscQuery** and **DiscDO** objects in the **data.disc** package to get data from the database. **DiscQuery** provides a set of methods for querying the DISC table; by default it performs the equivalent of SELECT * FROM DISC. It has methods that you can use to qualify the query (the WHERE clause of the SELECT statement) and order the result set. The **getDOArray( )** method returns an array of **DiscDO** objects returned from the query.

The **DiscDO** object is the basic data object representing a row of data from the DISC table. It has getter and setter methods for each column in the table. The above code only uses the getter methods **getTitle( )**, **getArtist( )**, **getGenre( )**, and **getIsLiked( )**, which returns a boolean value. All of them except **getIsLiked( )** return a string, so the method performs some simple logic to translate the boolean value to the appropriate string.

Run the Application

The last step is to recompile the project, by running **make** at the top level. Then start the multiserver with the start command, and load `http://localhost:9000` in your browser.

When you access the Simple page, you should see the following in your browser:

If you don't see this page, check the following:

1. Look in the `discRack.conf` file in the `output` directory to be sure that the database settings are correctly listed.
2. Check the output displayed in the shell window when you start the Multiserver for errors. If the database settings are in `discRack.conf` and the JDBC driver is in the application's `CLASSPATH`, there should be no errors listed when multiserver starts.
3. Re-run the JDBC connection test to verify that the database is correct and JDBC is working.
4. Try putting the wrong password into the application configuration file. Multiserver should start, but the application will return an SQL exception and a stack trace.
5. Make sure you do not have any extraneous Java VMs running. Sometimes, the classloader can fail to find the correct classes if it picks up an old `CLASSPATH` from a running VM.

CHAPTER

5

The DiscRack Example Application

This chapter introduces the DiscRack application, and uses it as a comprehensive example to illustrate some key concepts of Enhydra application development.

Building and Running DiscRack

Enhydra 3.0 includes the DiscRack application, which is installed to the *<enhydra_root>*/examples/DiscRack directory. Throughout this chapter, this top-level directory containing DiscRack will be referred to as *<DiscRack_root>*.

To build and run DiscRack, you need to make the following modifications to the installed files:

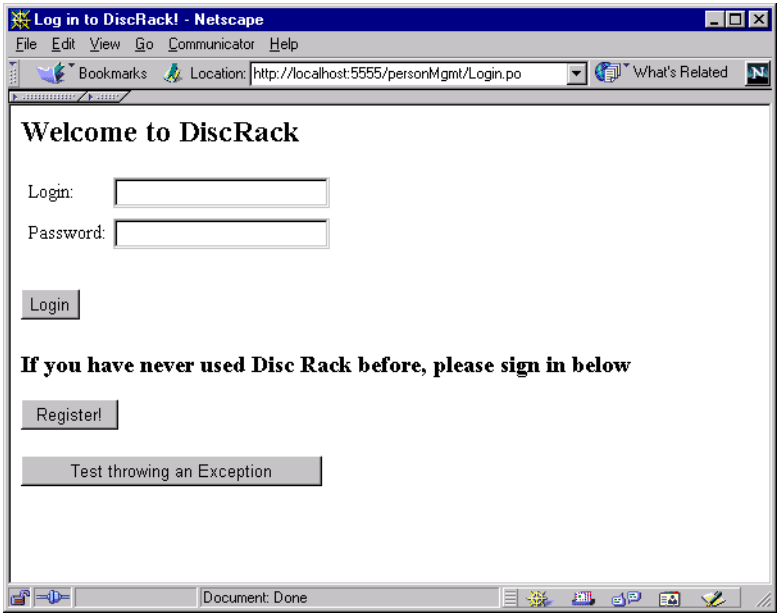
- Edit config.mk in *<DiscRack_root>* and change the ENHYDRA_DIR variable to your Enhydra root directory.
- Edit the configuration file discRack.conf.in in *<DiscRack_root>*/discRack, and make sure all the Database Manager configuration settings are correct, as described in "Configuring the Database Manager" on page 46. Also refer to Appendix A, "Database Configuration."
- Build the application by entering the **make** command from the *<DiscRack_root>* directory.
- Edit the start script in *<DiscRack_root>*/discRack, and make sure the CLASSPATH variable references the location of your JDBC library, as described in "Configuring the Database Manager" on page 46. Also, make sure the path to the multiserver executable is correct.

Important DiscRack uses the database and corresponding application data layer described in Chapter 4, "Tutorial: Building Enhydra Applications." Before you can run the application, you must load the database schema, as described in "Loading the Schema" on page 54. Alternatively, you can load the Microsoft Access database in *<DiscRack_root>*/discRack/data/discRack.mdb.

To run DiscRack, enter the following commands:

```
cd <DiscRack_root>/discRack/output
./start
```

To access the application, enter the URL `http://Localhost:5555` in your browser location field. Your browser displays the following screen:



Play around with the application to get a sense for how it works. Click on the “Sign Up!” button to add yourself as user, then add some discs to your inventory. Try viewing your inventory and editing one of the discs.

Process and Preliminaries

Before discussing the workings of the DiscRack application, it is useful to understand how you go about developing an Enhydra application in general. You can adapt the traditional software development process to Enhydra application development to ensure that:

- The application does what it is supposed to do
- You complete the project in a timely and cost-effective manner
- The application is easy to maintain and upgrade

An in-depth discussion of software methodology is beyond the scope of this book, but it is instructive to understand the basic principles and how they apply to the simple DiscRack application, so that you can reap the benefits when developing a more complex real-world application.

The following process is loosely based on Lutris Technologies' *Structured Delivery Process* (SDP), a rigorous methodology that Lutris developed over the course of many projects. The simplified process described below may be suitable for small projects; for more information on methodology for large, team development projects, see the information on the SDP on the Lutris web site at http://www.lutris.com.

A simplified Enhydra application development process consists of the following steps:

- **Requirements definition**: creating a problem statement of what the application is supposed to accomplish as specifically as possible. This statement essentially defines the high-level goals of the application.
- **Functional specification**: outlining how the application solve the problem stated in the requirements definition.
- **Design and storyboard**: designing the presentation, data, and business layers of the application, and then creating the storyboard.
- **Development and testing**: coding and testing the application.
- **Deployment**: packaging and installing the application in its operational environment.

This abbreviated methodology is presented here to illustrate the key aspects of the development process. Complex, real-world applications generally call for a more comprehensive process that includes project milestones, cost analysis, documentation, and so on.

DiscRack Requirements Definition

The Otter family needs a way to track their compact disc collections. Each family member has a CD collection, and they sometimes get mixed up: Otters forget who owns what. They decide that an Enhydra application would be the perfect way to help them manage their CDs. After some discussion, they arrive at a brief requirements statement:

> DiscRack will enable each user to keep track of his or her individual CD inventory; and to add, modify, and delete CDs as needed. The application will keep track of all the pertinent information about each CD, including artist and title.

DiscRack Functional Specification

Briefly, DiscRack will meet its requirements as follows:

- Maintain a list of users and passwords; users must log in with a user name and password to access their CD inventory.
- Allow new users to sign up by entering their name, a user name, and a password.
- Once logged in, a user can see his CD inventory and:
  - Add new CDs to the inventory

    - Edit existing CD entries
    - Delete an existing entry, with confirmation prompt
- The information that will be displayed for each CD includes artist, title, genre, and whether the user likes the CD.

Design and Storyboard

The bulk of this step consists of the engineering design for the application, including the design of database schema and corresponding data layer, business logic, and presentation logic. The user interface design can be largely encapsulated by a *storyboard.*

A storyboard is a visual way of describing a user's navigation paths through the application. It provides an outline of the application's user interface, and a framework from which the rest of the application design can proceed. A conceptual storyboard that is largely an application flow chart is sometimes referred to as a *site map,* in contrast to a mocked-up HTML storyboard. This book will refer to both as a storyboard.

The storyboard for DiscRack is shown in Figure 5.1.

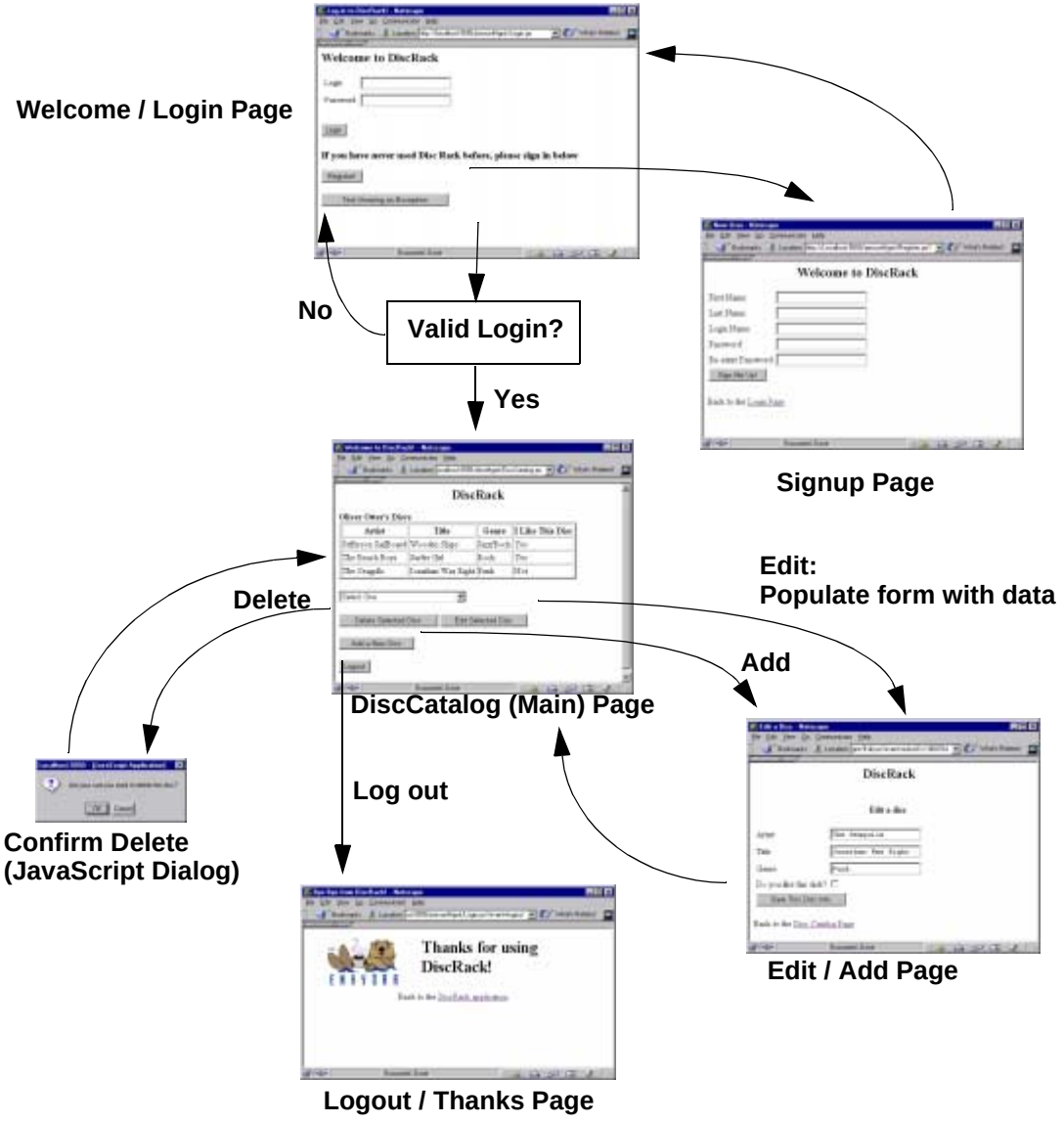


Figure 5.1 DiscRack Storyboard

You can see from the storyboard that there are five HTML pages in the application. You can also see that the DiscCatalog page that shows the CD inventory is the central page in the application. The first page the user sees will always be the Login page; the last page will always be the Logout page.

DiscRack includes a working storyboard (or application "mockup") in the resources directory. It is a set of static HTML pages that illustrate how the application works. To see the storyboard, load this file in your browser:

```
<DiscRack_root>/discrack/resources/personMgmt/Login.html
```

This displays DiscRack login page.

Click on the "Login" button to "log in" and see the disc catalog; click on the "Sign Up" button to display the Signup page. Click around on the links, and you can see the rest of the storyboard. The flow of the HTML pages follows Figure 5.1. Of course, none of the back-end logic is activated—all the HTML is static. But the storyboard gives you a good feel for how the application works.

Development, Testing, and Deployment

The remaining steps are development, testing, and finally deployment. Rather than go through the exercise of describing these steps for DiscRack, the rest of this chapter describes the DiscRack application itself.

When you build an application from the top level, the **make** files create an `output` directory containing the configuration files and the start script. Also, there will be a `lib` directory with a JAR file that contains all the application's class files, along with any other files (for example, GIFs or style sheets).

To deploy the application, you just need to copy these files to the server on which you want the application to run, and make the appropriate changes to the configuration files to reflect the new location. Of course, Enhydra must be installed on this server, and you need to have any ancillary libraries (such as your database's JDBC driver) available.

Overview of DiscRack

The DiscRack application consists of 23 classes in nine packages:

Layer/package	Class or package name	Description
discRack package	**DiscRack**	The application object
	DiscRackException	A simple base exception class
presentation layer / package	**BasePO**	An abstract base class for all presentation objects
	DiscRackSessionData	A container for session data
	ErrorHandler	A class to handle exceptions not caught elsewhere in the application
	DiscRackPresentationException	A presentation layer exception class
	presentation.personMgmt package	A package that contains classes for managing presentation related to the PERSON table: **Register** and `Login`

Layer/package	Class or package name	Description
	presentation.discMgmt package	A package that contains classes for managing presentation related to the DISC table: **Edit** and **DiscCatalog**
Business layer / package	**DiskRackBusinessException**	A business layer exception class
	business.person package	A package that contains two classes: **Person**, which represents a person **PersonFactory**, which has a single method that returns the **Person** object for a user name
	business.disk package	A package that contains two classes: **Disc**, which represents a disc. **DiscFactory**, which has methods to return a **Disc** object for an ID or for the owner's name.
Data layer		Described in "Loading the Schema" on page 54.

The six HTML files are in the resources directory. These correspond to the five HTML pages shown in the storyboard, plus an error page that appears when an error occurs that is not handled by an exception.

The Presentation Layer

The presentation layer includes all of the HTML, Java, and JavaScript that defines the user interface of the application.

Presentation Base Class

All of the Presentation objects in DiscRack are derived from a common base class, **BasePO**, which is an implementation of the Enhydra interface **HttpPresentation**. This interface has one method, **run( )**, which takes the HTTP request as a parameter.

A Presentation base class enables the application to group common functionality in one place. Notice that **BasePO** is an abstract class, so it cannot be instantiated itself, only sub-classed. Also, some of its methods are declared abstract, so subclasses must implement them.

BasePO has methods to handle some of DiscRack's key tasks:

- User log in and session maintenance
- Event handling, and calling the HTML generation methods in the subclass Presentation objects

It is important to realize that you are not required to use a base Presentation class. An alternative is to use the Enhydra Application object to perform common tasks.

The central method in **BasePO** is **run( )**, which makes method calls to perform session maintenance and event handling:

```
public void run(HttpPresentationComms comms) throws Exception {
  // Initialize new or get the existing session data
  initSessionData(comms);
  // Check if the user needs to be logged in for this request.
  if(this.loggedInUserRequired()) {
    checkForUserLogin();
  }
  // Handle the incoming event request
  handleEvent(comms);
}
```

Every time a client browser requests a presentation object URL, the application calls this method. Its logic is very simple:

- Initialize or get the existing session data by calling **initSessionData( )**.
- If this PO requires a log in—as determined by **loggedInUserRequired( )**, an abstract method implemented by each PO—then call **checkForUserLogin( )** to determine if the user has already logged in; if not, then redirect the browser to the login page.
- Call **handleEvent** to handle the current event and determine what HTML to generate.

Each of these methods is explained in the following sections.

The **run** method has a parameter, **comms**, that is an object containing information about the HTTP request. Its member properties include: **application**, **exception**, **request**, **response**, **session** and **sessionData**. These six properties provide all of the information for the request. For example, you can retrieve session data with **getComms( ).sessionData.get( )** and query string parameters with **getComms( ).request.getParameter( )**.

Session Data and Log In

The basics of Enhydra session maintenance were introduced in “Maintaining Session State” on page 35. In contrast to the way session information was handled in that previous example, DiscRack stores all its session information in a single object, **DiscRackSessionData,** and saves that object in the user’s session.

DiscRackSessionData is a simple container class that has two properties: a **Person** object that represents the user and a string called **userMessage** for error messages such as “Please choose a valid disc to edit.” **DiscRackSessionData** has member properties for these data, and methods to get and set them.

There are several advantages of keeping session data in one object:

- It centralizes control of session information; this is especially helpful when multiple presentation objects access the same session data.

- It is type-safe. Since **Session.getSessionData( )** returns a generic **Object**, if you store session data separately, you have to cast each item to the appropriate type, which can lead to runtime errors that are hard to debug.
- It facilitates session data maintenance. If there is a large amount of session data, you can periodically clean up the unneeded data. For example, say you wanted to store an array of hundreds of discs in the user's session to speed access, but you didn't necessarily want leave it there until they log out. With a session data object, you could easily implement a method to clean up unneeded data in the session.

The initSessionData( ) Method

The first thing each PO does is to call **initSessionData( )**. The main portion of this method is shown here:

```
Object obj = getComms().sessionData.get(DiscRackSessionData.SESSION_KEY);
if(null != obj) {
  this.mySessionData = (DiscRackSessionData)obj;
} else {
  this.mySessionData = new DiscRackSessionData();
  getComms().sessionData.set(DiscRackSessionData.SESSION_KEY,
this.mySessionData);
}
```

The first statement in this code snippet gets the session data object, using the session key "DiscRackSessionData." If the session data object exists, it gets type cast to **DiscRackSessionData**; otherwise, the code creates a new **DiscRackSessionData** object and saves it to the user's session with **set( )**.

The loggedInUserRequired( ) Method

BasePO has an abstract method called **loggedInUserRequired( )** that returns a boolean value. Thus, every PO is required to implement this method, which indicates whether a user is required to be logged in to access the associated page. In **BasePO.run( )**, if this method returns true, then **checkForUserLogin( )** is called.

The checkForUserLogin( ) Method

The **checkForUserLogin( )** method determines if a user has a valid login. If not, then it redirects the browser to the Login page:

```
...
Person user = getUser();
if (null == user) {
...
  throw new ClientPageRedirectException(LOGIN_PAGE);
}
...
```

Several statements that write debug messages to a log channel have been removed from this code for clarity. The call to **getUser( )** is really just a call to **getSessionData( ).getUser( )**, which retrieves the **Person** object saved in the current session. If the user has not logged in, or the session has timed out, then this method

will return `null`, and the code will throw a **ClientPageRedirectException** with the URL to the Login page as the argument to the constructor.

When a client browser is redirected by a **ClientPageRedirectException**, any parameters from a query string that were available to the original presentation object are lost. So if you want to pass an error message, you must put the information in the user's session or directly into the query string of the redirected URL.

Event Handling

In this context, an "event" refers to the task a user is performing. While you could create a separate PO for each task in an application, in many cases it makes sense to have a single PO handle multiple events. For example, the **Edit** PO responds to four events: showing the add page, showing the edit page, actually adding a disc to the database, and deleting a disc from the database. The **Login** PO handles three events: show page, login, and logout.

Setting the Event Parameter

DiscRack keeps track of the event it is processing with the "event" parameter, which is sent in the query string of a request. For example, the URL

```
http://Localhost:8000/discMgmt/Edit.po?event=showAddPage
```

specifies the event "showAddPage."

DiscRack illustrates several techniques for setting the event:

- The "showAddPage" event is defined in the `DiscCatalog.html` page by the JavaScript onClick event handler of the "Add a New Disc" button. This calls the JavaScript function **showAddPage()**, which explicitly adds the event to the URL requested: `document.location='Edit.po?event=showAddPage'`. This function is defined in `presentation/discMgmt/DiscCatalogScript.html`, not the DiscCatalog page, as explained in "Replacing JavaScript" on page 71.
- The "add" event (to add a disc to the database) is defined in the `Edit.html` page by a hidden form field: `<input type="hidden" name="event" value="add" id="EventValue">`. When the user clicks the Add button, "event=add" is added to the form submission request along with the other form data the user entered.
- The "exit" event is defined in the DiscCatalog.html page by the second form's ACTION attribute, `"../personMgmt/Exit.html"`. At compile time, this URL is replaced by `'../personMgmt/Login.po?event=logout'`, as explained in "URL Mapping" on page 70.

Although DiscRack does not demonstrate it, you can also set the event when you throw a **PageRedirectException**. You use this exception to transfer control from one PO to another. To specify an event, add the string "?event=someEvent" to the URL string passed to the constructor of **PageRedirectException**.

The handleEvent( ) Method

Once the event is set, the **handleEvent( )** method of **BasePO** performs the actual event handling:

```
String event = getComms().request.getParameter(EVENT);
String returnHTML = null;

if (event == null || event.length() == 0) {
  returnHTML = handleDefault();
} else {
  returnHTML = getPageContentForEvent(event);
}
getComms().response.writeHTML(returnHTML);
```

This method gets the "event" parameter from the request query string and calls the appropriate event handler. If it does not find "event" in the request query string, it calls **handleDefault( )** which is an abstract method, and so must be implemented by all **BasePO** subclasses. Otherwise, it calls **getPageContentForEvent( )** which returns the string content for the specific event and PO. This method contains the following three lines:

```
Method method = this.getClass().getMethod(toMethodName(event),
null);
String thePage = (String)method.invoke(this, null);
return thePage;
```

This code uses reflection (defined in the **java.lang.reflect** package) to call the method in the PO corresponding to the current event. Reflection allows you to call a method whose name is defined at runtime.

The call to **toMethodName( )** returns a string "handleXxx" where "xxx" is the current event (for example "handleShowAddPage" for "showAddPage"). The call to **method.invoke( )** then calls this method.

Reflection allows **BasePO** to call methods in its subclasses without knowing in advance the names of the methods. This scheme works as long as the presentation object code follows the appropriate naming conventions: for every event "foo," there must be a method **handleFoo( )** in the PO class that needs to handle that event.

HTML Pages

The HTML pages for DiscRack are in the *<discRack_root>*/discRack/resources directory. Keeping the HTML pages there rather than in the presentation directory cleanly separates the HTML files from the Java files. Although this is superfluous for small applications, it is a key advantage for large applications with a graphic design team and a programming team.

The **make** files in the presentation layer control how the application uses the HTML files. There are a total of three **make** files in the presentation layer, one in the top level, and one in each sub-directory. To keep the HTML files in a directory separate from the presentation classes, the **make** files use the HTML_DIR directive that specifies the relative path to the directory containing the HTML files. For example, in presentation/Makefile, you'll see:

```
HTML_DIR = ../resources
```

And in `presentation/discMgmt/Makefile`:

```
HTML_DIR = ../../resources/discMgmt
```

The **make** rules will also find any HTML files in the presentation directories (for example `discMgmt/DiscCatalogScript.html`).

The `HTML_CLASSES` directive indicates the names of the class files that XMLC creates, as explained in "Adding a New Page to the Application" on page 37.

Notice there is a `presentation/media` directory that contains only a **make** file. This directory mirrors the final package structure for the JAR file. A line in the **make** file copies the GIF into the finished jar file:

```
JAR_INSTALL = \
        ../../resources/media/*.gif
```

Maintaining the Storyboard

The storyboard is initially just a mockup of the application. But with a few simple steps, you can maintain the working storyboard throughout the entire development process. This capability becomes particularly important for large applications created by a team of programmers and graphic designers: each team can work on their part of the application separately from the other.

After the graphic designers complete their work, you can then replace the old, "mock up" user interface with the new improved interface, that may include improved graphics, JavaScript special effects, style sheets, and so on. An example of doing this is illustrated in "Replacing the User Interface" on page 72.

In addition to keeping the HTML files separate from the Java code, as described in the previous section, there are three steps you have to follow to maintain the storyboard during development:

1. Define rules to map URLs like `Login.html` to `Login.po`
2. Remove dummy data from the HTML files
3. Replace JavaScript, if necessary

Each of these steps is described in detail in the following sections.

URL Mapping

In the working storyboard, as in any static HTML pages, hyperlinks reference other HTML pages. That is, the URLs in hyperlinks end in `.html`. However, in the working application, links to dynamic pages reference presentation object URLs that end in `.po`.

So, you need to do something to convert the "normal" URLs in the storyboard to `.po` URLs. You do this by using the XMLC **-urlmapping** option to map URLs from one form to another. You use this option like this:

```
-urlmapping oldURL newURL
```

To use this option in the make process, you must create an XMLC options file, and then identify the file in the **make** file with the `XMLC_HTML_OPTS_FILE` directive. For example:

```
XMLC_HTML_OPTS_FILE = options.xmlc
```

The `presentation/discMgmt/options.xmlc` file contains the lines:

```
-urlmapping 'Edit.html' 'Edit.po'
-urlmapping 'DiscCatalog.html' 'DiscCatalog.po'
-urlmapping '../personMgmt/Exit.html' '../personMgmt/Login.po?event=logout'
```

When XMLC compiles the files in this directory, it will replace occurrences of the first string (for example, "Edit.html") with the second string (for example, "Edit.po") in hyperlink URLs and FORM ACTION attributes.

Removing Dummy Data

HTML files often contain "dummy" data to make the storyboard pages look more representative of their actual runtime appearance. You need to remove this dummy data from the production application.

Look in `presentation/discMgmt/options.xmlc` again; in particular, look at the last line:

```
-delete-class discardMe
```

The **-delete-class** option tells XMLC to remove any tags (and their contents) whose `CLASS` attribute is "discardMe." For example, if you look in `resources/discMgmt/DiscCatalog.html`, you see this HTML:

```
<tr class="discardMe">
  <td>Sonny and Cher</td>
  <td>Greatest Hits</td>
  <td>Boring Music</td>
  <td>Not</td>
</tr>
```

It's not that we don't love Sonny and Cher: the `CLASS` attribute in the table row definition marks the row for deletion. Unlike `ID`, the value of a `CLASS` attribute does not have to be unique in the page. You can remove all of the dummy in the application with the same "discardMe" value.

Replacing JavaScript

In addition to replacing URLs, you often need to replace JavaScript in the storyboard with JavaScript to be used in the "real" application. For example, `resources/DiscCatalog.html` contains the following script:

```
<SCRIPT id="DummyScript">
<!--
function doDelete()
{
  document.EditForm.action='DiscCatalog.html';
  if(confirm('Are your sure you want to delete this disc?')) {
    document.EditForm.submit();
  }
}
```

```
function showAddPage()
{
  document.location='Edit.html';
}
//-->
</SCRIPT>
```

These functions help to keep the storyboard working. At runtime, though, the application needs to use the "real" functions, which are defined in `presentation/DiscCatalogScript.html`, for example:

```
...
function showAddPage()
{
  document.location='Edit.po?event=showAddPage';
}
...
```

Because XMLC views JavaScript as a comment, the URL mapping option will not work on this URL inside the JavaScript function. So, you have to replace it at runtime with the following code in `DiscCatalog.java`:

```
DiscCatalogHTML page = new DiscCatalogHTML();
HTMLScriptElement script = new DiscCatalogScriptHTML().getElementRealScript();
XMLCUtil.replaceNode(script, page.getElementDummyScript());
```

This is an example of replacing a node with a node from another document. This implementation uses the **XMLCUtil** class.

Because this action happens at runtime, it may have a slight effect on performance. If performance is critical, you may wish to replace the JavaScript in the final deployed version of the application.

Maintaining the storyboard seems like additional unnecessary work, but it is worth the work when your HTML is evolving in parallel with the Java code. As an example of the power of a working storyboard, you can exchange the HTML in Disk Rack from the basic HTML to designed HTML.

Replacing the User Interface

Once the graphic design is completed, you can replace the user interface of the application with its final version. DiscRack includes a `resources_finished` directory containing "finished" versions of the HTML pages, along with a graphic and a stylesheet.

To replace the original storyboard resources with the "finished" resources:

- Rename the `resources` directory to `resources_old`
- Rename the `resources_finished` directory to `resources`
- Edit *<DiscRack_root>*`/discRack/presentation/media/Makefile` and in the JAR_INSTALL directive, remove the two comment symbols (`#`), and add a continuation character (`\`) after the first line; so that it looks like this:

```
JAR_INSTALL = \
../../resources/media/*.gif \
```

```
../../resources/media/*.css \
../../resources/media/*.jpg
```

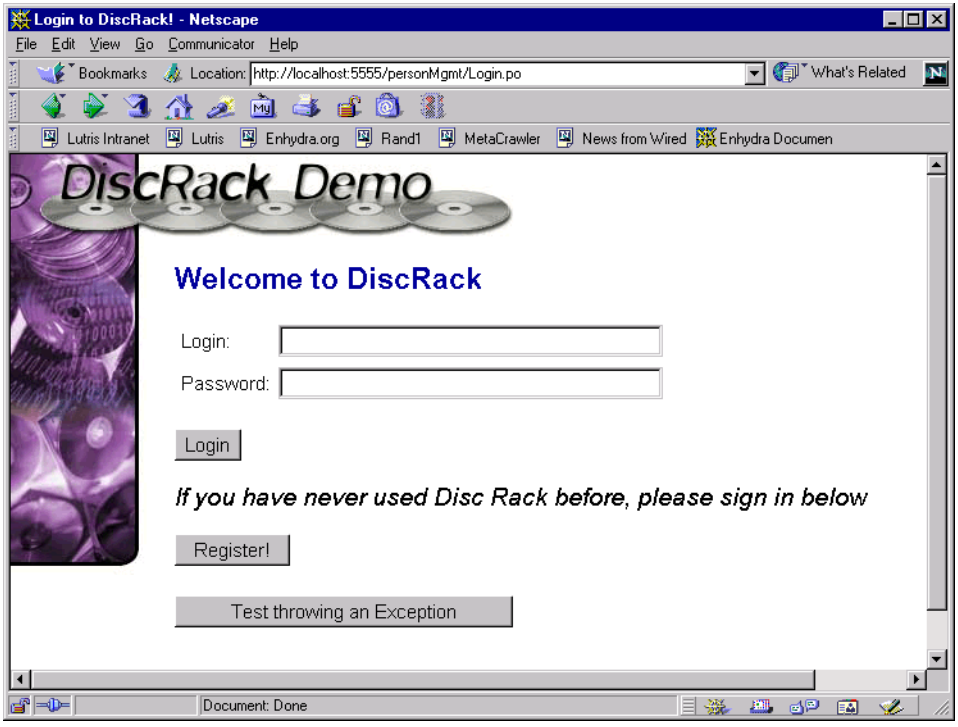
This ensures that the new JPEG graphics files and the style sheet file are included in the packaged application JAR file.

- Rebuild the presentation package, by entering the following commands from the directory *<DiscRack_root>*/discRack/presentation:

```
make clean
make
```

The **make clean** command will remove all the old classes, so that **make** will completely rebuild the application from scratch.

Now, restart and access the application. You see the new and improved user interface:



Populating a List Box

The DiscCatalog page illustrates how to populate a SELECT list box, which is a common task. First, look at the HTML for the SELECT tag in DiscCatalog.html:

```
<SELECT id="TitleList" Name="discID">
<OPTION selected VALUE="invalidID">Select One</OPTION>
<OPTION id="templateOption">Van Halen: Van Halen One</OPTION>
<OPTION class="discardMe">Sonny and Cher: Greatest Hits</OPTION>
```

```
  <OPTION class="discardMe">Sublime: 40 oz. to Freedom</OPTION>
  </SELECT>
```

Now look in DiscCatalog.java for the code that populates the list box.

```
  HTMLOptionElement templateOption = page.getElementTemplateOption();
  Node discSelect = templateOption.getParentNode();
```

The first line above retrieves the DOM object corresponding to the template OPTION tag. The second line calls **getParentNode( )** to get the container SELECT tag. Since the SELECT tag has an ID attribute, this line could have also been:

```
  Node discSelect = page.getElementTitleList();
```

Then, following some code for populating the table, there is one line to remove the template row.

```
  templateOption.removeChild(templateOption.getFirstChild());
```

The other OPTION tags have CLASS="discardMe," which causes XMLC to remove them at build time, as explained in "Removing Dummy Data" on page 71.

Then, within the **for** loop that iterates over the discs belonging to the current user, the following lines actually populate the list box:

```
HTMLOptionElement clonedOption = (HTMLOptionElement)
templateOption.cloneNode(true);
clonedOption.setValue( currentDisc.getHandle() );
Node optionTextNode =
  clonedOption.getOwnerDocument().createTextNode(currentDisc.getArtist() + ":
" +
  currentDisc.getTitle());
clonedOption.appendChild(optionTextNode);
discSelect.appendChild(clonedOption);
```

The first line copies (clones) the template option element into a DOM object of type **HTMLOptionElement**. The second line sets the VALUE attribute to the value returned by **getHandle( )**, which is the disc's OBJECTID, an unique identifier. The third (very long) line creates a text node consisting of "*artistName*: *titleName*." Finally, the last two lines append the text node to the option node, and then append the option node to the select node.

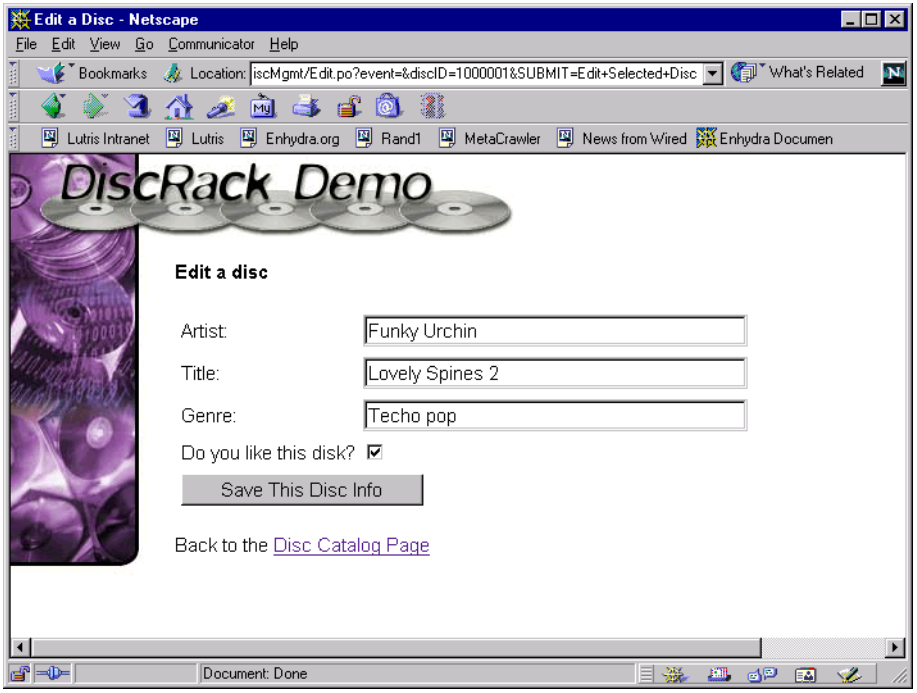
The resulting runtime HTML will look something like this:

```
  <SELECT name='discID' id='TitleList'>
  <OPTION value='invalidID' selected>Select One</OPTION>
  <OPTION value='1000001'>Funky Urchin: Lovely Spines</OPTION>
  <OPTION value='1000021'>The Seagulls: Screaming Fun</OPTION>
  </SELECT>
```

Although this example might seem obscure, it is fairly short, and you can extend its basic functionality to handle more complex situations. For example, you modify it to set the default selection based on a second query.

Populating a Form

When a user chooses a disc from the list box and clicks on the "Edit Disc" button, a form appears that is populated with the existing values for that disc. The user can then edit the values and submit them back to the database.



Here is the HTML for the form elements in `Edit.html`. The `TABLE` tags have been omitted for clarity:

```
<INPUT TYPE="hidden" NAME="discID" VALUE="invalidID" ID="DiscID">
Artist: <input name="artist" id="Artist" >
Title: <input name="title" id="Title" >
Genre: <input name="genre" id="Genre" >
Do you like this disk?
<input TYPE="checkbox" name="like" CHECKED ID="LikeBox">
<INPUT TYPE="submit" VALUE="Save This Disc Info">
```

In `Edit.java`, the event-handling method **handleDefault( )** calls **showEditPage( )** with a null parameter to populate the form with the selected disc's values. Ordinarily, the only request parameter (other than the event type) is the disc ID, accessed by this statement:

```
String discID = this.getComms().request.getParameter(DISC_ID);
```

These statements also access the other request parameters, but ordinarily they are null (but see the error-handling case discussed later):

```
String title = this.getComms().request.getParameter(TITLE_NAME);
String artist = this.getComms().request.getParameter(ARTIST_NAME);
String genre = this.getComms().request.getParameter(GENRE_NAME);
```

Then, a call to **findDiscByID( )** retrieves a **Disc** data object that has that ID:

```
disc = DiscFactory.findDiscByID(discID);
```

Then, there is a series of if statements that check the values of **title**, **artist**, **genre**, and **isLiked**, which are normally null. Therefore, the following statements are executed (the surrounding **if** statements are not shown for brevity):

```
page.getElementDiscID().setValue(disc.getHandle());
page.getElementTitle().setValue(disc.getTitle());
page.getElementArtist().setValue(disc.getArtist());
page.getElementGenre().setValue(disc.getGenre());
page.getElementLikeBox().setChecked(disc.isLiked());
```

These statements use XMLC calls to set the `VALUE` attributes of the form elements; the values are retrieved from the **disc** object.

When the user finishes editing, and clicks "Save this Disc Info," **handleEdit( )** processes the changes. This method calls **saveDisc( )**, which attempts to save the new values—if successful, it redirects the client to the DiscCatalog page; if any of the new values are null, though, **saveDisc( )** throws an exception. The **catch** clause then calls **showEditPage( )** with an error string and with request parameters.

Note that **ClientPageRedirectException** is a subclass of **java.lang.Error**, so it is not caught by the **catch** clause when it is thrown.

```
try {
  saveDisc(disc);
  throw new ClientPageRedirectException(DISC_CATALOG_PAGE);
} catch(Exception ex) {
return showEditPage("You must fill out all fields to edit this disc");
}
```

The result is that when a user tries to edit a disc and delete some of the values, the edit page will re-display, maintaining all the non-null form element values, and restoring the previous values to the null-valued form elements. The page will also display the error string.

The Business Layer

The DiscRack business layer is simple, consisting primarily of two packages, Disc and Person, and two corresponding factory classes **DiscFactory** and **PersonFactory**. A *factory* is an object whose primary role is to create other objects.

The Business Objects

The business objects **Disc** and **Person** are largely wrappers for the corresponding data layer classes, **DiscDO** and **PersonDO**, with get and set methods for each property in the data objects (or column in the database tables). For example, **Disc** has **getArtist( )** and **setArtist( )** methods.

The objects in the business layer perform all the interfacing with the data layer. So, if the data layer needs to change, nothing in the presentation layer is affected; conversely, if the presentation layer changes, nothing in the data layer is affected.

DiscFactory has two static methods:

- **findDiscsForPerson( )**, that returns an array of **Disc** objects that belong to the **Person** object specified as the method's argument.
- **findDiscByID( )**, that returns the single **Disc** object that has the ID specified in the method's argument.

PersonFactory has one static method, **findPerson( )**, that returns a **Person** object that has the user name specified in the method's argument. If the method finds more than one person in the database, then it writes an error message to the log channel and throws an exception.

Using Data Objects

To help understand ow DiscRack uses DODS data layer code, look at the **findPerson( )** method in **PersonFactory**. The comments have been removed from this code for brevity.

```
  public static Person findPerson(String username)
  throws DiscRackBusinessException
  {
    try {
      PersonQuery query = new PersonQuery();
      query.setQueryLogin(username);
      query.requireUniqueInstance();
      PersonDO[] foundPerson = query.getDOArray();
      if(foundPerson.length != 0) {
        return new Person(foundPerson[0]);
      } else {
        return null;
      }
    } catch(NonUniqueQueryException ex) {
  ...
```

First, this method instantiates a new **PersonQuery** object. **PersonQuery** is a data layer object used to construct and execute a query on the person table. It has a number of **setQueryXXX( )** methods for qualifying the query parameters (that is, setting the values to be matched in the `WHERE` clause of the `SELECT` statement). For example, the above code calls **setQueryLogin( )**, with **username** as a parameter, to set the value to be matched in the LOGIN column.

Next, the method calls **requireUniqueInstance( )**, which indicates that the query is to return a single row, and will throw an exception otherwise. Then, it calls **getDOArray( )**, which executes the query, returning an array of **PersonDO** objects. Finally, the method returns a single **Person** object returned by the query; if the query did not return any rows, it returns `null`.

APPENDIX

A

Database Configuration

This appendix provides information on connecting Enhydra applications to specific database types. In general, you need to add the database configuration information to the application configuration file, *appName*.conf.

Driver Configuration

Enhydra connects to databases using a JDBC driver. Enhydra has its own class loader, but the JDBC driver must be loaded by the system class loader. Therefore, it is important to specify the path to JDBC driver in your system CLASSPATH and not in the Enhydra application's CLASSPATH.

A common way to do this is to save the driver in a lib directory in the project and define the CLASSPATH in the start script. To do this, follow these steps:

1. Create a lib directory in the top level of your project and copy your JDBC driver to this directory.
2. Edit your application's start file (in the appName/appName directory) to place the driver in your CLASSPATH. For example, the bold set CLASSPATH:

```
#!/bin/sh

echo "***"
echo "***  Connect to http://`hostname`:9000/"
echo "***"

CLASSPATH="../lib/classes111.zip"
export CLASSPATH

/usr/local/lutris-enhydra3.0/bin/multiserver ./multiserver.conf
```

3. Build the project with **make**, which will copy the start script to the directory appName/output. Use this script to start your application.

Be careful to keep the right driver with your application. For example, there are multiple versions of the Oracle JDBC driver, `classes111.zip`. When your application goes into production, make sure that the project administrator knows to reference the correct driver when the database is upgraded in the future.

Oracle

This section presents an example of an Oracle configuration. To use this example, change the values shown in bold.

```
#-------------------------------------------------------------------
#                   Database Manager Configuration
#-------------------------------------------------------------------
DatabaseManager.Databases[] = "database_id"
DatabaseManager.DefaultDatabase = "database_id"
DatabaseManager.Debug = "false"
DatabaseManager.DB.database_id.ClassType = "Oracle"
DatabaseManager.DB.database_id.JdbcDriver = "oracle.jdbc.driver.OracleDriver"
DatabaseManager.DB.database_id.Connection.Url =
"jdbc:oracle:thin:@server_name:1521:db_instance"
DatabaseManager.DB.database_id.Connection.User = "User"
DatabaseManager.DB.database_id.Connection.Password = "Password"
DatabaseManager.DB.database_id.Connection.MaxPreparedStatements = 10
DatabaseManager.DB.database_id.Connection.MaxPoolSize = 30
DatabaseManager.DB.database_id.Connection.AllocationTimeout = 10000
DatabaseManager.DB.database_id.Connection.Logging = false
DatabaseManager.DB.database_id.ObjectId.CacheSize = 20
DatabaseManager.DB.database_id.ObjectId.MinValue = 1
```

The driver used here is the Oracle thin driver, and “db_instance” is the name of the Oracle database instance.

Informix

This section presents an example of an Informix configuration. To use this example, change the values shown in bold.

```
#-------------------------------------------------------------------
#                   Database Manager Configuration
#-------------------------------------------------------------------
DatabaseManager.Databases[] = "database_id"
DatabaseManager.DefaultDatabase = "database_id"
DatabaseManager.Debug = "false"
DatabaseManager.DB.database_id.ClassType = "Informix"
DatabaseManager.DB.database_id.JdbcDriver = "com.informix.jdbc.IfxDriver"
DatabaseManager.DB.database_id.ConnectionURL =
jdbc:informix-sqli://<hostname>:<port#>:INFORMIXSERVER=<db_inst>;user=<user>;
password=<password>
DatabaseManager.DB.database_id.Connection.user = "<user>"
DatabaseManager.DB.database_id.Connection.Password = "<password>"
DatabaseManager.DB.database_id.Connection.MaxPreparedStatements = 10
DatabaseManager.DB.database_id.Connection.MaxPoolSize = 30
```

```
DatabaseManager.DB.database_id.Connection.AllocationTimeout = 10000
DatabaseManager.DB.database_id.Connection.Logging = false
DatabaseManager.DB.database_id.objectID.CacheSize = 20
DatabaseManager.DB.database_id.MinValue = 1
```

Sybase

This section presents an example of a Sybase configuration. To use this example, change the values shown in bold.

```
#---------------------------------------------------------------------
#                   Database Manager Configuration
#---------------------------------------------------------------------
DatabaseManager.Databases[] = "database_id"
DatabaseManager.DefaultDatabase = "database_id"
DatabaseManager.Debug = "true"
DatabaseManager.DB.database_id.ClassType = "Sybase"
DatabaseManager.DB.database_id.JdbcDriver = "com.sybase.jdbc2.jdbc.SybDriver"
DatabaseManager.DB.database_id.Connection.Url =
"jdbc:sybase:Tds:<hostname>.sybase.com:7100"
DatabaseManager.DB.database_id.Connection.User = "name"
DatabaseManager.DB.database_id.Connection.Password = "password"
DatabaseManager.DB.database_id.Connection.MaxPoolSize = "2"
DatabaseManager.DB.database_id.Connection.AllocationTimeout = "2"
DatabaseManager.DB.database_id.Connection.Logging = "true"
DatabaseManager.DB.database_id.Connection.MaxPreparedStatements = "2"
DatabaseManager.DB.database_id.ObjectId.CacheSize = 2
DatabaseManager.DB.database_id.ObjectId.MinValue = 1
```

MySQL

MySQL is an Open Source database that is lightweight and fast. However, it does not support transactions. Because of this, you have to make a small patch to the Enhydra code to use MySQL. For more information, see the Enhydra mailing list archive.

Patch

Because MySQL does not support transactions, and therefore does not support autocommit, you have to make a small change to the code and rebuild Enhydra. You need to change the file `com/lutris/appserver/server/sql/standard/StandardDBConnection.java` and comment out one line, as shown below:

```
  public void setAutoCommit(boolean on) throws SQLException {
    validate();
    logDebug("ignores set auto commit: " + on);
    // connection.setAutoCommit(on);
  }
```

You must then rebuild this Enhydra package. For details, see the Enhydra mailing list archive.

Configuration

This section presents an example of a MySQL configuration. To use this example, change the values shown in bold.

```
#-----------------------------------------------------------------
#                   Database Manager Configuration
#-----------------------------------------------------------------
DatabaseManager.Databases[] = database_id
DatabaseManager.DefaultDatabase = database_id
DatabaseManager.Debug = true
DatabaseManager.DB.database_id.ClassType = Standard
DatabaseManager.DB.database_id.Connection.User = username
DatabaseManager.DB.database_id.Connection.Password = password
DatabaseManager.DB.database_id.Connection.MaxPoolSize = 5
DatabaseManager.DB.database_id.Connection.AllocationTimeout = 10000
DatabaseManager.DB.database_id.Connection.Logging = true
DatabaseManager.DB.database_id.ObjectId.CacheSize = 1024
DatabaseManager.DB.database_id.ObjectId.MinValue = 100
DatabaseManager.DB.database_id.JdbcDriver = org.gjt.mm.mysql.Driver
DatabaseManager.DB.database_id.Connection.Url ="jdbc:mysql://[host]:[port]/
[inst]"
```

PostgreSQL

PostgreSQL is a popular open source database used with Enhydra. However, as explained in “Loading the Schema” on page 54, DODS requires a special column named OID in each table—however OID is a reserved word in PostgreSQL. Fortunately, the column names used for OID and VERSION are configurable.

To configure these names, add the following lines to you application configuration file:

```
DatabaseManager.ObjectIdColumnName = "ColName_for_ObjectID"
DatabaseManager.VersionColumnName = "ColName_for_Version"
```

where *ColName_for_ObjectID* and *ColName_for_Version* are the column names you want to use instead of OID and VERSION.

Microsoft Access

Microsoft Access is not a true SQL database server; as such, it is suitable for development and testing, but not for a production database. Access does not have a JDBC driver. However, Access does support ODBC, and there is a JDBC-ODBC bridge in the Sun JDK, which enables Access to work with Enhydra.

Since Access cannot read in files containing SQL commands, you must create tables in the Access GUI. See the Access documentation for more information. For the DiscRack example, you can also use the Access database provided in `<enhydra_root>/examples/DiscRack/discRack.mdb`.

You can test the ODBC access alone using the test program in , “Establishing a JDBC Connection” on page 43. Use the driver and connect strings from the configuration file listed here. If you encounter problems, be sure your data values are valid.

To use Enhydra with Access:

1 Register the database as an ODBC data source:

- Go to Start | Settings | Control Panel and click on ODBC Data Sources.
- Select the add button in the window that comes up.
- Select the Microsoft Access Driver in the Create New Datasource window and click Finish.
- The ODBC Microsoft Access Setup window appears. Choose a name, like discRack, for the Data Source Name. Under Database, click the Select button, browse to the `*.mdb` file, select it, and click the OK button.

2 Place database information in the application’s configuration file, as shown in the example below.

Note You don’t have to place the JDBC driver in the application’s `CLASSPATH`, because the ODBC/JDBC bridge is in the JDK and thus is already in the system’s `CLASSPATH`.

This section presents an example of an Access configuration. To use this example, change the values shown in bold.

```
#-----------------------------------------------------------------
-
#                   Database Manager Configuration
#-----------------------------------------------------------------
-
DatabaseManager.Databases[] = "database_id"
DatabaseManager.DefaultDatabase = "database_id"
DatabaseManager.Debug = "false"
DatabaseManager.DB.database_id.ClassType = "Standard"
DatabaseManager.DB.database_id.JdbcDriver =
"sun.jdbc.odbc.JdbcOdbcDriver"
DatabaseManager.DB.database_id.Connection.Url =
"jdbc:odbc:discRack"
DatabaseManager.DB.database_id.Connection.User = "Admin"
DatabaseManager.DB.database_id.Connection.Password = ""
DatabaseManager.DB.database_id.Connection.MaxPreparedStatements =
10
DatabaseManager.DB.database_id.Connection.MaxPoolSize = 30
DatabaseManager.DB.database_id.Connection.AllocationTimeout = 10000
DatabaseManager.DB.database_id.Connection.Logging = false
DatabaseManager.DB.database_id.ObjectId.CacheSize = 20
DatabaseManager.DB.database_id.ObjectId.MinValue = 1
```

APPENDIX

B

Using the Multiserver Administration Console

This appendix introduces the Enhydra Multiserver Administration Console, which you can use to start, stop, add, delete, and modify your applications and servlets.

Launching the Administration Console

To launch the Multiserver Administration Console (the *Console*), follow these steps:

1. Type the following command to start the multiserver:
   ```
   ./bin/multiserver
   ```
   If the Console does not start, then your path is not correctly set. See "Setting the PATH Environment Variable" on page 22 for more information.
2. In your browser, display the console by entering the following URL:
   ```
   http://localhost:8001/
   ```

3 The Console displays a password entry dialog box, as shown in Figure B.1. To get started, enter the default username (`admin`) and password (`enhydra`).

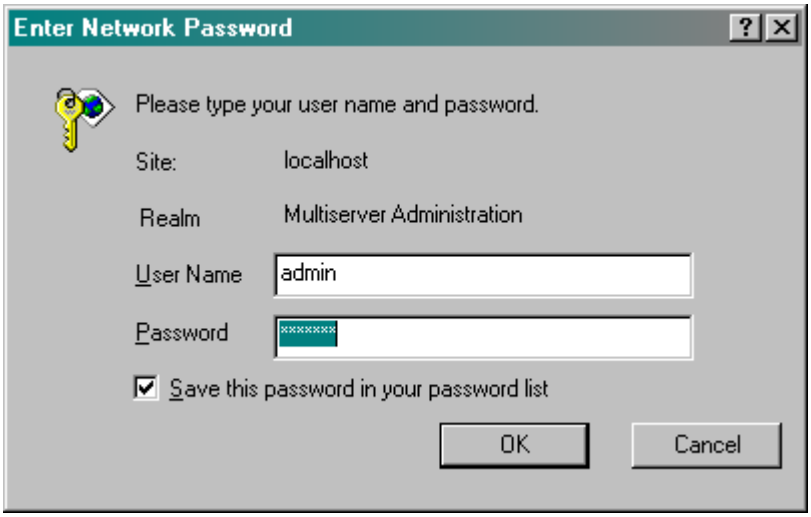


Figure B.1 The Multiserver Administration Console password dialog box

The Enhydra Console will display in your browser, as described in the next section.

The Console Display

Figure B.1 shows the initial view of the Console in a browser window. The Console has two frames:

- The frame on the left is the Control Frame, which contains the buttons that you use to operate the console tools.
- The frame on the right is the Content Frame. This frame is initially empty. It displays the results of actions that you perform.

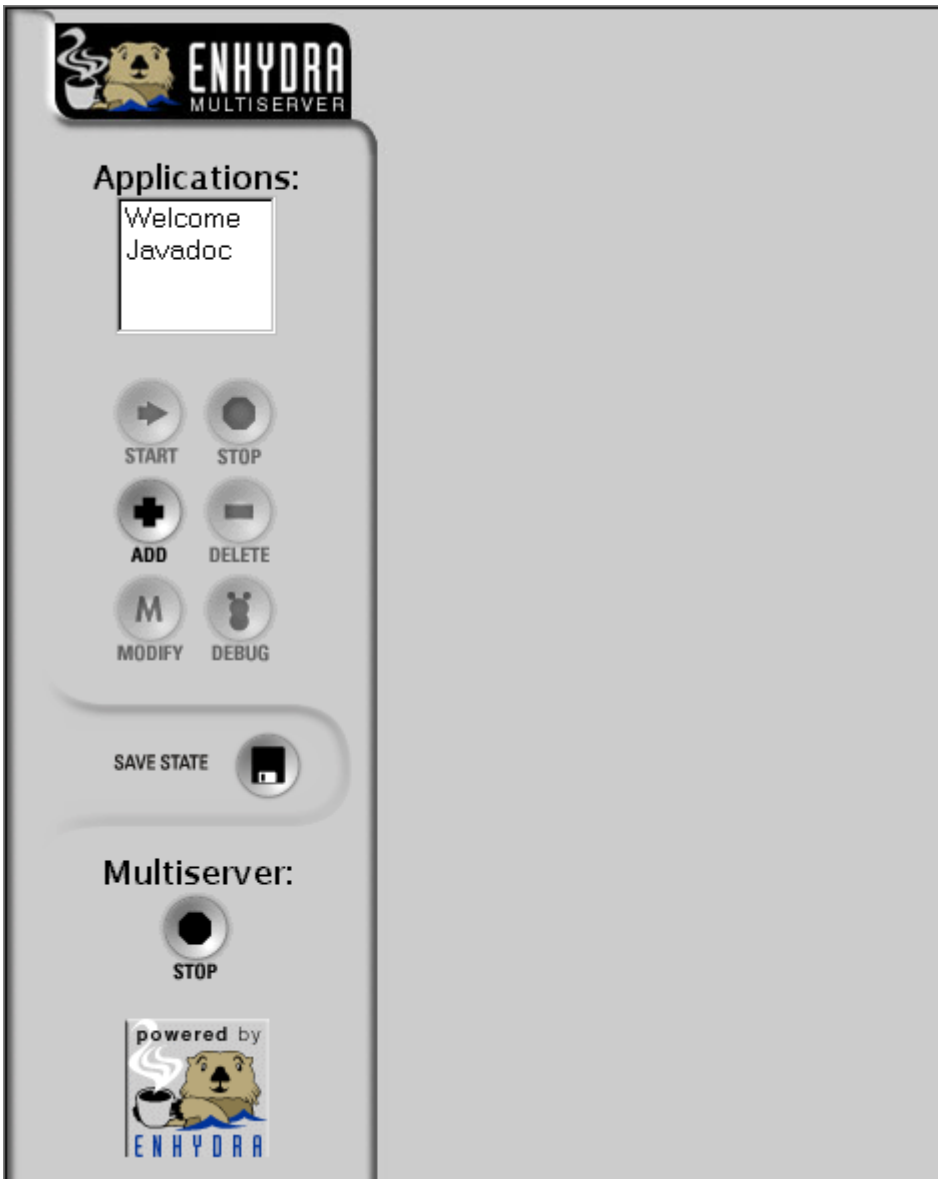


Figure B.2 The Multiserver Administration Console display

The Control Frame has two components:

- The Applications window
- The Console Tool buttons

The Applications Window

The Applications window contains a list of all of the applications and servlets available in the Enhydra Multiserver. You will see two applications that are initially available:

- The Welcome application
- The Javadoc application

Use the **ADD** button to add new applications or servlets to the Multiserver, and the **DELETE** button to delete applications or servlets from the Multiserver.

Note To permanently add or delete an application or servlet, you must save the console's state by clicking the **SAVE STATE** button.

The Console Tool Buttons

Table B.1 describes the function of each console tool button in the Console.

Table B.1 The Console Tool buttons

Button	Description
	Starts the servlet or application that is currently selected in the Applications window. Note that this button is unavailable when the selected application or servlet is already running.
	Stops the servlet or application that is currently selected in the Applications window. If the application has active users, you are prompted to verify that you want the application stopped. Note that this button is unavailable when the selected application or servlet is not already running.
	Adds a servlet or application to the Enhydra Multiserver. For more information, see "Adding an Application or Servlet" on page 91.
	Removes the servlet or application that is currently selected in the Applications window from the Enhydra Multiserver. For more information, see "Deleting an Application or Servlet" on page 93.

Table B.1 The Console Tool buttons

Button	Description
	Modifies the configurable attributes of the servlet or application that is currently selected in the Applications window. For more information, see “Modifying the Configuration of An Application or Servlet” on page 93.
	Invokes the debugging utility for the servlet or application that is currently selected in the Applications window. When you click this button, the debugging control panel displays, as described in “Debugging an Application or Servlet” on page 94.
	Saves the state of the Enhydra Multiserver. For more information, see “Saving the State of The Multiserver” on page 95.
	Stops the Enhydra Multiserver.

The Content Frame

This section describes the Content Frame of the Console, which shows you information about the application of servlet that is currently selected in the Applications window. Some of the Console tools also use the Content Frame to display information or to ask you for input.

You can select the information you want displayed in the Content Frame by clicking on one of its two tabs:

- Click on the Application or Servlet tab to display status information for the application or servlet. Figure B.3 shows an example of the status display for an application. Note that the status display for a servlet is slightly different.
- Click on the Connections tab to display connection status information for the application or servlet. Figure B.4 shows an example of the connection status display for an application.

Figure B.3 shows a portion of the status display for the Welcome application.

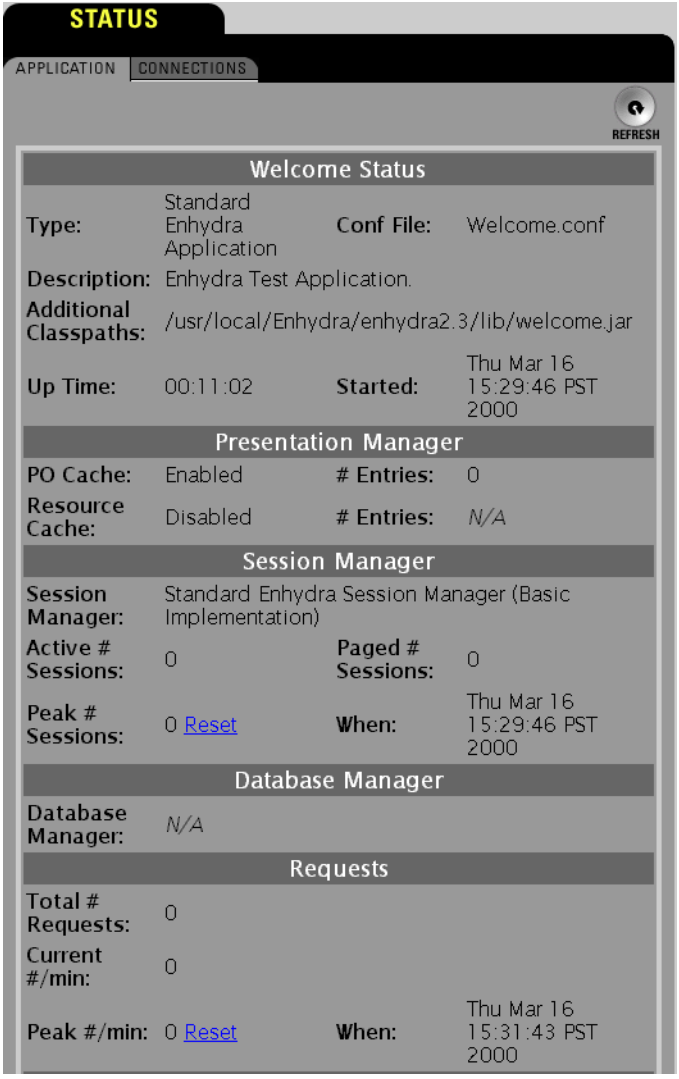


Figure B.3 The status display for a running application

Figure B.4 shows the connection status display for the Welcome application.

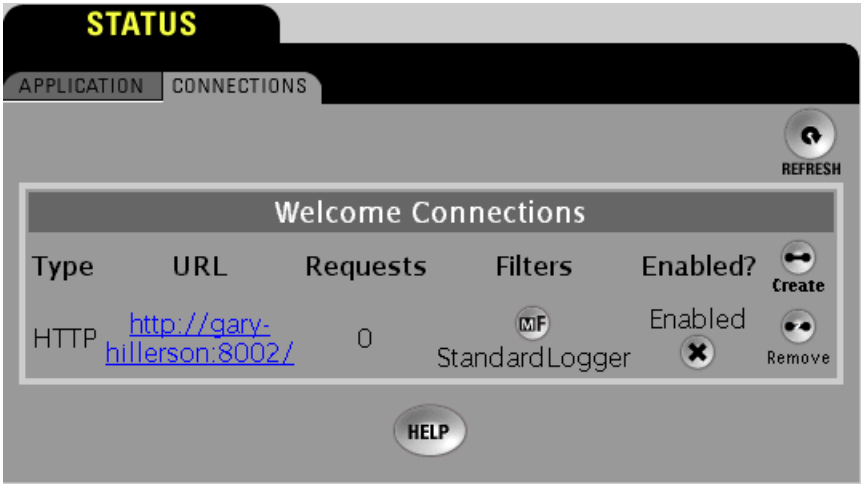


Figure B.4 The connection status display for a running application

Using the Console Tools

This section describes how to use the console tools to work with your Enhydra applications and servlets. This section tells you how to:

- Add a new servlet or application to the Enhydra Multiserver
- Delete an application or servlet
- Modify the configuration of an application or servlet
- Use the debugging tool
- Save the state of the Multiserver

Adding an Application or Servlet

To add an application or servlet to those that can be managed from the administration console, the application's configuration (`.conf`) file must be in the Enhydra applications directory. The default location for this directory is `<enhydra_root>/apps`.

Use the following steps to add an application to the Multiserver:

1 Click the **ADD** button. The Multiserver Administration Console displays the Add New Application / Servlet dialog box, which is shown in Figure B.5.

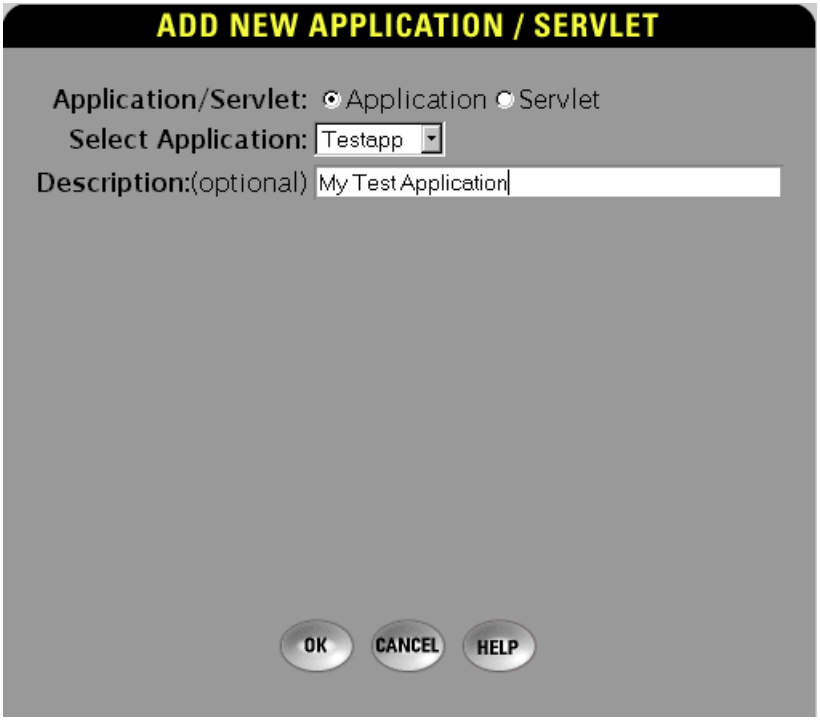


Figure B.5 The Add Application dialog box

2 Select the name of your application from the dropdown list. Your application's name only appears in the list if its configuration file is in the `/<enhydra_root>/apps` directory.

3 Click the **OK** button to add your application to the Multiserver.

If you are adding a servlet instead of an application, you need to enter additional information in the dialog box, including:

- The name of the servlet
- The name of the class to instantiate for the servlet
- Any additional class paths required for the servlet
- The root of the servlet's file system on disk
- Optionally, any initial arguments for the servlet

Deleting an Application or Servlet

Note When you remove an application or servlet from the Multiserver, you are not deleting the application or servlet, or its configuration file from your computer. You are simply removing it from the Multiserver's configuration file.

Use the following steps to delete an application or servlet from the Multiserver:

1. If the application is running, you must first stop it.
2. Select the application in the Applications window.
3. Click the **DELETE** button.

Modifying the Configuration of An Application or Servlet

Use the following steps to modify the configuration of an application in the Multiserver:

1. Select the application in the Applications window.
2. Click the **MODIFY** button. The Content Frame displays the Modify window, as shown in Figure B.6.

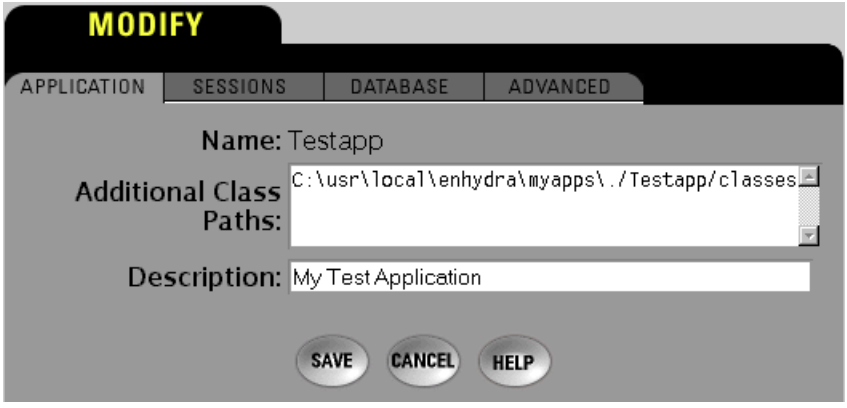


Figure B.6 The Modify Configuration window

The Modify Configuration window features tabs that you can use to modify the application or servlet. If you are modifying an application, you can choose from among four tabs. Use the:

- Application tab to add additional class paths for the application or servlet
- Sessions tab to modify the application's Session Manager parameters
- Database tab to modify the applications database connection.
- Advanced tab to modify the application's default URL.

If you are modifying a servlet, there is only one tab to choose. You can use the Servlet tab to modify the Servlet's configuration options, which are the same options that you specify when adding the Servlet.

Debugging an Application or Servlet

The debugging tool provides you with a window into the operation of a running application or servlet; it traces the flow of requests to and responses from an application or servlet.

Use the following steps to debug an application or servlet in the MultiServer:

1 Select the application in the Applications window.

2 Click the DEBUG button. The Debug popup window displays, as shown in Figure B.7.

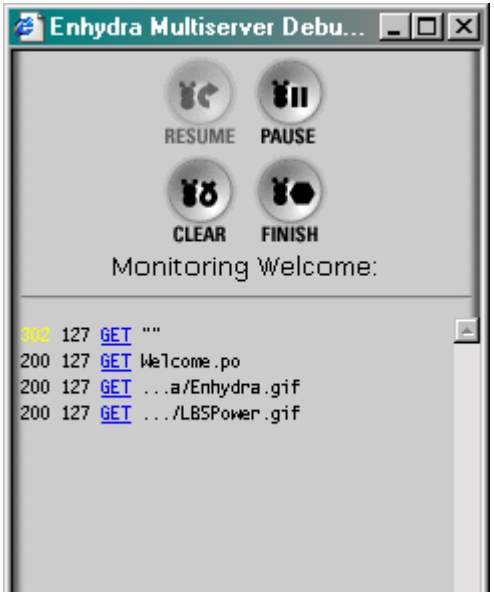


Figure B.7 The Debug popup window

The scrolling area in the window shows the active event list. You can use the debugging buttons as follows:

- Click the **PAUSE** button to pause the debugging function, which stops the accumulating of events in the scroll list.
- Click the **RESUME** button to resume the debugging function.
- Click the **CLEAR** button to clear the list of events.
- Click the **FINISH** button to halt debugging and close the popup window.

Saving the State of The Multiserver

When you add or delete applications or servlets with the Console, you are changing the current configuration of the Multiserver. If you want the changes to be retained, you must save the configuration.

Use the following steps to write the current configuration to the Multiserver configuration file:

1. Click the **SAVE STATE** button.
2. Click the **OK** button in the confirmation dialog box.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

U

V

W

X