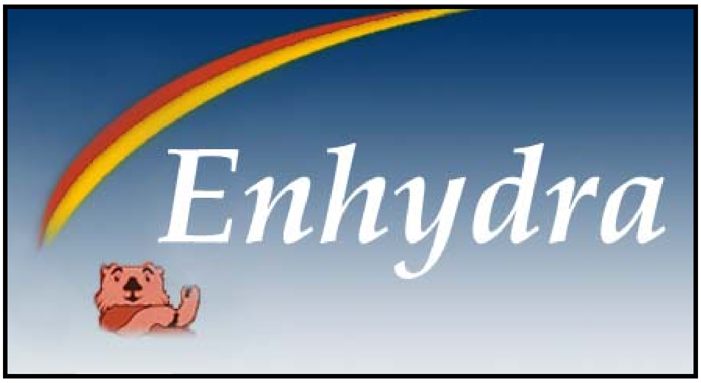
Getting Started

with

Enhydra



Contents

Chapter 1

Introduction

This book introduces the Enhydra™ ver. 5.1 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 to illustrate some of the key principles of Enhydra applications.

What you should already know

This book assumes you have the following basic skills:

- General understanding of the Internet, the World Wide Web (Web), and Hypertext Markup Language (HTML).
- Good working knowledge of the Java programming language. Some knowledge of Java servlets is also helpful.
- Good understanding of relational databases; knowledge of SQL is helpful.

Conventions used in this book

The typographical conventions used in this book are listed in Table 1.1.

Table 1.1 Typographical conventions

Convention	Description
Italics	Indicates variables, new terms and concepts, and book titles. For example, • A *servlet* is a Java class that dynamically extends the functionality of aWeb server.
Fixed-width	Used to indicate several types of items. These include: • Commands that you enter directly, code examples, utility programs, and options. For example, • cd mydir • System.out.println("Hello World"); • make utility • -keep option • Java packages, classes, methods, objects, and other identifiers. For example, • ErrorHandler class • run() method • Session object **Note:** Method names are suffixed with empty parentheses, even if the method takes parameters. **Note:** Only specific references to object names are in fixed-width; generic references to objects are shown in plain text. • File and directory names. For example: • /usr/local/bin **Note:** UNIX path names are used throughout and are indicated with a forward slash (/). If you are using the Windows platform, substitute backslashes (\) for the forward slashes (/).

Table 1.1 Typographical conventions (continued**)**

Convention	Description
Fixed-width italic and <*Fixed-width italic*>	Indicates variables in commands and code. For example, • xmlc [*options*|*optfile*.xmlc ...] *docfile* **Note:** Angle brackets (< >) are used to indicate variables in directory paths and command options. For example, • -class <*class*>
Boldface	Used for the words **Note**, **Tip**, **Important**, and **Warning** when they are used as headings that draw your eye to essential or useful information.
Keycaps	Used to indicate keys on the keyboard that you press to implement an action. If you must press two or more keys simultaneously, keycaps are joined with a hyphen. For example, • Ctrl-C.
{ } (braces)	Indicates a set of required choices in a syntax line. For example, • {a|b|c} means you must choose a, b, or c.
[ ] (brackets)	Indicates optional items in a syntax line. For example, • [a|b|c] means you can choose a, b, c, or nothing.
...(horizontal ellipses)	Used to indicate that portions of a code example have been omitted to simplify the discussion, and to indicate that an argument can be repeated several times in a command line. For example, • xmlc [*options*|*optfile*.xmlc ...] *docfile*
plain text	Used for URLs and generic references to objects. For example, • http://www.lutris.com/documentation/index.html • The presentation object is in the presentation layer
ALL CAPS	Indicates SQL statements. For example: • CREATE statement

Table 1.2 Additional conventions

Convention	Description
Enhydra root directory	When you install Enhydra, you install the Enhydra executables and libraries in a directory of your choice. This directory is referred to as the Enhydra root directory or <*enhydra_root*>.
Paths	UNIX path names are used throughout and are indicated with a forward slash (/). If you are using the Windows platform, substitute backslashes (\) for the forward slashes (/). For example, • /usr/local/bin
URLs	URLs are indicated in plain text and are generally fully qualified. For example, • http://www.lutris.com/documentation/index.html
Screen shots	Most screen shots reflect the Microsoft Windows look and feel.

Getting Started

Getting Started with Enhydra introduces the fundamentals of Enhydra. The purpose of this book is to introduce Enhydra and provide a groundwork for understanding and working with Enhydra and its associated tools. It includes a detailed tutorial and an explanation of the Enhydra DiscRack sample application.

Enhydra 5.1 information available on enhydra.org

You can find a variety of information about open-source Enhydra at the Enhydra website:

http://www.enhydra.org .

The Enhydra site is the home of the Enhydra 5.1 open-source community, one of Enhydra's greatest assets. The Enhydra community consists of numerous entities, including community sponsors, technology providers, users, and of course, developers.

Chapter 2

Installation

Complete step-by-step installation instructions for Enhydra and related software (including bundled third-party software) are provided in Enhydra documentation, directory ***<enhydra_root>/doc***. To begin, refer to the top-level ***index.html*** file of this directory.

For convenience, we recommend that you print the file containing the Enhydra installation instructions prior to installation. (The instructions are included in HTML and PDF format). However, you can also follow the step-by-step installation instructions online (you can toggle back and forth between the installation program and browser).

Chapter 3

Overview

This chapter provides a high-level overview of Enhydra, Enhydra applications, and the tools used to create Enhydra applications. The following topics are covered:

- What is Enhydra 5.1?
- What's new in Enhydra 5.1
- Anatomy of an Enhydra application
- Multiserver runtime component
- Enhydra application framework
- Enhydra tools
- Kelp tools

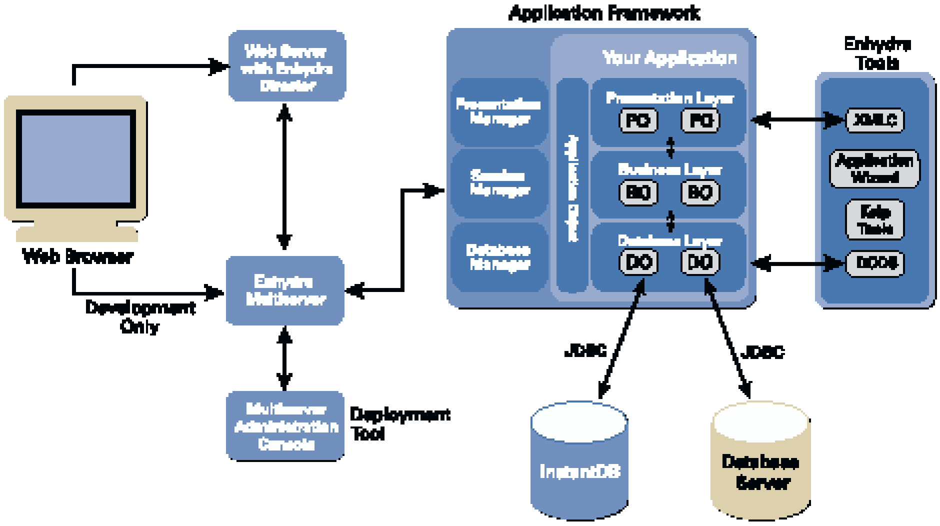
What is Enhydra 5.1?

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.

An *Enhydra application* is a Java program that runs in Multiserver and uses the Enhydra application framework at runtime.

Figure 3.1 Enhydra application model



As illustrated in Figure 3.1, Enhydra has three parts:

- **Multiserver:** Runs Enhydra applications either by itself or with a Web server.
- **Application framework:** Collection of Java classes, which provide the runtime infrastructure for Enhydra applications.
- **Enhydra tools:** Use to develop Enhydra applications.

The following sections describe Enhydra and Enhydra applications in more detail.

For more information on	See this topic
Enhydra application architecture	“Anatomy of an Enhydra application” in the chapter 3
Multiserver	“Multiserver runtime component” in the chapter 3
Application framework	“Enhydra application framework” in the chapter 3
Enhydra tools	“Enhydra tools” in the chapter 3

Documentation

The documentation for this release is located in ***<enhydra_root>/doc/index.***

Documentation updates

- *Getting Started with Enhydra* has been updated to reflect changes to Enhydra and revised for accuracy.

Anatomy of an Enhydra application

An Enhydra application can be either:

- An *Enhydra super-servlet application* that uses Enhydra’s own application model
- A *servlet application* that uses the J2EE servlet application model

These two kinds of applications are similar in many ways, but have some important differences. Generally, a servlet application has a servlet for each page (HTML, WML, and so on) in the application. In contrast, a super-servlet application consists of a single servlet that contains a *presentation object* for each page.

You can use Enhydra tools such as XMLC and DODS to create both kinds of applications, and you can run both kinds of applications in any standard servlet runner, such as the Enhydra Multiserver.

Enhydra super-servlet applications

An Enhydra application has (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 "Enhydra application framework", later in this chapter.

Application objects

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

- Name of the application
- Status of the application (for example, running/stopped/dead)
- Name and location of the configuration file that initializes the application
- Log channel to use for logging
- References to the application's session manager, database manager, and presentation manager (see"Enhydra application framework", later in this chapter).

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 can make a vector containing the data a property of the application object so all pages can easily access it.

Methods

Each application object has the following methods:

- **startup()** starts the application

You can extend this to perform other startup functions, such as reading settings from the configuration file.

- **requestPreprocessor()** 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; presentation objects perform these tasks. The next section describes presentation objects.

Presentation objects

A *presentation object* generates dynamic content for one or more pages in an Enhydra super-servlet 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 calls the presentation object Xyz.

Note Enhydra only calls a presentation object for URLs with a .po suffix. The Web server generally serves 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. 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 presentation object can use to write data in response to HTTP requests (similar to a servlet's service() method). Presentation objects usually handle GET requests (for example, form submissions) and respond by writing HTML, but they can perform other functions (for example, read files sent by a POST request).

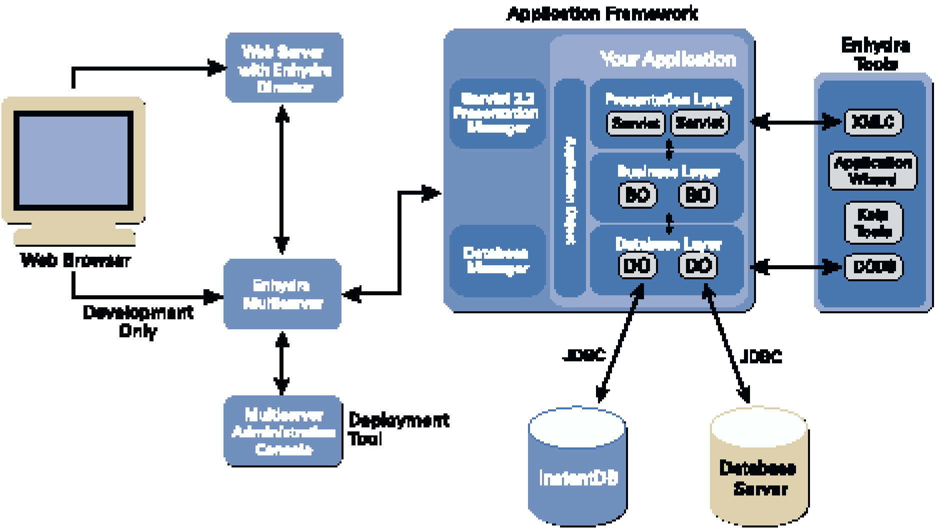
Servlet applications

In addition to super-servlet applications, you can also create and run standard servlet applications (sometimes called Web applications) with Enhydra. Servlet applications conform to the Java servlet API specification, part of the Java 2 Enterprise Edition (J2EE) specification from Sun Microsystems. It is a popular application model for interactive Web applications.

For a detailed information on the servlet application model, see http://java.sun.com/products/servlet/index.html .

In a servlet application, each servlet is responsible for a single page of output (although this is not required, it is common practice). Each servlet must be a subclass of javax.servlet.http.HttpServlet, and will generally override the doGet() method, and possibly other methods, such as init(). The general architecture of a servlet application is illustrated in Figure 3.2.

Figure 3.2 Servlet application model



Although JavaServer Pages (JSPs) are often used to create the presentation layer of servlet applications, XMLC is generally better because it provides a cleaner separation between layout code (such as HTML) and presentation logic code. Fortunately, you can use XMLC to help generate presentation code for servlet applications too.

Servlet versus super-servlet applications

While the servlet application model and the super-servlet application model are similar in many ways, they also have some key differences. You can run both kinds of applications in Enhydra Multiserver, and you can use Enhydra tools such as XMLC and DODS to help create both kinds of applications.

Since the servlet API performs its own presentation management, there is no need for the Enhydra Presentation Manager (see “Presentation Manager”, later in this chapter). Likewise, the servlet API provides session management, so there is no need for the Enhydra Session Manager (see “Session Manager”, later in this chapter).

One of the key differences in the two application models is how the objects in the presentation layer are instantiated. In a super-servlet application, each request creates a new instance of the requested presentation object, and the PO executes in a single-thread. In contrast, in a servlet application there is only one instance of any given servlet, and it is multithreaded (one thread for each request). So, POs can have member variables that are local to each instance, while in a servlet application, any member variable is global to all threads of the instance of the servlet.

Presentation objects have several features that you cannot take advantage of in a servlet application:

- Dynamic page recompilation (so you can change page content while an application is running)
- URL-encoding of session information for cookieless session maintenance
- Automatic setting of MIME-types, for applications that generate multiple document types (for example, HTML and WML)

The Enhydra application framework provides a number of capabilities that are very useful, including:

- Database management
- Logging

Although these are not part of the standard servlet application model, you can save a lot of development time by using them; however, your application will then be dependent on the Enhydra class libraries (contained in jar files).

Application layers

Regardless of the application model you use, you should divide your application into three distinct parts or *layers* for modularity and ease of maintenance:

- The **Presentation layer** handles how the application is presented to Web browsers through HTML. In a super-servlet application, this layer consists of presentation objects (POs); in a servlet application, it consists of servlets.
- The **Business layer** contains business objects. Business objects contain the application’s business logic, including algorithms and specialized functions, but not data access or display functions.
- The **Data layer** handles the interface with the persistent data source, which is typically a relational database.

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 creates data objects to populate the data layer, and creates both Java code and SQL code to create the corresponding tables in the database. For more on DODS, see “Data Object Design Studio (DODS)”, later in this chapter.

Note 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, lets you change the data layer without affecting the presentation layer.

Multiserver runtime component

Multiserver is the runtime component of Enhydra. It provides the services that an Enhydra application uses to communicate with the Web server, and performs all 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 a part of the Java 2 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, and later, 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 Because 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 stopping the server. To update an existing application, you simply restart its class loader.

Enhydra Director

Enhydra Director provides superior scalability for applications 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.

The Enhydra 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 through the Apache Module interface
- Netscape servers through Netscape Application Programming Interface (NSAPI)
- Microsoft servers through the Internet Server Application Programming Interface (ISAPI)
- Other Web servers through the Common Gateway Interface (CGI)

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

Multiserver Administration Console

Enhydra Multiserver provides an Admin Console for managing applications through a Web browser. The *Admin Console* lets you:

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

The Admin Console is described in more detail in “Multiserver Administration Console” in the appendix B.

Enhydra application framework

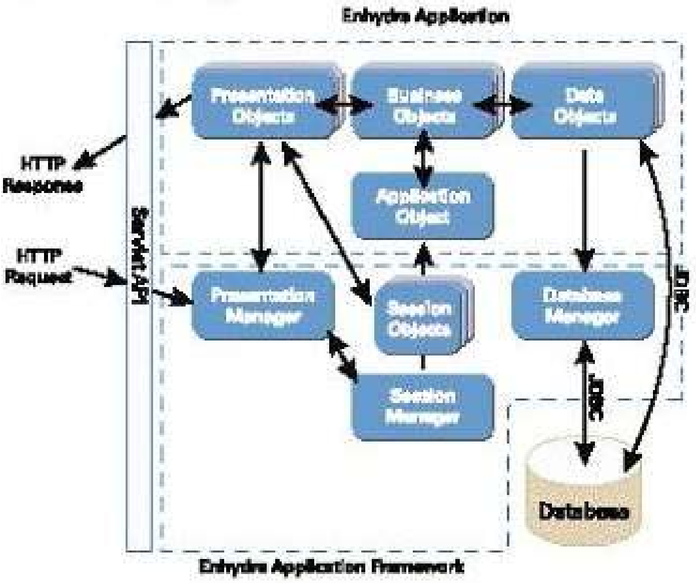
The Enhydra application framework includes:

- Presentation Manager
- Session Manager
- Database Manager

In general, the application framework includes all the classes in the com.lutris.appserver.server.* packages, which 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 3.3, “Enhydra application and Enhydra framework.”

Figure 3.3 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 presentation objects and calls the **run()** method of the presentation object.

Each Enhydra application has one instance of a Presentation Manager. To increase performance, the Presentation Manager caches presentation objects 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 presentation objects 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, 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 a pool of database connections for the application. 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, Informix, Microsoft SQL Server, PostgreSQL, InterBase, and InstantDB.

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:

- Enhydra Application Wizard
- Extensible Markup Language Compiler (XMLC)
- Data Object Design Studio (DODS)
- Kelp tools

Enhydra Application Wizard

The Enhydra Application Wizard (**appwizard**) is a tool with both a command-line and a graphical user interface. The wizard creates a basic framework for an Enhydra application. The wizard lets you create and run a new “stub” application in a matter of minutes, giving your development project a jump-start. For an example of using the Application Wizard GUI, see “Creating your first application” in the chapter 4.

Note The Application Wizard has changed significantly with the release of Enhydra 3.5. Previously, the Application Wizard was a command-line tool, started by entering newapp with a parameter for the project name. The command for starting the Application Wizard and the parameters required to run it as a command-line tool have changed. The basic framework of files and directories generated by the Application Wizard has changed as well.

The Enhydra Application Wizard is also incorporated into the Enhydra Kelp tools, so you can create a basic framework for an Enhydra application from a graphical integrated development environment (IDE).

Extensible Markup Language Compiler (XMLC)

The *Extensible Markup Language Compiler* (XMLC) creates a Java object that mirrors the structure of a eXtensible Markup Language (XML) document. *XML*, 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 let programs 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 lets you separate HTML templates in your application. These templates are typically created by page designers from Java code, which is usually created by programmers. This functionality 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.

This command-line tool generates a Java class file from a 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 using the XMLC, see “Tutorial: Building Enhydra applications” in the chapter 4.

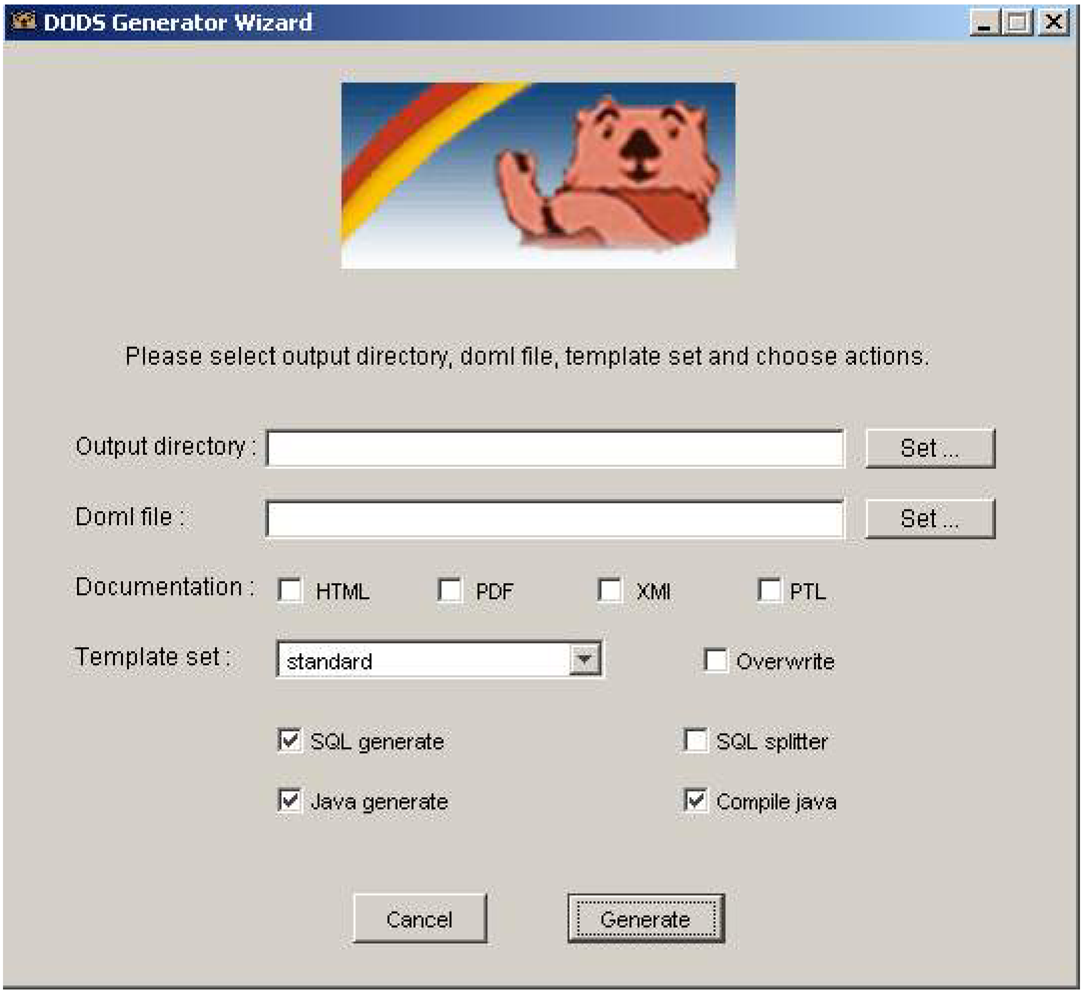
Dynamic recompilation

XMLC dynamic recompilation lets you change HTML layouts at runtime without restarting an application. With this feature, you can make any changes to the static content of HTML pages. The application automatically picks 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.

Data Object Design Studio (DODS)

The Data Object Design Studio (DODS), shown in Figure 3.4, is a tool which, for the given doml file, can generate SQL script files for creating tables (for each table separately and one cumulative file for creating all tables), one file for deleting all tables, and/or java code for data objects described in the given doml file. DODS also has possibility to compile generated java classes and to parse SQL files (to split cumulative SQL into more separated SQL files using SQLSplitter tool).

Figure 3.4 DODS Generator Wizard



Please select output directory, doml file, template set and choose actions.

Output directory :

Doml file :

Documentation : ☐ HTML ☐ PDF ☐ XMI ☐ PTL

Template set : standard ☐ Overwrite

☑ SQL generate ☐ SQL splitter

☑ Java generate ☑ Compile java

Data objects described in the given DOML file correspond to tables in the database. Each data object has *attributes*, which describe database columns, and *reference attributes*, which refer to other data objects. Reference attributes let you create a hierarchy of data objects (for example, many-to-one or many-to-many relationships).

For the given DOML, DODS generates all of the code to implement it. For example:

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

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

- A file named **thing.sql** containing the SQL CREATE TABLE command to construct a table in a relational database.

- Java source file defining a data object representing a row in 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**.

- Java source file that defines a query class, which provides SQL query access to the database table.

  The query class returns a collection of **ThingDO** objects that represent the rows found in the table matching criteria passed from the application.

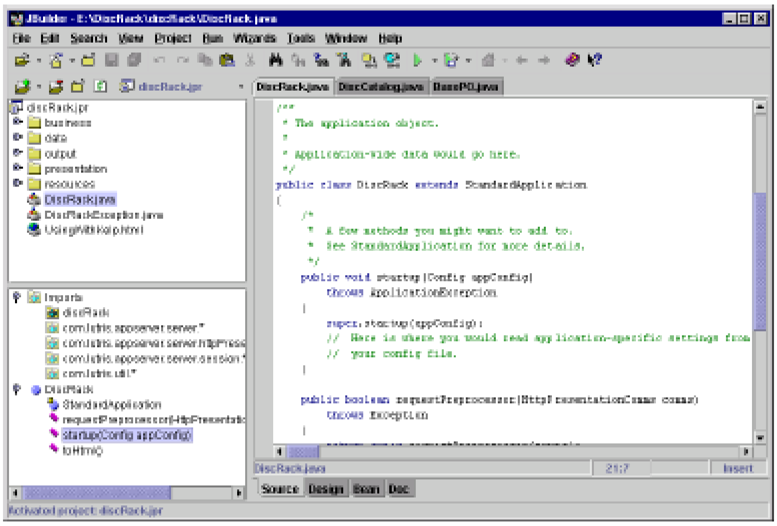
For an example using DODS, see "Using DODS" in the chapter 4.

Kelp tools

The Kelp tools let you develop Enhydra applications in Borland JBuilder, Oracle JDeveloper, NetBeans and Eclipse integrated development environments (IDEs).

Note You can use Kelp in place of Enhydra's shell scripts and **ant** files.

Figure 3.5 JBuilder project generated with Kelp



```
/**
 * The application object.
 *
 * Application-wide data would go here.
 */
public class DiscRack extends StandardApplication
{
    /*
     *  A few methods you might want to add to.
     *  See StandardApplication for more details.
     */
    public void startup(Config appConfig)
        throws ApplicationException
    {
        super.startup(appConfig);
        //  Here is where you would read application-specific settings from
        //  your config file.
    }

    public boolean requestPreprocessor(HttpPresentationComms comms)
        throws Exception
    {
```

Enhydra Application wizard

The Enhydra Application wizard incorporated into Kelp 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" in the chapter 4. 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 let you 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 4

Tutorial: Building Enhydra applications

This chapter describes how to build an Enhydra application from the ground up, and provides important tips on Enhydra application development. In this tutorial, you will:

- Use the Application Wizard to create a starting framework
- Use XMLC to expand the application
- Add simple database access

If you are already familiar with the basics of Enhydra, you may want to skip to the chapter 5, “DiscRack sample application,” for a look at an application with more advanced features.

Creating your first application

The Enhydra Application Wizard (sometimes referred to as appwizard) is a quick way to get up and running with Enhydra. The Application Wizard generates basic Java files and directory structures for new applications. For the tutorial, you will use the Application Wizard GUI.

To create a simple application with the Application Wizard:

1 Create a directory to contain your new application and name it anything you want.
For example:

```
mkdir myapps (UNIX) or

md myapps (WINDOWS)
```

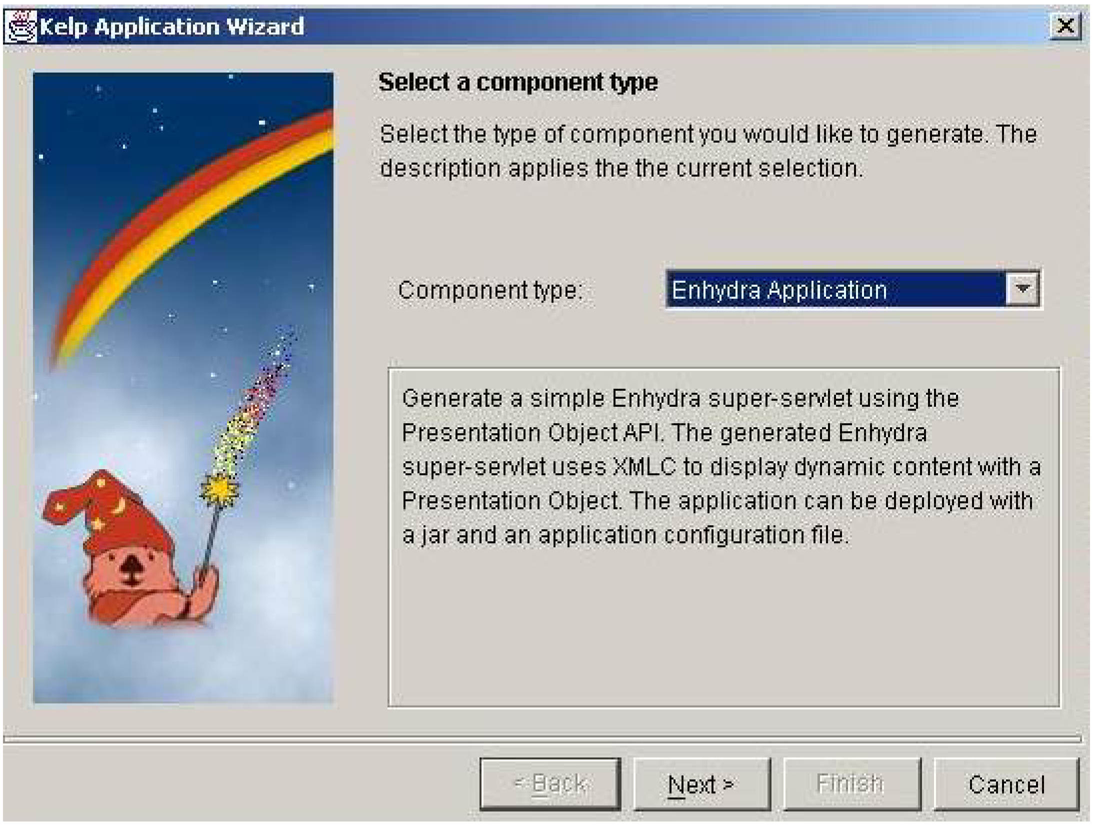
2 Open a shell window (UNIX) or command window (WINDOWS) and make the new directory the current directory.
For example:

```
cd myapps
```

3 Start the Application Wizard GUI by entering **appwizard** at the command prompt. Entering **appwizard** with no arguments brings up the Application Wizard GUI. The Application Wizard can generate two distinct types of Enhydra projects: a Web Application, and an Enhydra super-servlet application. For this tutorial, you will generate a super-servlet application.

Note If the Application Wizard does not start, the path environment variable is not set correctly. The Enhydra installation instructions provide information about setting your path environment variable. The installation instructions are available in HTML and PDF format in Enhydra documentation, ***<enhydra_root>/doc*** directory (refer to the top-level **index.html**).

Figure 4.1 Application Wizard GUI



Select a component type

Select the type of component you would like to generate. The description applies the the current selection.

Component type: Enhydra Application

Generate a simple Enhydra super-servlet using the Presentation Object API. The generated Enhydra super-servlet uses XMLC to display dynamic content with a Presentation Object. The application can be deployed with a jar and an application configuration file.

Note The Application Wizard has changed significantly with the release of Lutris Enhydra 3.5. Previously, the Application Wizard was a command-line tool, started by entering newapp with a parameter for the project name. The command for starting the Application Wizard and the parameters required to run it as a command-line tool have changed. The basic framework of files and directories generated by the Application Wizard has changed as well.

4 Use the Application Wizard GUI to generate a simple Enhydra application. The Application Wizard GUI steps you through the process of generating an Enhydra project.

1 Select a Component type.

Select Enhydra Application from the Component type pull-down menu and click Next.

2 Specify Client type and directory details.

Accept the default client type of HTML. Enter **simpleApp** for the Project directory name. Enter **simpleapp** (note the difference in case) for Package. Set the Root path to /myapps.

3 Specify the copyright material to use.

Click Next to accept the default, No copyright setting.

4 Specify which Supplemental files to generate.

Select Generate start script and command line build files and click Finish.

The Application Wizard creates a new directory called **simpleApp.** This directory is sometimes referred to as the *application root directory*.

5 Make the application root directory the active directory:

```
cd simpleApp
```

6 Browse the application root directory and note the following items created by the Application Wizard:

- file **build.xml**
- file **readme.html** that contains some simple instructions to build and run the application
- file **dods.properties**; if DODS (Data Object Design Studio) is used, you should uncoment the only line that this file contains:

  ```
  dods.doml=<yourDataBase>.doml
  ```

  This line contains information about your database's doml file name. You should change <*yourDataBase*> with the name of your database. For example:

  ```
  dods.doml = discRack.doml
  ```

- A *source directory*, **src**, containing all the source code for the application
- An *input directory***, input**, containing templates of the configuration files and run scripts for the application.

The contents of this application root directory are explained in “Directories and files in SimpleApp”, later in this chapter. In the next section, you will finish building your simple application.

Building the application

To build the application:

1 In the shell window (UNIX) or in the command window (WINDOWS), enter the **ant** command from the application root directory:

```
cd /myapps/simpleApp
ant
```

This creates two subdirectories in the application root directory:

- **classes** directory contains the application’s class files
- **output** directory contains everything needed to run the application

File **build.xml** contains directives that tell **ant** to recursively descend the application directory tree.

When you build the application ant compiles the files located in the simpleApp source directory (simpleApp/src) 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/lib directory, along with the configuration files needed to run the application.

2 To start the application, enter the following commands in the Enhydra shell (UNIX) or in the command window (WINDOWS).

1. **UNIX**

```
cd output
./run
```

2. **WINDOWS**

```
cd output
run
```

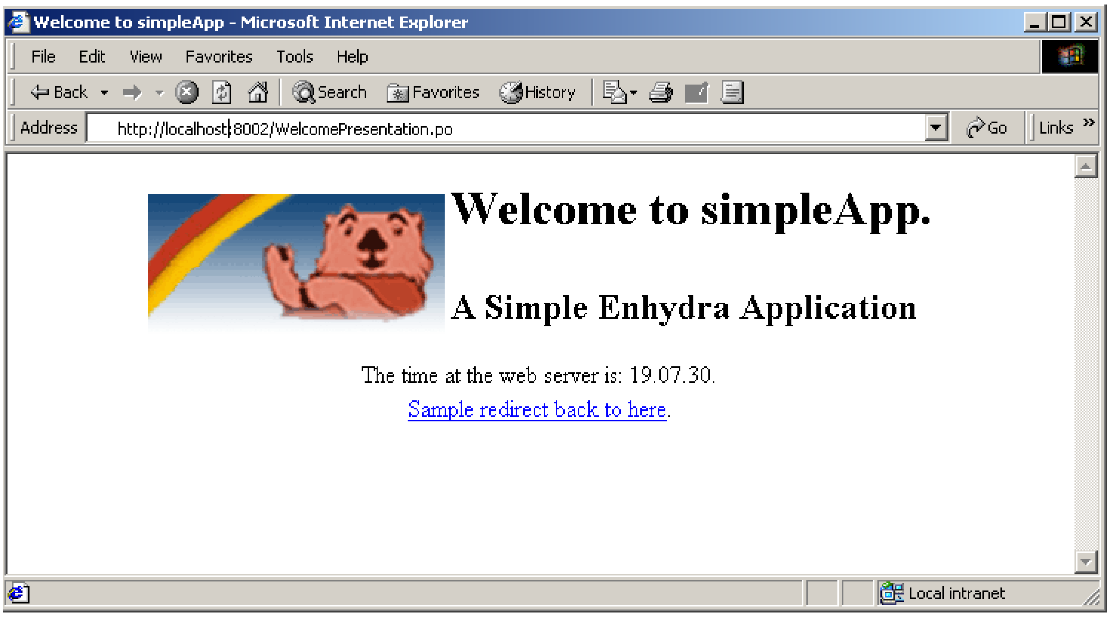
Note The Multiserver Administration Console provides a GUI for managing applications. Among other functions, the Admin Console can be used to start and stop applications. Refer to “Launching the Admin Console”, explained later in this chapter, for additional information and usage instructions.

3 To access the application, enter the following URL in your browser’s location field:

http://localhost:8002

The browser will display the Welcome page for the simpleApp application.

Figure 4.2 Browser with the simpleApp welcome page loaded



Welcome to simpleApp - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost:8002/WelcomePresentation.po
Go Links
Local intranet

Welcome to simpleApp.

A Simple Enhydra Application

The time at the web server is: 19.07.30.

Sample redirect back to here.

You have just built and run your first Enhydra application.

4 Now press Ctrl-C in the shell window (UNIX) or in command window (WINDOWS) 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/src/simpleapp/resources/Welcome.html, which contains a few dozen HTML tags. Notice tags such as these:

```
<center>
   The time at the web server is:
   <span id="time">1/1/00 00:00:00 (static)</span>.
</center>
```

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

Look also at the **WelcomePresentation.java** file in the directory myapps/simpleApp/src/simpleapp/presentation. In particular, notice these lines of code:

```
WelcomeHTML welcome;
String now;
...
welcome = (WelcomeHTML)comms.xmlcFactory.create(WelcomeHTML.class);
now = DateFormat.getTimeInstance(DateFormat.SHORT).format(new Date());
welcome.setTextTime(now);
comms.response.writeDOM(welcome);
```

This code is used for replacing the text inside the <SPAN> tags.

The first couple lines in the code snippet define welcome as an instance of the WelcomeHTML class, and **now** as an instance of the String class. The xmlcFactory in the next line is used to instantiate your HTML page. Next, the variable now is set to the current Date formatted as time. The last line of the snippet sets the time element in the welcome class to the value of now, and returns the value.

When you build the application, the Extensible Markup Language Compiler (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 getElementTime(). The application uses getElementTime() to modify the text content of the <SPAN> tag.

Note In general, XMLC will create a **getElementxxx()** method for each <SPAN> tag with ID attribute value **xxx**. The **xxx** in the method name is replaced by the capitalized spelling of the ID attribute value of the SPAN tag.

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 **write(buffer)** writes the document out to the HTTP response, looking something like this:

```
...
<CENTER>
  The time at the web server is: <SPAN>10:40 AM</SPAN>.
</CENTER>
...
```

For a more detailed explanation of XMLC, see “Using XMLC”, later in this chapter. For more information about the Enhydra Application Wizard, see “Enhydra Application Wizard” in the chapter 3.

Directories and files in SimpleApp

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

- The **src** directory contains the source code for the application. It contains a directory, simpleapp, corresponding to the Package name you assigned in the Applicatin Wizard. The simpleapp directory is divided into three subdirectories for the **business**, **data**, and **presentation** layers and usually there is one more directory: **resources**. The business and data directories are empty in the new application. The resources directory is in connection with the presentation directory and they contain Java, HTML, and media files.

- The **classes** subdirectory contains the application’s compiled Java classes in their package hierarchy and any associated media files, such as .gif files.

- The **output** subdirectory contains the application’s configuration files**, multiserver.conf** and **simpleApp.conf**. It also has a directory **lib** with a **simpleApp.jar** file containing an archived package of everything in the classes directory.

- The **input** directory contains templates of the configuration files and run script. Template files have the extension **.in**. When you build your project from the application root directory, the templates will overwrite the corresponding files in the output directory.

The finished Enhydra application includes the JAR file, simpleApp.jar, and the application configuration file, simpleApp.conf. The ant program also copies the Multiserver configuration file, multiserver.conf, and the run 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. These files include:

- Multiserver configuration file, multiserver.conf, located in the conf subdirectory
- Application configuration file, which is named <AppName>.conf by default (for example, simpleApp.conf)

Note The Admin Console is a GUI for managing applications running on the Multiserver. The Admin Console has its own configuration file, multiserverAdmin.conf. Additionally, files managed by the Admin Console use the default multiserver configuration file, multiserver.conf, located in the **<enhydra_root>** directory.

The **run** script, which is not a configuration file, specifies the CLASSPATH that the application will use.

Important If you do not specify a CLASSPATH in this file, the application will use the system CLASSPATH, which may not be correct.

If you edit the configuration files, you can change the various settings contained in the files, as explained in the text that follows.

Note The Application Wizard creates “input” versions of the configuration files in the *<app_root>/*input directory (for example, simpleApp/input). These are the files that you should edit. Running **ant** creates “runtime” versions of the files in the <app_root>/output directory (for example, simpleApp/output). You should not edit the runtime versions of the files because they will be overwritten each time you build the project.

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

- Name and location of the application configuration file
- Name of the log file and other log file information
- TCP port on which the application will run
- Classname and description of the filter used to log requests made to the server

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

- CLASSPATH that this application will use

  The difference between this CLASSPATH and the one specified in the Multiserver **run 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.

- Class name of the application object

  For example:

```
Server.AppClass = simpleapp.SimpleApp
```

- Prefix used to derive presentation object class names and paths from URLs
  For example:

```
Server.PresentationPrefix = "simpleapp/presentation"
```

- Maximum length (in minutes) of a user session and the session idle time
  For example:

```
SessionManager.Lifetime = 60
SessionManager.MaxIdleTime = 2
```

- Default URL for the application
  For example:

```
Application.DefaultUrl = "WelcomePresentation.po"
```

  When a browser requests the application URL, the Multiserver returns this URL (typically a presentation object) by default.

Launching the Admin Console

You can use the Admin Console to manage your Enhydra applications, as well as Java servlets, and Web archives (WAR files). In this Web-based console, you can add applications to the console, delete them from the console, start or stop applications, modify application settings, and perform some basic application debugging.

In this section of the tutorial, we will:

- launch the Admin Console from the Enhydra shell (UNIX) or from command window (WINDOW),
- add the simpleApp applicatin to the console,
- and then use the Admin Console to start and stop simpleApp.

To launch the Admin Console:

1 Enter the following command in the Enhydra shell (UNIX) or in the command window (WINDOWS) at the prompt:

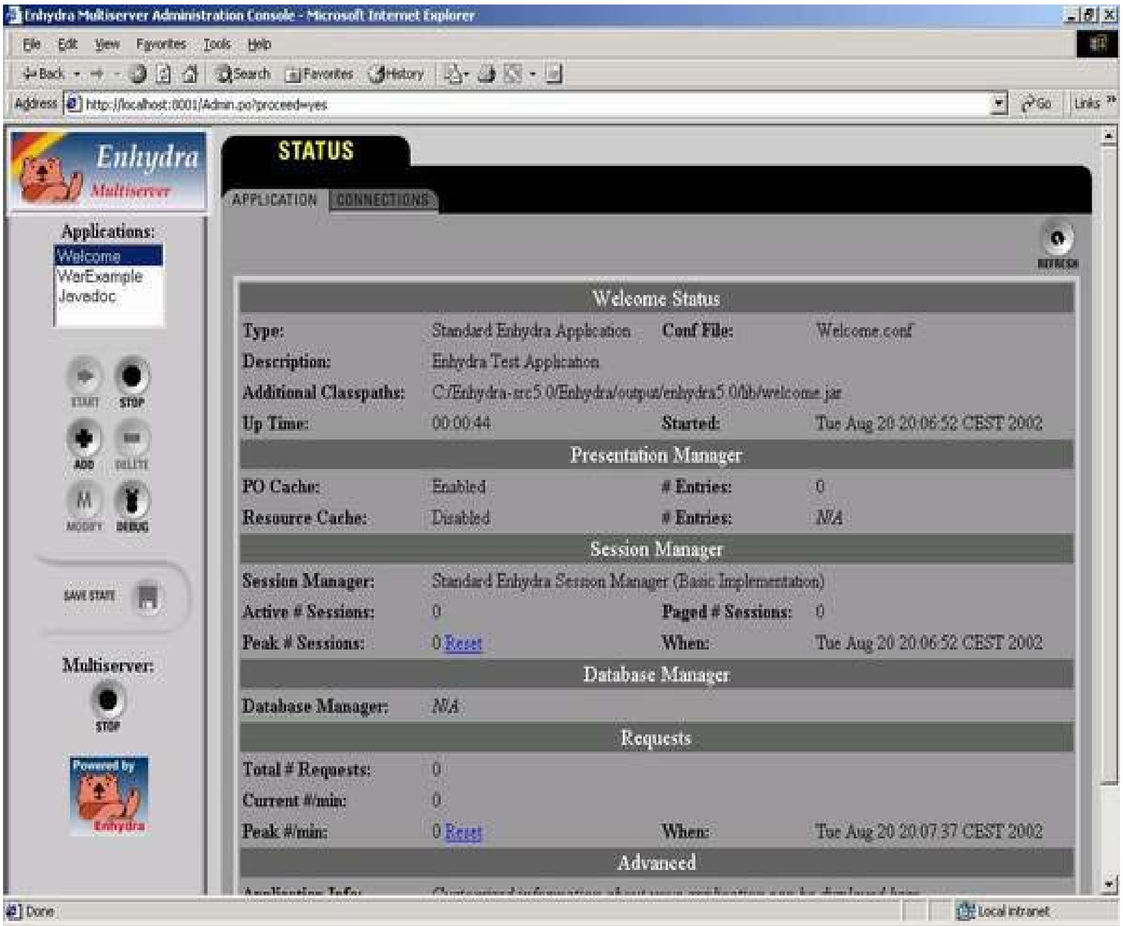
```
multiserver
```

2 In your browser, display the Admin Console by entering this URL:

http://localhost:8001/

The Admin Console displays a password dialog box. Enter the default user name, **admin**, and password, **enhydra**, to bring up the Admin Console in the browser as shown in Figure 4.3.

Figure 4.3 Admin Console display



Welcome Status			
Type:	Standard Enhydra Application	**Conf File:**	Welcome.conf
Description:	Enhydra Test Application		
Additional Classpaths:	C:/Enhydra-src5.0/Enhydra/output/enhydra5.0/lib/welcome.jar		
Up Time:	00:00:44	**Started:**	Tue Aug 20 20:06:52 CEST 2002
Presentation Manager			
PO Cache:	Enabled	**# Entries:**	0
Resource Cache:	Disabled	**# Entries:**	*N/A*
Session Manager			
Session Manager:	Standard Enhydra Session Manager (Basic Implementation)		
Active # Sessions:	0	**Paged # Sessions:**	0
Peak # Sessions:	0 Reset	**When:**	Tue Aug 20 20:06:52 CEST 2002
Database Manager			
Database Manager:	*N/A*		
Requests			
Total # Requests:	0		
Current #/min:	0		
Peak #/min:	0 Reset	**When:**	Tue Aug 20 20:07:37 CEST 2002
Advanced			

Adding simpleApp to the Admin Console

Follow these steps to add simpleApp to the Multiserver, using the Admin Console:

1 First, copy the file **simpleApp.conf** from the simpleApp/output/conf directory to <enhydra_root>/apps, the central repository for console-managed applications.

You can either do this in the Enhydra shell (UNIX) or in command window (WINDOWS) with the cp command, or by copy-and-pasting from one folder to another in Windows. For example:

```
cp simpleApp.conf /enhydra5.1/apps/
```

2 Next, in the new simpleApp.conf file, locate the server.Classpath variable. Comment out the first line, and uncomment the second line. Set the variable equal to its new absolute path, /enhydra/myapps/simpleApp/output/lib/simpleApp.jar, like this:

```
#server.Classpath[] = ../classes
Server.Classpath[] = "/enhydra/myapps/simpleApp/output/lib/simpleApp.jar"
```

Close the simpleApp.conf file.

3 In the Admin Console window, click the Add tool to display the Add New Application/Servlet window.

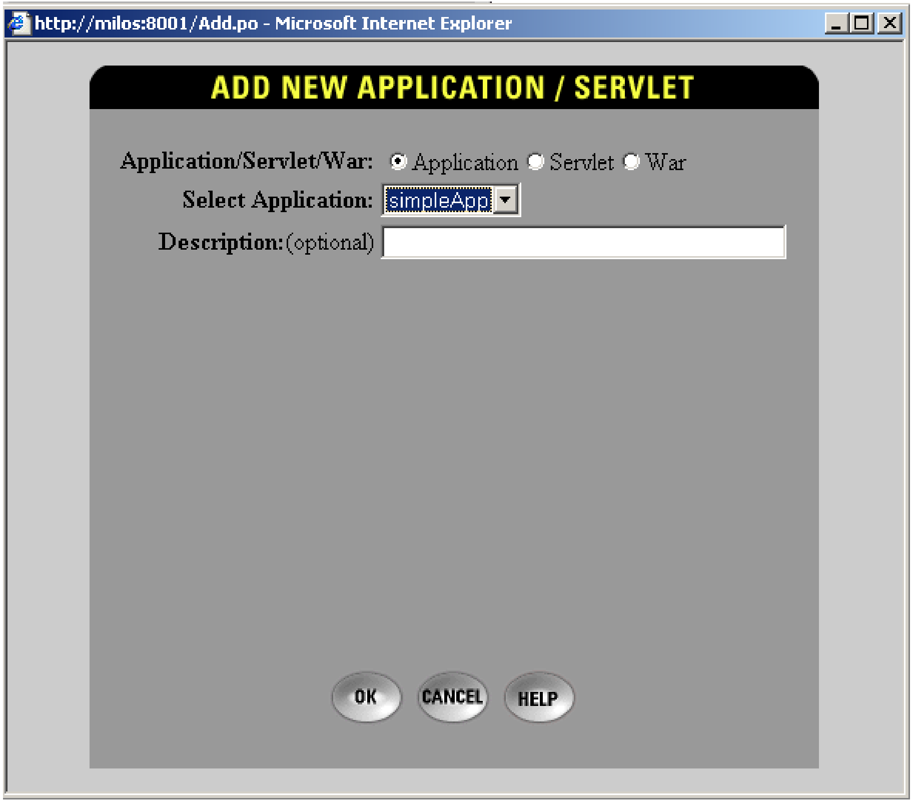
4 Since simpleApp is an application, select the Application radio button at the top of the screen.

5 Choose simpleApp from the Select Application pull-down menu.

The names in the menu represent applications with configuration files located in the *<enhydra_root>/app/* directory, but which have not been added to the console.

This list is empty when there are no more files fitting this criteria.

Figure 4.4 Adding simpleApp using the Admin Console



http://milos:8001/Add.po - Microsoft Internet Explorer
ADD NEW APPLICATION / SERVLET
OK
CANCEL
HELP

Application/Servlet/War: ● Application ○ Servlet ○ War

Select Application: simpleApp

Description: (optional)

6 Optionally, complete the Description field.

7 Click OK twice to return to the Admin Console.

The Applications window is updated to reflect the new application, with its status appearing in the content frame.

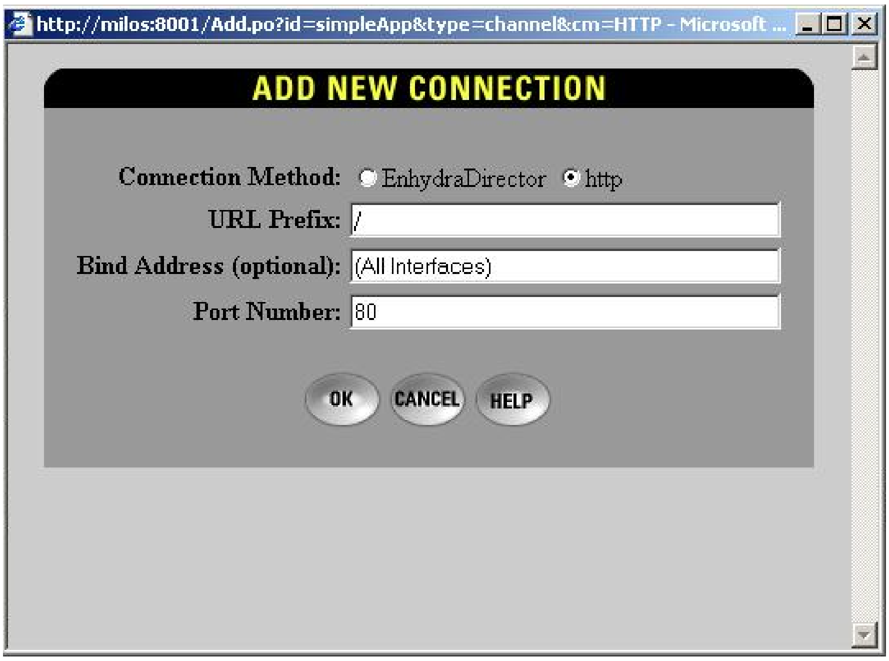
Specifying a connection method

Now it's time to specify a connection method.

1 Start by clicking the Connections tab in the content frame.

2 Click Create in the Connections screen to bring up the Add New Connection dialog box as shown in Figure 4.5.

Figure 4.5 Creating a connection for simpleApp using the Admin Console



http://milos:8001/Add.po?id=simpleApp&type=channel&cm=HTTP - Microsoft ...

ADD NEW CONNECTION

Connection Method: ○ EnhydraDirector ● http

URL Prefix: /

Bind Address (optional): (All Interfaces)

Port Number: 80

3 Choose HTTP for the Connection Method.

You have three choices for Connection Method:

- HTTP for a standard Web connection, typically in a development environment.
- HTTPS for a secure Web connection, also in a development environment. This option is only available if you have configured your Enhydra installation with Sun's Java Secure Socket Extension Kit.
- Enhydra Director for connection via a Web server.

4 Leave the URL Prefix field as is for now.

5 For Port Number, enter 8080.

In reality, you can enter any port number above 1024, as long as it doesn't conflict with a connection for another application.

6 Click OK to add the connection and return to the Connections screen.

Starting and stopping an application

The newly added application or servlet is in the stopped state. To run your newly added application,

1 Make sure **simpleApp** is still selected in the Applications window, and click the Start button.

When you start simpleApp, the Admin Console makes the URL in the Connections screen an active link.

2 Click the URL (in our example http://localhost:8080/) in the Connections screen to open a new browser and display simpleApp.

The information in the Application screen of the Admin Console updates as you view and use the application in the browser. Click the Refresh button on the Application screen to update the displayed session data.

3 To make this addition permanent beyond the current Multiserver session, click the Save State button.

When you save state, the current session data, including connection data (method, URL prefix, and port number) and application state (started or stopped) for each managed application is saved in the **multiserver.conf** file. The next time you start the Admin Console, **simpleApp** will be started by the Admin Console and included in the list of managed applications.

4 Click the Multiserver Stop button to stop the Multiserver and all managed applications.

Using XMLC

XMLC, the Extensible Markup Language Compiler, was introduced in the chapter 3, “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.

Note In general, XMLC can work with XML pages, but for practical reasons, the remainder of this chapter focuses on how it works with HTML pages.

XMLC parses a 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).

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.

1 Find the files **Welcome.html** (directory resources) and **WelcomePresentation.java** in the presentation directory.

2 Add the following line of HTML to **Welcome.html** before the closing the last </CENTER> tag:

```
<P>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.

3 Add the lines of code shown in **bold** in the following code sample, to WelcomePresentation.java:

```
  //Add the following line
static int hitCount=0;//All Welcome PO's will share this

   public void run(HttpPresentationComms comms)
       throws HttpPresentationException, IOException {
```

```
    WelcomeHTML welcome;
    String now;

    welcome = (WelcomeHTML)comms.xmlcFactory.createWelcomeHTML.class);
    now = DateFormat.getTimeInstance(DateFormat.MEDIUM).format(new Date());
    welcome.setTextTime(now);
//Increment the count and write into the html
//Add the following line
    welcome.setTextHitCount(String.valueOf(++hitCount));
    comms.response.writeDOM(welcome);
  }
}
```

4 Build the application by running **ant** from the top-level simpleApp directory. Then restart Enhydra, either by starting the Admin Console (see “Launching the Admin Console”, explained befor in this chapter) or by entering the following commands in the shell window (UNIX) or in command window (WINDOWS):

1. **UNIX**:

```
cd /myapps/simpleApp
ant
cd output
./run
```

2. **WINDOWS**:

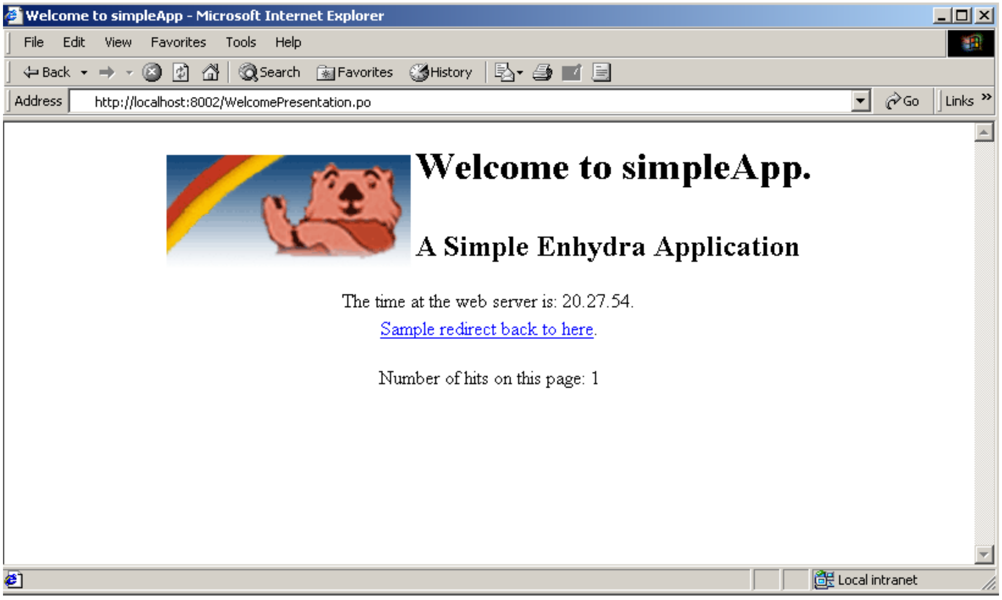
```
cd /myapps/simpleApp
ant
cd output
run
```

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

5 Test the application by loading http://localhost:8002 in your browser.

The browser will display the Welcome page for the simpleApp application. The Welcome page should now have a hit counter beneath the redirect link, as shown in Figure 4.6.

Figure 4.6 Browser displaying the simpleApp Welcome page with a hit counter



Welcome to simpleApp - Microsoft Internet Explorer

http://localhost:8002/WelcomePresentation.po

Welcome to simpleApp.

A Simple Enhydra Application

The time at the web server is: 20.27.54.

Sample redirect back to here.

Number of hits on this page: 1

The page now displays the number of times it has been accessed.

6 Reload the page several times to verify that it works correctly.

The count should increment each time you access the application.

The application is doing two things:

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

Recall that the Presentation Manager instantiates a presentation object for each request. So, the WelcomePresentation class is instantiated once per browser request. Because hitCount is a static property, it is shared by all WelcomePresentation objects and its value gets incremented by each request.

In the same way that it added a **getElementTime()** 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 Document Object Model

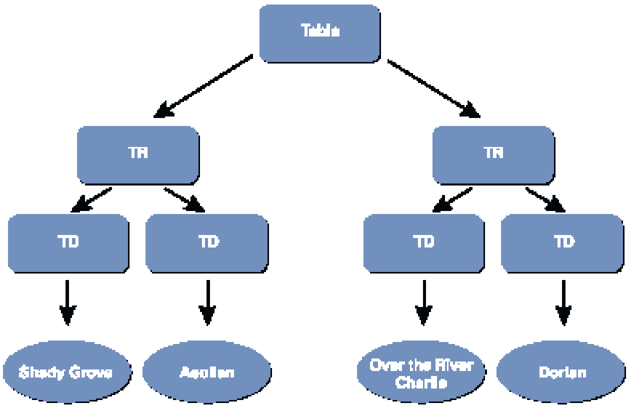
HTML documents have a hierarchical or tree-like structure that can be modeled in an object-oriented language like Java. The Worldwide Web Consortium (W3C) standard for the XML/HTML object model is called the Document Object Model (DOM).

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, which in turn contain `<TD>` tags containing text (or data). This defines a tree-like hierarchy, as illustrated in Figure 4.7.

Figure 4.7 DOM tree of HTML



Each box or ellipse in this figure is a ***node*** in the tree. The node above another node in the hierarchy is called its ***parent***. The nodes below the parent are called its ***children***.

Some nodes (like HTML tags) have attributes (for example, a table cell has a background color attribute). 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.

For example, use code like the following to set the color of one of the table cells:

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

In this example, the class **HTMLTableElement** and the method **setBgColor()** come from the W3C packages; **getElementCellOne()** comes from XMLC.

This code illustrates one important thing that 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 be 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 (CSS). Outside of that, they are largely ignored by browsers. However, XMLC makes extensive use of them.

- 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 inline tag. An *inline tag* is any tag that does not cause a line break in the layout; for example, <A> (anchor) or <B> (bold) tags.

Note 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 cause a line break in the HTML layout.

Using XMLC from the command line

Previously, you ran XMLC implicitly when you built the project with **ant**.

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**.

-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.

1 Create a new file called Simple.html in the simpleApp/src/simpleapp/resources directory.

2 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>
```

3 Change to the resources directory and enter this command:

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

XMLC displays the following in the shell window (UNIX) or in command window (WINDOWS):

```
DOM hierarchy:
  LazyHTMLDocument:
    HTMLHtmlElementImpl: HTML
      HTMLHeadElementImpl: HEAD
        HTMLTitleElementImpl: TITLE
            LazyText: Simple Enhydra Page
      HTMLBodyElementImpl: BODY
        HTMLHeadingElementImpl: H1: id="MyHeading"
            LazyText: Ollie Says
        LazyText: The current time is
            LazyHTMLElement: SPAN: id="time"
                LazyText: 00:00:00
            LazyText: .
```

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 the 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.

-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. To create a class with a different name, 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. The source file is useful primarily for understanding how XMLC and the DOM API works. Use the **-keep** option to keep the Java source file.

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

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

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

2 Open **SimpleHTML.java** and look at the generated code.

Within the file, you will find two methods, **getElementMyHeading()** and **getElementTime().** XMLC recognized the ID attributes of the heading and <SPAN> tags in Simple.html and generated these methods. Look through the file to get an idea of the object that XMLC creates for a very simple document.

3 Once you are done looking at SimpleHTML.java and SimpleHTML.class, delete them.

You are 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 more topics essential to Enhydra application development:

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

Maintaining session state

Because 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” in the chapter 3.

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 these methods:

- A set() method to which you pass a string key and an object to store
- A get() method which 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 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.

1 Add these four lines of HTML just before the closing the last `</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>
```

2 Now add this import statement to **WelcomePresentation.java:**

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

3 In **WelcomePresentation.java**, add this member property to the WelcomePresentation class (just after **hitCount**):

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

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

4 Add the following code to the **run()** method, placing the line beneath the line you added with the **setTextHitCount** method.

You can find this code in the SessionMaint.java file located in the *<enhydra_root>/doc/getting_started/samples* directory.

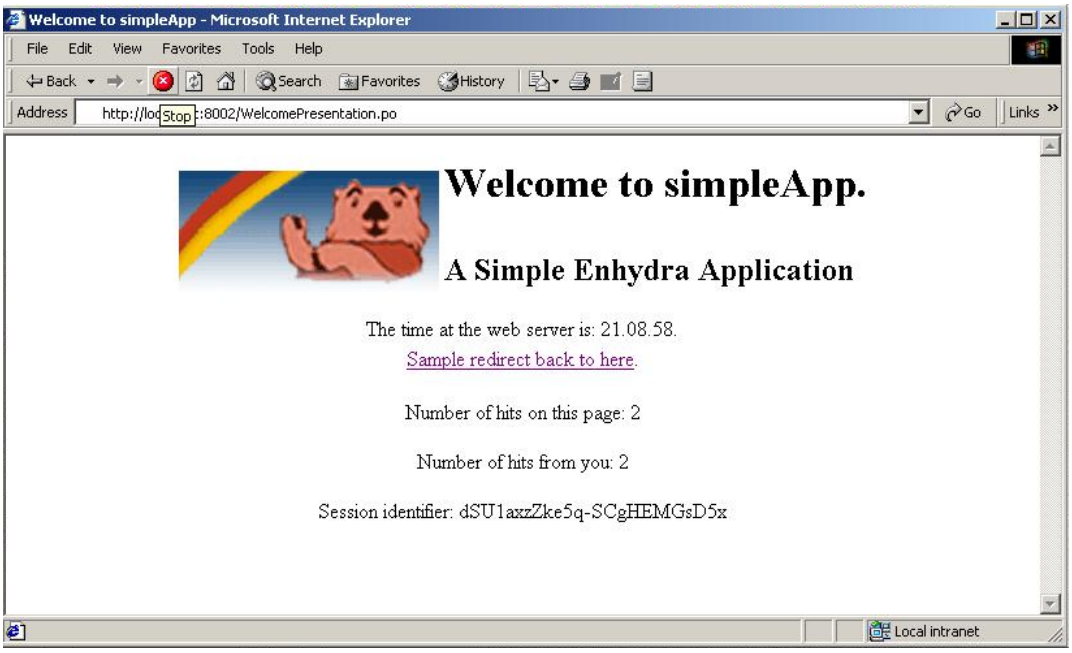
```
try {
  Integer personalHits = (Integer)comms.session.getSessionData().get(hits);
  if(personalHits == null) {
    personalHits = new Integer(1);
  } else {
    personalHits = new Integer(personalHits.intValue() + 1);
  }
  comms.session.getSessionData().set(hits, personalHits);
  // Save personalHits 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.lang.Objects, you have to typecast it to Integer. If the object has not been previously stored in the session, the code creates a new Integer of value 1 (one). If the object exists, the value 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.

5 Rebuild and start the application, and access the page with your browser. The Welcome page displays the total number of hits, as well as the number of hits from a particular client, as well as the client’s unique Session Identifier, as shown in Figure 4.8.

Figure 4.8 Browser displaying the simpleApp Welcome page with a Session Identifier


Welcome to simpleApp - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://loc Stop :8002/WelcomePresentation.po
Go Links
Local intranet

Welcome to simpleApp.

A Simple Enhydra Application

The time at the web server is: 21.08.58.

Sample redirect back to here.

Number of hits on this page: 2

Number of hits from you: 2

Session identifier: dSU1axzZke5q-SCgHEMGsD5x

Because you are running the application on your Localhost server, it is not accessible to any other clients, so these numbers will always be the same. However, if the application was run 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.

To create a new presentation object:

1 Copy the file **WelcomePresentation.java** and call it **SimplePresentation.java**.

2 Open **SimplePresentation.java** and change the name of the class from **WelcomePresentation** to **SimplePresentation**.

3 Remove all the session-related code.

4 Change all the occurrences of **WelcomeHTML** to **SimpleHTML**.

5 Change the **welcome** identifier to **simple**.

Now you have “stripped down” presentation object. The run() method should look like this:

```
public void run(HttpPresentationComms comms)
      throws HttpPresentationException, IOException {

      SimpleHTML simple;
      String now;

      simple = (SimpleHTML)comms.xmlcFactory.create(SimpleHTML.class);
```

```
        now = DateFormat.getTimeInstance(DateFormat.MEDIUM).format(new Date());
        simple.setTextTime(now);

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

6 Add these statements at the top of the file, after the other import statements:

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

7 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 to center the heading on the page
- Gets the child object of the heading (a Text object)
- Sets a new value for the text string, “Mr. Ollie Otter says:”

You could have done the same thing by putting a <SPAN> tag around the text in the heading. XMLC would then have generated a **setTextMethod()** that you could have called in the code. This example, however, 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.

The convention is to create fooHTML class for a file foo.html.

8 Save all files and run **ant** in the applications root directory to build the package.

To create a link from the Welcome page to your new page:

1 Add the following HTML at the bottom of **Welcome.html**:

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

2 If you have not already done so, stop your Multiserver by pressing Ctrl-C in the shell window (UNIX), or in command window (WINDOWS), and build the application from the top level:

1. **UNIX**:

```
cd /enhydra/myapps/simpleApp
ant
cd output
./run
```

2. **WINDOWS**:

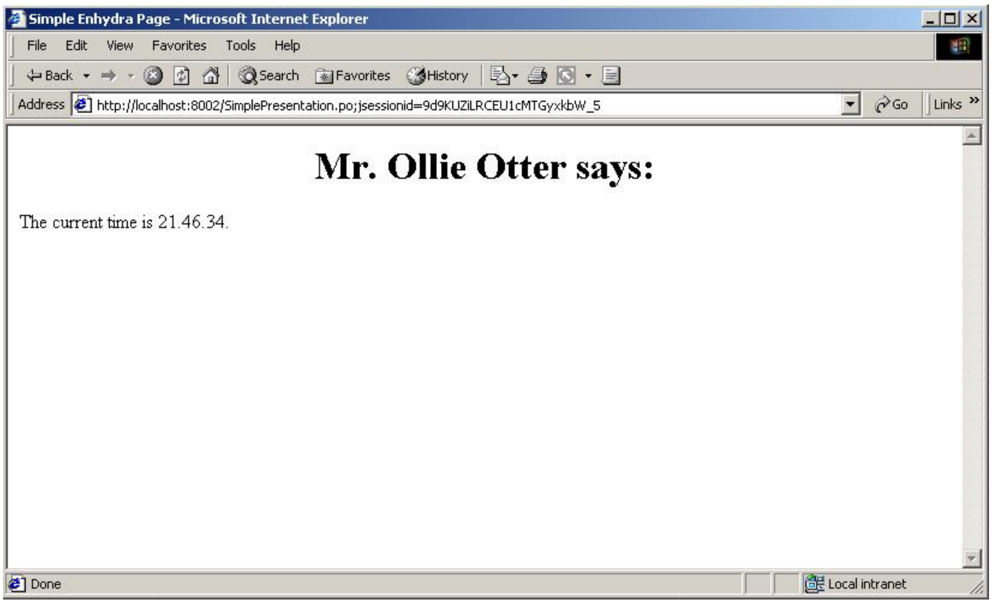
```
cd /enhydra/myapps/simpleApp
ant
cd output
run
```

3 Now access the application from your browser, as you did before.

4 Click the **Go to Simple Page** link to view the SimplePresentation PO.

The Simple presentation object has only a heading and the current time. You're going to make this page more interesting in the next section.

Figure 4.9 Browser displaying the simpleApp Simple presentation object


Simple Enhydra Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
History
Address
Go
Links
Done
Local intranet

http://localhost:8002/SimplePresentation.po;jsessionid=9d9KUZiLRCEU1cMTGyxkbW_5

Mr. Ollie Otter says:

The current time is 21.46.34.

Populating a table

Another common task in Web application development is populating 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:

- Create the table in HTML
- Programmatically populate the table
- Rebuild and run the application

Create the table in HTML

In the **Simple.html** file, create an HTML table with a template row and an ID attribute.

1 Edit the file **presentation/Simple.html** in your simpleApp project.

2 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/getting_started/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 (in the second `<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

To programmatically populate the table, edit the presentation object corresponding to **Simple.html**. In the following steps, you will add code to **Simple.java** to iteratively replace the HTML table elements with text from an array of strings.

1 Copy the file <*enhydra_root*>/doc/getting_started/samples/TableCode.java into your application’s presentation directory and rename it **SimplePresentation.java**.

Note If you like, you can save your old SimplePresentation.java to SimplePresentation.sav for future reference.

2 Now, look at your new **SimplePresentation.java**.

In addition to the standard features of a presentation object, the first thing you’ll notice in this code is a member property 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", "Jazz", "Yes" },
{ "Funky Urchin", "Lovely Spines", "Techno Pop", "Yes" },
{ "Stinky Pups", "Shark Attack", "Hardcore", "No" } };
```

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 each ID in the document to 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 that calls it; in this case, templateRow. The boolean argument true 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.

1 If you have not already done so, stop your Multiserver by pressing Ctrl-C in the shell window (UNIX), or in command window (WINDOWS), and build the application from the top level:

1. **UNIX**:

```
cd /enhydra/myapps/simpleApp
ant
cd output
./run
```

2. **WINDOWS**:

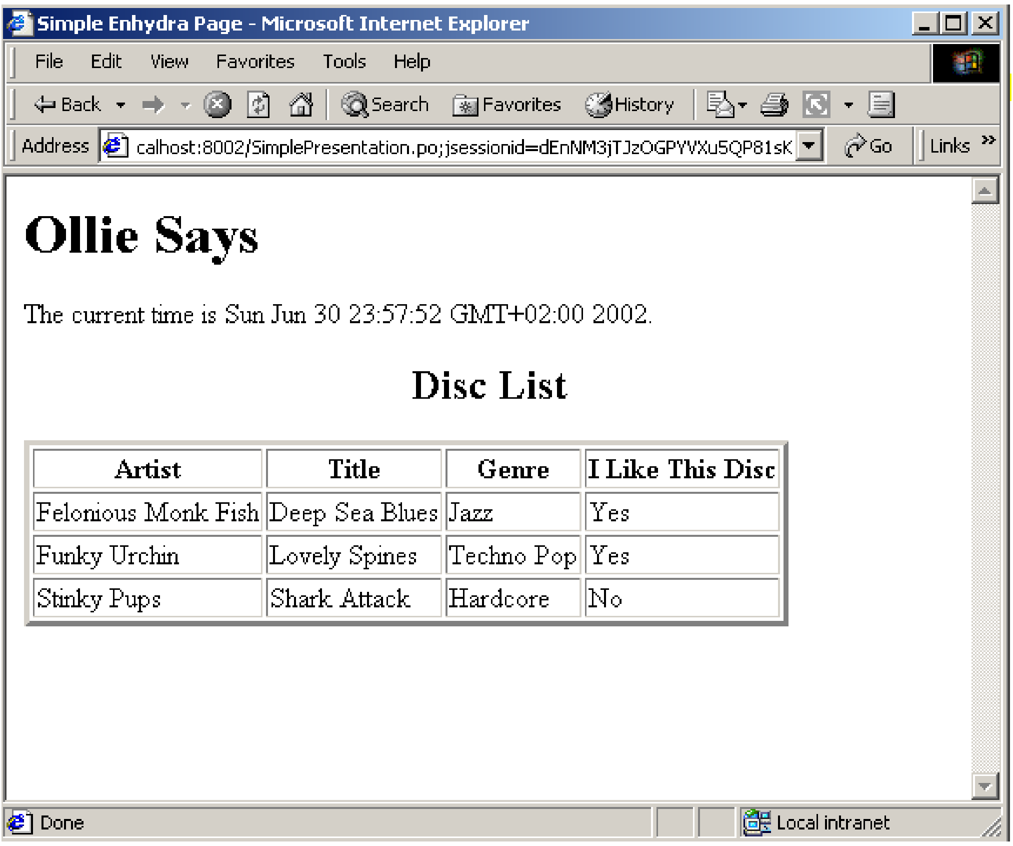
```
cd /enhydra/myapps/simpleApp
ant
cd output
run
```

2 Now access the application from your browser, as you did before.

3 Click the Go to Simple Page link to view the new SimplePresentation PO.

The Simple presentation object now includes a Disc List table, as shown in Figure 4.10.

Figure 4.10 Simple PO with a programmatically populated Disc List table


Simple Enhydra Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address
Go
Links
Done
Local intranet

calhost:8002/SimplePresentation.po;jsessionid=dEnNM3jTJzOGPYVXu5QP81sK

Ollie Says

The current time is Sun Jun 30 23:57:52 GMT+02:00 2002.

Disc List

Artist	Title	Genre	I Like This Disc
Felonious Monk Fish	Deep Sea Blues	Jazz	Yes
Funky Urchin	Lovely Spines	Techno Pop	Yes
Stinky Pups	Shark Attack	Hardcore	No

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 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 illustrate a basic role of business objects as you proceed.

To add a business object:

1 Create a new file called **SimpleDiscList.java** in the business directory, simpleApp/src/simpleapp/business. Because SimpleDiscList.java is in your application's business package, the first line in the file will be:

```
package simpleapp.business;
```

2 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 **discList**) in the body of the run method:

```
public class SimpleDiscList {
   String[][] _discList =
      {   { "Felonious Monk Fish", "Deep Sea Blues", "Jazz", "Yes" },
         { "Funky Urchin", "Lovely Spines", "Techno Pop", "Yes" },
         { "Stinky Pups", "Shark Attack", "Hardcore", "No" } };
      public SimpleDiscList() {
   }
   public String[][] getDiscList() {
```

```
            return _discList;
        }
    }
```

3 Now, back in the presentation directory, edit **SimplePresentation.java** as follows:

1 Import the new class:

```
import simpleapp.business.SimpleDiscList;
```

2 Add these two lines to create an instance of your new business object, and call it **getDiscList()** method. These lines take the place of the static array initializer in the previous section.

```
SimpleDiscList sdl = new SimpleDiscList();
String[][] discList = sdl.getDiscList();
```

4 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. In that case, 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, InstantDB, DB2 and QED.

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

Important For this section of the tutorial, it is assumed that you have installed InstantDB and set the CLASSPATH environment variable to include the InstantDB JAR files. Refer to the Enhydra CD for installation instructions. If you choose to use a different database, you must edit the CLASSPATH accordingly.

Creating a database table

The remainder of this section requires the existence of a specific table in your database, so you need to create that table before proceeding. This section tells you how to create a table in your InstantDB database.

Note Most databases provide a tool for directly executing SQL statements. For example, InstantDB provides ScriptTool and Oracle supplies SQL*Plus. It is important to note, however, that the SQL format varies from database to database. The SQL sample provided for this part of the tutorial works with InstantDB, and may not work with other databases.

To create a table in InstantDB:

1 Create a new directory called **data** for your database.

```
cd /enhydra/myapps
mkdir data
```

2 Copy the sample properties file, **sample.prp**, from the InstantDB Examples directory into your data directory, and rename it **simpleApp.prp**. You can do this from the command line with a command like the following:

```
cd /data
cp /idb/Examples/sample.prp simpleApp.prp
```

3 Copy the **tutorial_create_idb.sql** SQL file located in the samples directory <enhydra_root>/doc/getting_started/samples, into your data directory. For example, the copy command might look like the following:

```
cp /usr/local/lutris-enhydra3.5/doc/getting_started/samples/tutorial_create_idb.sql
```

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

4 From the command line, enter the following command to run the ScriptTool using the **tutorial_create_idb.sql** file as input to create the LE_TUTORIAL_DISCS table:

```
java org.enhydra.instantdb.ScriptTool tutorial_create_idb.sql .
```

Look in the data directory and notice the changes. New directories have been created, including a tables directory.

Note The samples directory contains a SQL file for Oracle databases. Here is the command to create a table in Oracle using SQL*Plus and the supplied SQL file:

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

5 Use InstantDB's DBBrowser to verify that the table was created correctly.

1 Enter the following command to start DBBrowser:

```
java -Xms16m -Xmx32m org.enhydra.instantdb.DBBrowser
```

2 In DBBrowser, click Browse to select a database. Locate and select **simpleApp.prp** and click Open.

3 Click Connect to connect to the database.

4 Select LE_TUTORIAL_DISCS from the Tables column, and click Submit to query the database.

The default query is SELECT * FROM LE_TUTORIAL_DISCS, and DBBrowser should display the following table data in response.

```
Rockin Apps,Enhydra Orchestra,Rock and Roll,1
Beethoven Symphony No.9,LA Philharmonic,Classical,1
Material Girl,Madonna,Modern Rock,0
```

5 Click Disconnect to disconnect from the simpleApp database, and then close DBBrowser.

Note The **samples** directory contains a SQL file, tutorial_create.sql, that works with Oracle databases. Here is the command for Oracle SQL*Plus:

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

If you are using a database other than InstantDB, see 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 tells you how to write and execute a simple standalone program to establish a JDBC connection to the database server. Starting with a standalone program lets you isolate any problems that may occur. If you have already configured JDBC on your system, you can skip this section.

In our simple program, we will do the following:

- Load the JDBC driver
- Get a database connection using the appropriate ***connection string***
- Create a statement object
- Run a query
- Print the results

For the program to run, you must have installed and configured your database, and created the LE_TUTORIAL_DISCS table.

Note This example works specifically with InstantDB. If you want to adapt the program to work with another database, you will need to change the driver information in the program, as well as the connection string. Refer to the appendix A, “Database configurations,” for configuration information for other databases.

To use this program:

1 Create a **jdbcTest** directory for the test program and make it the current directory.

For example, to create the new directory in the **tmp** directory for your database, you might use the following commands:

```
cd /enhydra/myapps/data/tmp
mkdir jdbcTest
cd jdbcTest
```

2 Copy JDBCTest.java from the samples directory into the new directory using the cp command.
For example,

```
cp usr/local/lutris-enhydra3.5/doc/getting_started/samples/JDBCTest.java
```

Look at the program code to verify the JDBC driver and connection string. The JDBC driver and the connection string are shown in **bold** in the following code sample from the program.

```
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 {
      // To test with a different database, replace JDBC driver information below
```

```
        Class.forName("org.enhydra.instantdb.jdbc.idbDriver");
        /* To test with a different database, provide appropriate connection data
        (e.g., database connection string, username, and password) */
        con = DriverManager.getConnection( "jdbc:idb:/enhydra/myapps/data/simpleApp.prp");
        stmt = con.createStatement();
        rs = stmt.executeQuery("SELECT * FROM LE_TUTORIAL_DISCS");
        rs.next();
        System.out.println("Title = " + rs.getString("title") + " -- Artist = " + rs.getString("artist"));
        con.close();
      }
      catch(ClassNotFoundException e) {
        System.err.println("Couldn't load the driver: "+ e.getMessage());
      }
      catch(SQLException e) {
        System.err.println("SQLException caught: " + e.getMessage());
      }
      }
  }
```

3 Edit the **JDBCTest.java**, if necessary, to update the connection string. These appear in the call to **getConnection(),** the second statement in the try block (see the previous code example).

4 Compile the file using the javac command:

```
javac JDBCTest.java
```

5 Run the program:

```
java JDBCTest
```

If you have populated the table as instructed previously, you will see the following in the shell window (UNIX), or in command window (WINDOWS):

```
main SELECT * FROM LE_TUTORIAL_DISCS
Title = Rockin Apps -- Artist = Enhydra Orchestra
Database simpleApp is shutting down...
Database simpleApp shutdown complete.
```

If there was an error, you will see some exception messages in the shell window (UNIX), or in command window (WINDOWS), that should help you isolate the problem. Refer to your database JDBC documentation, and confirm your database driver and connection string information.

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. It is a good idea to set the CLASSPATH in your application's run script. This prevents conflicts in the event that you have multiple applications using different drivers. This also makes your application more portable.

There are two copies of the run script: the template run.in is located in the application's input directory, and run is in the output directory. In template run.bat.in external file setclass.bat (for WINDOWS) is called to set CLASSPATH. As with configuration files, building the application overwrites the run script in the output directory with the template file. Therefore, set the CLASSPATH in the script located in the input 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.

To put JDBC drivers in the CLASSPATH for your application:

1 Edit the file simpleApp/input/**run.in** and add the following lines at the beginning of the section titled "Build up classpath:"

In UNIX:

```
CLASSPATH="<JDBC_LIB>"
export CLASSPATH
```

In WINDOWS:

```
SET CLASSPATH="<JDBC_LIB>"
```

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

```
CLASSPATH=/idb/Classes/idb.jar\;/idb/Classes/jta-spec1_0_1.jar
```

Be careful not to put any blank spaces in this line because they will prevent the script from working properly.

Note For ease of maintenance and greater portability, it is desirable to make the path to your JDBC driver library a variable. This can save time if you change drivers and need to update your CLASSPATH. The DiscRack and AirSent example projects both use variables for the path to a JDBC driver library. The example projects are located in the <enhydra_root>/examples directory.

Configuring the Database Manager

Now that you have verified your JDBC connection, you need to provide the database connection parameters to your simpleApp application.

You do this in the application configuration file, **simpleApp.conf**. In this section, you will edit the template file, **simpleApp.conf.in**. The ant utility copies this file to the output/conf directory after every build.

1 Open simpleApp/input/conf/**simpleApp.conf.in** in a text editor.

2 Add the following lines to the bottom of the file:

```
#-----------------------------------------------------------------------------
# Database Manager Configuration
#-----------------------------------------------------------------------------
DatabaseManager.Databases[] = "simpleApp"
DatabaseManager.DefaultDatabase = "simpleApp"
DatabaseManager.Debug = "false"
DatabaseManager.DB.simpleApp.ClassType = "Standard"
DatabaseManager.DB.simpleApp.JdbcDriver = "org.enhydra.instantdb.jdbc.idbDriver"
DatabaseManager.DB.simpleApp.Connection.Url =
"jdbc:idb:/enhydra/myapps/data/simpleApp.prp"
DatabaseManager.DB.simpleApp.Connection.User = ""
DatabaseManager.DB.simpleApp.Connection.Password = ""
DatabaseManager.DB.simpleApp.Connection.MaxPoolSize = 30
DatabaseManager.DB.simpleApp.Connection.AllocationTimeout = 10000
DatabaseManager.DB.simpleApp.Connection.Logging = false
DatabaseManager.DB.simpleApp.ObjectId.CacheSize = 20
DatabaseManager.DB.simpleApp.ObjectId.MinValue = 99
```

Note This is an example of the configuration file for InstantDB. For information on using your application with other databases, see the appendix A, "Database configurations."

3 Verify that all the items shown in italics match your connection parameters as follows:

- Verify that the Databases[] and DefaultDatabase entries are set to simpleApp.

- Ensure simpleApp is used for the database ID attribute. For example, in the line

```
DatabaseManager.DB.simpleApp.ClassType = "Standard",
simpleApp is the database ID attribute.
```

Refer to the appendix A, “Database configurations,” for addtional information.

- Verify the JDBC driver entry. This example uses the driver for InstantDB:

```
org.enhydra.instantdb.jdbc.idbDriver
```

- Verify the connection string. This example uses a connection string for InstantDB:

```
jdbc:idb:/enhydra/myapps/data/simpleApp.prp
```

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

4 Save and close the configuration file.

5 Build the application and propagate your changes by running ant from the application root directory, simpleApp. For example:

```
cd /enhydra/myapps/simpleApp
ant
```

Adding data access functionality

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”, explained before in this chapter. 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.

1 Copy the file **SimpleDiscQuery.java** from <enhydra_root>/doc/getting_started/samples to your data layer (that is, the simpleApp/src/simpleapp/data directory).

2 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:

```
connection = 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, vResultSet, consisting of a Vector, vRow, for each row in the result set. Although each row is known to contain only four elements (because 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.

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

- Add two import statements at the top:

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

- Add a member variable corresponding to the data object:

```
SimpleDiscQuery _sdq;
```

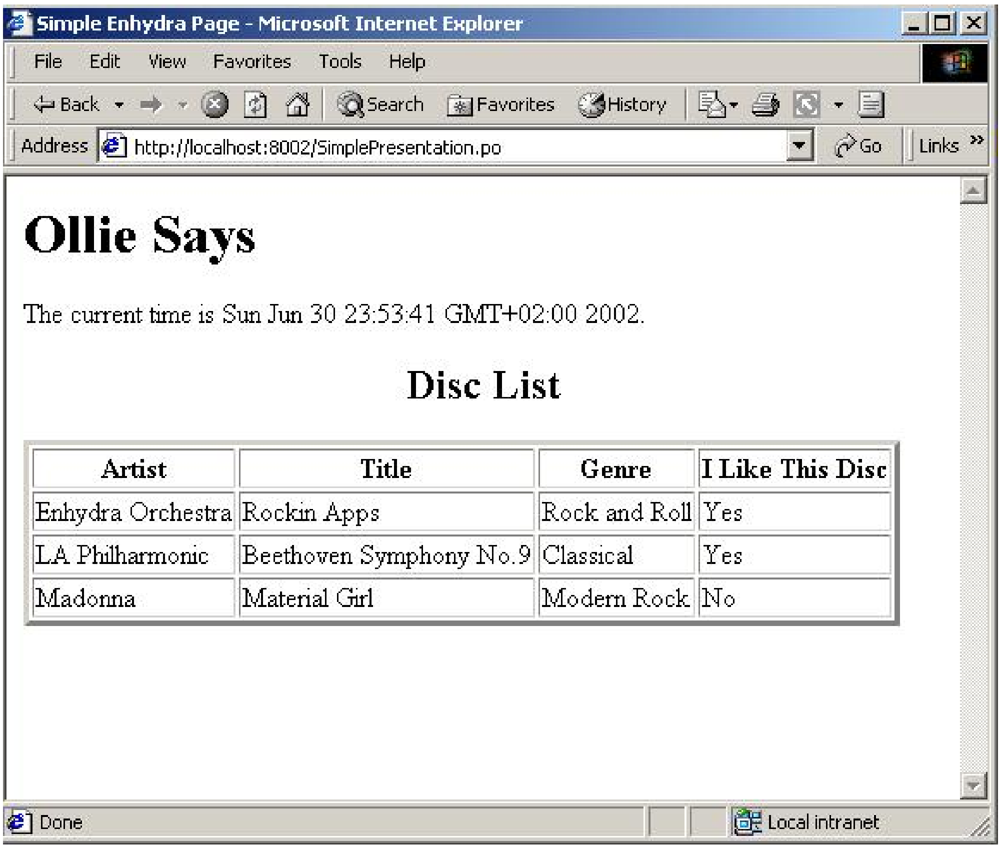
- Replace the body of the constructor **SimpleDiscList()** with the code in the new file
- Replace the body of the **getDiscList()** method with the code in the new file. It converts the Vector of Vectors returned by **query()** to a two-dimensional String array that the presentation object expects.
  Notice that you did not have to change the presentation object at all. The data object provides a buffer between the presentation object and the data object.

4 Now, build the application from the top level, using the ant command.

```
cd /enhydra/myapps/simpleApp
ant
```

5 When you get the application to compile, try running it.
The Disc List table in your Simple presentation object should now contain data from your database, as shown in Figure 4.11.

Figure 4.11 Simple PO with a Disc List table generated from the database


Simple Enhydra Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost:8002/SimplePresentation.po
Go Links
Done
Local intranet

Ollie Says

The current time is Sun Jun 30 23:53:41 GMT+02:00 2002.

Disc List

Artist	Title	Genre	I Like This Disc
Enhydra Orchestra	Rockin Apps	Rock and Roll	Yes
LA Philharmonic	Beethoven Symphony No.9	Classical	Yes
Madonna	Material Girl	Modern Rock	No

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

Using DODS

The Data Object Design Studio (DODS) is a tool that for a given input DOML file 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 one part of Enhydra 5.1. If Enhydra 5.1 is installed, so is DODS. In this case, DODS home directory *<dods_home> is: <enhydra_home>/dods.*

Since this version, DODS has become independent from Enhydra, which means that can be used without it. In this case, DODS home directory *<dods_home>* is the directory in which independent DODS is installed.

DODS independence is detailly explained in section "DODS independence", later in this chapter.

DODS source building

If Enhydra5.1 is installed, DODS is included within, so nothing more needs to be done.

If DODS will be used independent (without Enhydra), it is necessary to do the following things:

- Edit file build.properties in the <DODS_SOURCE> directory (<DODS_SOURCE> is location of DODS source).

  In build.properties file, you must specify the following variable:

  jdk.dir - the path to your installation of the Java Development Kit

  For example:

  ```
  jdk.dir=C:/j2sdk1.4.0
  ```

  Please note that Unix stile slashes (/) must always be used instead of Dos stile backslashes (\).

  Beside this variable, you can specify the variable:

  dods.inst.dir

  This is location of the directory in which DODS will be built.The default directory is "output".

  If you want to change location of any of external jars (ant.jar, optional.jar, log4j.jar, avalon-framework-cvs-20020315.jar, batik.jar, fop.jar, xerces.jar, xalan.jar, util.jar) or DocBook directory, you can specify the new location in this file. The procedure is the same as for directory for building Enhydra.

- If you don't have ant.bat file in your system PATH, you have to put it there.

- Start Command Promt and go to <DODS_SOURCE> directory.

DODS building is completely Ant based. You can give one of the following options to the compile command:

- **compile buildAll** - builds and configures DODS with javadoc and docbook documentation
- **compile buildOptimize** - builds, optimizes and configures DODS with javadoc and docbook documentation
- **compile buildNoDoc** - builds and configures DODS without documentation building
- **compile copyBuild** - copies and configures DODS without source compiling
- **compile buildDistribution** - builds and configures DODS with javadoc and docbook documentation and creates distribution; nsis 2.0b should be included in DODS if doesn't exist (files makensis.exe and makensisw.exe in Dods/Install/Windows/install directory)
- **compile buildOptimizeDistribution** - builds and configures DODS with javadoc and docbook documentation and creates optimized distribution; nsis 2.0b should be included in DODS if doesn't exist (files makensis.exe and makensisw.exe in Dods/Install/Windows/install directory)
- **compile clean** - removes the output folder (in order to start a new compilation from scratch)

where <DODS_HOME> is directory in which DODS is built.

- After DODS building, you MUST add <DODS_HOME>\bin directory to the begining of the system PATH.

DOML file syntax

This section describes the syntax of DOML files, which are used by the Data Object Design Studio (DODS) to generate data access code for Enhydra applications.

Structure

The hierarchy of tags in a DOML file is:

```
<doml>
  <database>
    <package>
      <package>
      ....
      </package>
      <table>
        <column>
          <type/>
          <referenceObject/>
          <initialValue/>
          <javadoc/>
        </column>
        <index>
          <indexColumn/>
        </index>
      </table>
    </package>
  </database>
</doml>
```

Tag reference

This section contains an alphabetical reference of all the tags allowed in DOML files. Each entry corresponds to an XML tag, and contains the subsections:

- **Content** - tags that the tag can contain.
- **Attributes** - attributes the tag can have.
- **Context** - tags within which the tag can appear, in other words, the tags that can contain it.

So, for a tag <*sampleTag*>*,* whose attributes are attribute1, attribute2, and so on,
whose context is <*contextTag*>*,* and which can contain contentTag, its general syntax
would look like:

```
<contextTag>
   <sampleTag attribute1 attribute2 ...>
      <contentTag/>
   </sampleTag>
</contextTag>
```

<author>

Author of doml project.

Content	None
Attributes	None
Context	< doml >

<column>

<column> describes a column in a database table.

Content	<error> <javadoc> <referenceObject> <type> <initialValue>
Attributes	**id** The name of the column in the database table. **usedForQuery** Specifies whether the values of the column will be used for queries. The possible

values for usedForQuery are:

- true
- false

isConstant

Specifies whether the column contains a constant value. The possible values for isConstant are:

- true
- false

isPrimaryKey

This attribute specifies whether the table column will be a primary key. If isPrimaryKey is true, a primary key is created on this column in the table. The possible values for isPrimaryKey are:

- true
- false

isUnique

Specifies whether the column must contain unique values. The possible values for isUnique are:

- true
- false

Context <table>

<database>

Contains the package hierarchy, and specifies the database.

Content <package>

Attributes **database**

The database attribute specifies the database vendor. The valid types are as follows:

Table: Valid database attributes:

1. Standard - generated SQL code will conform to standard JDBC SQL. This is

the default value.

2. Oracle - DODS will generate SQL optimized for Oracle databases.
3. Informix - DODS will generate SQL optimized for Informix databases.
4. MySQL - DODS will generate SQL optimized for MySQL databases.
5. MSQL - DODS will generate SQL optimized for MSQL databases.
6. Sybase - DODS will generate SQL optimized for Sybase databases.
7. PostgreSQL - DODS will generate SQL optimized for PostgreSQL databases.
8. DB2 - DODS will generate SQL optimized for DB2 databases.
9. QED - DODS will generate SQL optimized for QED databases.

templateset

Template set that will be used for java code generation. The possible values for templateset are:

- standard
- multidb
- <any user defined template>

Default value is "standard".

Context <doml>

<doml>

Root element of DOML files. This tag contains a database hierarchy that contains all the packages.

Content <database>

<author>

<project>

Attributes None

Context None

<index>

<index> is used to specify index columns.

Content <indexColumn>

Attributes **id**

The name of the index constraint in the database table.

Unique

Specifies whether the index constraint is unique. The possible values for unique are:

- true
- false

clustered

Specifies whether the index constraint is clustered. The possible values for clustered are:

- true
- false

Context <table>

<indexColumn>

<indexColumn> is used to specify each index column in the constraint index.

Content None

Attributes **id**

The name of the column in the database table.

Context < index >

<initialValue>

<initialValue>is used to specify a default initial value for the column.

Content None

Attributes None

Context <column>

<javadoc>

The <javadoc> tag contains the text for Javadoc entries for the column.

Content	None
Attributes	None
Context	<column>

<package>

Each package can contain a sub-package or a table structure.

Content	<package> <table>
Attributes	**id** The name of the package. The format for the name includes the parent package's id value. For example, if I had a package myPackage, and a sub-package of it called mySubPackage, mySubPackage's id value would be myPackage.mySubPackage.
Context	<database>

<project>

Contains information about project.

Content	None
Attributes	None
Context	< doml >

<referenceObject>

If the column is a reference to another table, <referenceObject> specifies the table.

Content	None
Attributes	**Constraint** Specifies whether the specified table row must exist. The possible values for constraint are: • true • false

foreignKeyColumn

Specifies the column in the referenced table.

ForeignKeyGroup

Specifies the group of the foreign keys.

Reference

Specifies the ID of the referenced table.

Context <column>

<table>

<table> describes a table in a database.

Content <column>

Attributes **id**

Similar to the id attribute in <package>, <table>'s id contains the value of the table name located in the package. For example, if I had a package myPackage, a subpackage mySubPackage, and a table myTable, the id value is myPackage.mySubPackage.myTable.

DbTableName

The actual SQL table name. By default this is the same as the id value, minus the package information. For example, myPackage.mySubPackage.myTable's dbTableName is myTable.

isLazyLoading

This attribute specifies whether the DO will use lazy loading. If a DO uses lazy loading, when you supply a known ObjectId to create a DO instance, the DO instance is created but the corresponding row in the table is not retrieved until the first get() or set() method call is made. It delays the hit on the database until the moment the data is actually needed. The possible values for isLazyLoading are:

- true
- false

IsIndex

This attribute specifies whether the data object is index. The possible values for isIndex are:

- true
- false

Default value is "false".

IsAbstract

This attribute specifies whether the data object is abstract. In DODS, all data objects that are extended must be abstract. This is because of how DODS handles database tables. Only non-abstract leaf classes have tables in the database. The possible values for isAbstract are:

- true
- false

IsView

This attribute is not currently used by DODS.

ExtensionOf

This attribute is not currently used by DODS. The name of another table of which this table is an extension. The value of extensionOf must match the id of a previously defined table.

notUsingOid

Specifies whether the table is oid based.

Context <package>

<type>

<type> dictates the form of the data stored in the column. If no <type> is specified, the column contains all default values.

Content None

Attributes **Size**

Specifies the size of data types that are commonly measured in width, like VARCHAR. Size must be an integer.

CanBeNull

Specifies whether the column can contain null values. The possible values for canBeNull are:

- true
- false

dbType

Specifies the internal SQL data type the database will use for this column. The default value of dbType is VARCHAR.

JavaType

Specifies the Java data type returned by the DO to the user when querying this attribute of the DO. The default value of javaType is String .

Context <column>

Sample DOML file

The following snippet shows content of a DOML file, **sample.doml**, which creates tables containing data about cars, car dealers, and car owners.

```
<?xml version="1.0"encoding="UTF-8"?>
<doml>
   <database database="Standard" templateset=“standard”>
      <package id="sample">
         <table id="sample.Dealer">
            <column id="Name">
               <type dbType="VARCHAR"javaType="String"/>
            </column>
         </table>
         <table id="sample.Owner">
            <column id="Name">
               <type dbType="VARCHAR"javaType="String"/>
            </column>
            <column id="Age">
               <type dbType="INTEGER"javaType="int"/>
            </column>
         </table>
         <table id="sample.Car">
            <column id="LicensePlate">
               <type dbType="CHAR"javaType="String"/>
            </column>
            <column id="Dealer">
               <referenceObject reference="sample.Dealer"/>
               <type dbType="none"javaType="sample.DealerDO"/>
            </column>
```

```
        </table>
        <table id="sample.CarOwner">
          <column id="Car">
            <referenceObject reference="sample.Car"/>
            <type dbType="none"javaType="sample.CarDO"/>
          </column>
          <column id="Owner">
            <referenceObject reference="sample.Owner"/>
            <type dbType="none"javaType="sample.OwnerDO"/>
          </column>
          <column id="IsCurrent">
            <type dbType="BIT"javaType="boolean"/>
          </column>
          <index id="index_1" unique="true">
            <indexColumn id="Car"/>
          </index>
        </table>
      </package>
    </database>
  </doml>
```

Sample of part of DOML file for using indexes

The following snippet shows part of a DOML file, Computers.doml (creates tables containing data about computers and their parts.

```
   <table id="firm.computers.hardware.computers" notUsingOid="true" dbTableName="Computers">
     <column id="computerName" isPrimaryKey="true">
      <type dbType="CHAR" javaType="String" size="35"/>
     </column>
     <column id="compOwnerFirstName">
       <referenceObject constraint="true" reference="firm.general.employee.employee"
           foreignKeyColumn="employeeFirstName" foreignKeyGroup="employee"/>
       <type canBeNull="false" dbType="CHAR" javaType="String" size="40"/>
     </column>
     <column id="compOwnerLastName">
       <referenceObject constraint="true" reference="firm.general.employee.employee"
           foreignKeyColumn="employeeLastName" foreignKeyGroup="employee"/>
       <type dbType="CHAR" javaType="String" size="40"/>
     </column>
     <column id="division">
       <type dbType="CHAR" javaType="String" size="20"/>
     </column>
<!-- There is only one clustered index per table. computerName, compOwnerFirstName and compOwnerLastName
are  unique because one computer could has only one owner. At the begining you have to create tables and
columns,  after that indexes and after that primary keys. -->
     <index id="employeeName" clustered="true" unique="true">
       <indexColumn id="computerName"/>
       <indexColumn id="compOwnerFirstName"/>
       <indexColumn id="compOwnerLastName"/>
     </index>
    </table>
    ....
    <table id="firm.computers.hardware.parts.processor" notUsingOid="true" dbTableName="Processor">
      <column id="manufacturrer" isPrimaryKey="true">
        <type canBeNull="false" dbType="CHAR" javaType="String" size="40"/>
      </column>
      <column id="type" isPrimaryKey="true">
        <type canBeNull="false" dbType="CHAR" javaType="String" size="40"/>
      </column>
      <column id="speed" isPrimaryKey="true">
        <type canBeNull="false" dbType="DECIMAL" javaType="java.math.BigDecimal" size="9,2"/>
      </column>
      <column id="cache">
        <type canBeNull="true" dbType="DECIMAL" javaType="java.math.BigDecimal" size="9,2"/>
      </column>
      <column id="compName">
        <referenceObject constraint="true" reference="firm.computers.hardware.computers"
           foreignKeyColumn="computerName"/>
        <type dbType="CHAR" javaType="String" size="35"/>
      </column>
<!-- Each computer has only one processor. (There are all common computers.) -->
      <index id="computerName" unique="true">
```

```
        <indexColumn id="compName"/>
      </index>
    </table>
```

The whole DOML file **Computers.doml** in *<dods_home>/examples/doml_examples* directory.

Starting dods generator

There are two different ways to run dods generator. If you want to start generator quickly, you can start wizard by typing:

dods

without any parameter. Those files are located in :

- *<enhydra_home>/bin* folder, for DODS in Enhydra.
- *<dods_home>/bin* folder, for independent DODS.

Note: *<enhydra_home>/bin* (in the case DODS is used in Enhydra), or *<dods_home>/bin* folder (for independent DODS) should be added in the system path. Then, DODS can be started from any directory (by typing **dods**).

This will be described in the **Quick Compile** section.

If you want to start generator without wizard, you need to type (in the command line) **dods** with additional parameters. You can find details in **Custom Compile** section.

File location

After generating, locations of generated files are:

```
<OUTPUT_DIRECTORY>\SQLcreate.sql
<OUTPUT_DIRECTORY>\<PACKAGE_0>\..\<PACKAGE_N>\<TableName>DataStruct.java
<OUTPUT_DIRECTORY>\<PACKAGE_0>\..\<PACKAGE_N>\<TableName>DOI.java
<OUTPUT_DIRECTORY>\<PACKAGE_0>\..\<PACKAGE_N>\<TableName>DO.java
<OUTPUT_DIRECTORY>\<PACKAGE_0>\..\<PACKAGE_N>\<TableName>Query.java
<OUTPUT_DIRECTORY>\<PACKAGE_0>\..\<PACKAGE_N>\<TableName>.xml
```

where <OUTPUT_DIRECTORY> is base directory of your project, <PACKAGE_0>\..\<PACKAGE_N> is generated from last package id attribute of DOML file, and <TableName> is the name of the table from your database. For example, if part of your DOML file looks like this:

```
<package id="discRack">
  <package id="discRack.data">
    <table id=" discRack.data.Person" notUsingOId="true" dbTableName="Person">
                    ...
    </table>
```

you will get file structure as follows:

```
<OUTPUT_DIRECTORY>\discRack\data\PersonDataStruct.java
<OUTPUT_DIRECTORY>\discRack\data\PersonDOI.java
<OUTPUT_DIRECTORY>\discRack\data\PersonDO.java
<OUTPUT_DIRECTORY>\discRack\data\PersonQuery.java
<OUTPUT_DIRECTORY>\discRack\data\Person.xml
```

There are transient XML files that are generated from DOML file, before Java code is generated. The java code, mentioned before, is actually generated from those transient xml files. If you want, you can change these xml files instead of DOML file and generate Java code directly, without using the DOML file. You can find instructions for this in **Advanced Custom Compile** section.

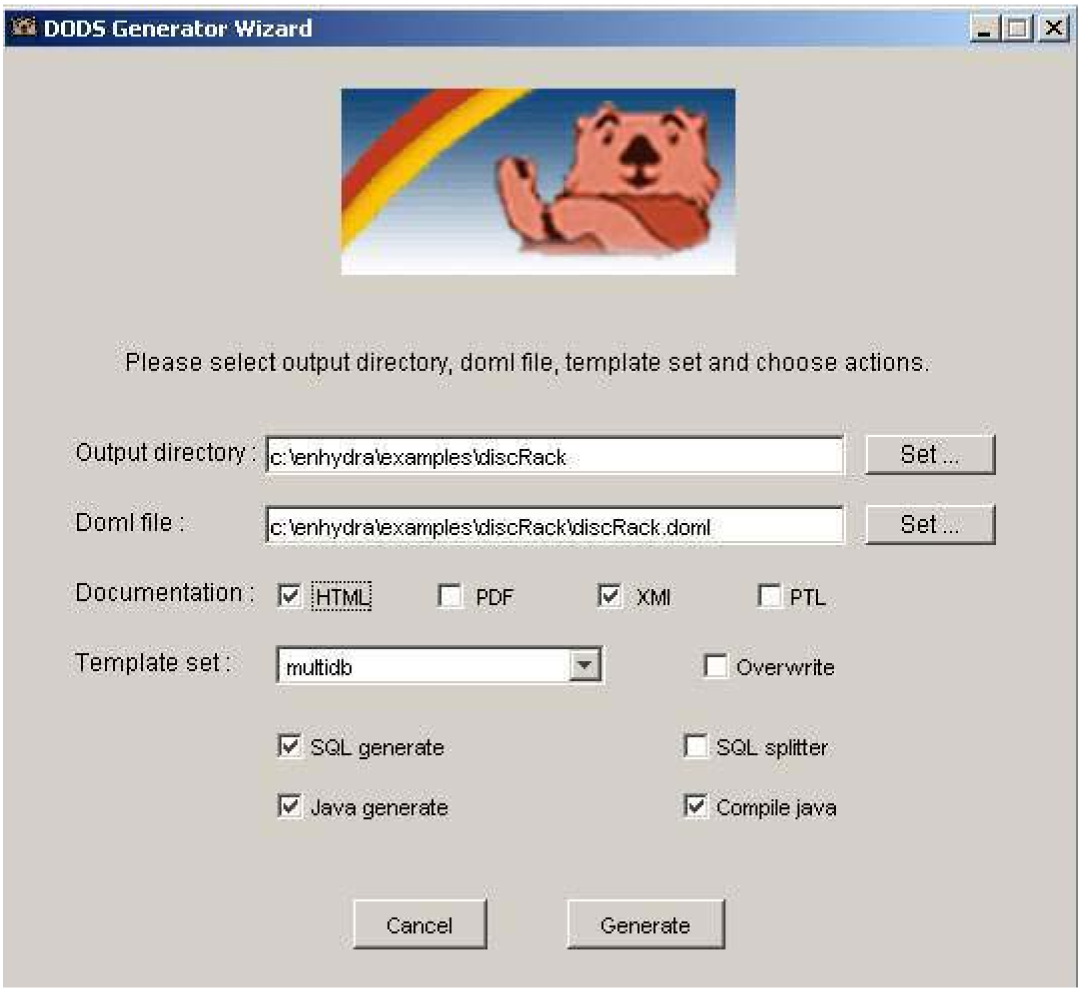
If you change the DOML file, all java classes will be generated again, but, if you change transient xml files instead of the DOML file, only changed xml files are generated in java files. Other java files (whose xml files are not changed) are left as they are.

Quick Compile

DODS Generator Wizard is a graphical tool that helps you to easily generate Java and SQL files. It is recommended for the first time users.

When you start **dods,** you will get window like on Figure 4.12.

Figure 4.12 DODS Generator Wizard



Please select output directory, doml file, template set and choose actions.

Output directory : c:\enhydra\examples\discRack

Doml file : c:\enhydra\examples\discRack\discRack.doml

Documentation : ☑ HTML ☐ PDF ☑ XMI ☐ PTL

Template set : multidb

☐ Overwrite

☑ SQL generate

☐ SQL splitter

☑ Java generate

☑ Compile java

In the **Output directory** field you should input directory with full path of output directory that will be used. **DOML file** field should be used for entering your DOML file.

There are four options on the Generator Wizard:

- **Generate SQL:**

  This field should be checked if you want to generate: SQL files for each table separately, one cumulutave SQL file for creating all tables (SQLcreate.sql), and one file for deleting those tables (SQLdrop.sql).

    - **SQL Splitter:**

      It is used for creating separated cumulative SQL files (for creating tables, for adding foreign keys, primary keys and for deleting tables). This option enables creating tables without cross references, and after their creation, adding needed references.

      SQL Splitter copies all SQL commands from all SQL files which are situated in the working directory and all its subdirectories into SQL files.

      Original SQL files are created by DODS - Enhydra.

      All SQL commands are copied into file **separateCreate.sql** except sql commands which reference to foreign and primary key columns.

      In the **separateIntegrity.sql** file class puts ALTER TABLE sql commands with adding foreign key references.

      In the **separatePrimary.sql** file class puts ALTER TABLE sql commands with adding primary keys.

      In the **separateDropTable.sql** file class puts DROP TABLE sql commands for all tables which were created by create table SQL statements in the first file (separateCreate.sql).

      In the **separateIndex.sql** file class puts CREATE INDEX sql commands for all tables which were created by create table SQL statements in the first file(separateCreate.sql).

      In the **separateDropIntegrity.sql** file class puts DROP foreign key sql commands for all tables which were created by create table SQL statements in the first file (separateCreate.sql).

      In the **separateDropPrimary.sql** file class puts DROP primary sql commands for all tables which created by create table SQL statements in the first file (separateCreate.sql).

      All others Sql commands class puts into separate file.

    Unless Generate SQL field is checked, this field can not be checked. If this option is checked, Generator Wizard doesn't create cumulative SQL files.

- **Generate Java:**

  This field should be checked if you want to generate Java files (DO, Query, DOI and DataStruct objects).

    - **Compile Java:**

      It is used for compiling generated java files. Compiled files will be located in folder *<output_directory>/classes*. Unless Generate Java field is checked, this field can not be checked.

  If you do not need both Java and SQL generation, you can choose one of them instead of both. At least one of the these options must be checked.

There is one combo box on the Generator Wizard. It contains following template sets:

- **standard**

  If this template set is chosen, DODS generates standard code.

- **multidb**

  If this template set is chosen, beside standard code, DODS also generates extra code that provides possibility of working with multi databases. Standard code without this extra code provides possibility of working with one database. You can see description of these new methods that work with multi databases in generated API.

- **<user_defined_templates>:**

  Users can define their own tempate sets.

Selected template set depends on <template_set> tag in doml file. If this tag is not set, defaut template set is "standard". If this tag is set, the value of this tag will be selected in template set combo box.

There is a possibility on the Generator Wizard for generating four types of documentation:

- **HTML:**

  If you check this field, doml file will be converted into html file.

- **PDF:**

  If you check this field, doml file will be converted into pdf file.

- **XMI:**

  If you check this field, doml file will be converted into xmi file.

- **PTL:**

  If you check this field, doml file will be converted into ptl (Rational Rose) file.

On the Generator Wizard, there is also a check box:

- **overwrite**

for code generating (java and sql), no matter if the code already existed.

Custom Compile

In case you want to generate Java and SQL code manually, type dods in the command line with desired parameters.

Command line:

```
        dods  [-?/help] [-a action] [-t templateset] [-b/-database] [-f/force]
              [-h/html] [-p/pdf] [-x/xmi] [-r/ptl] domlfile outputdir
```

where:

- **outputdir** is full path to output directory that will be used.
- **domlfile** is full path to .doml file for generating code.

options:

[**-? -help**] shows help.

[**-a action**] - ant task parameter for code generation:

- **dods:build_all** - to create all sql files and java classes (default).
- **dods:sql** - to create only sql files.
- **dods:java** - to create only java files and to compile them.
- **dods:javaNoCompile** - to create only java files and not to compile them.
- **dods:noCompile** - to create SQL files and java files and not to compile them.
- **dods:build_all_split** - to create all sql files and java classes and to compile it. SQL files will be divided into separate files using SQLSplitter .
- **dods:sqlsplit** - to create only sql files and separate in different files using SQLSplitter.
- **dods:noCompileSplit** - to create SQL files and separate sql commands using SQLSplitter and java files and not to compile them.

[**-t templateset**] - template set for generating java and sql code:

- **standard** - generate standard java code (default).
- **multidb** - generate java code with multi database support.
- **<user defined>** - any user defined template set.

[-b/-database] - sets database vendor for generating sql

[**-f/-force**] - with this switch, code will be always generated, without it, only changes will be regenerated.

[**-h/-html**] - generates DODS html documentation from .doml file.

[**-p/-pdf**] - generates DODS pdf documentation from .doml file.

[**-x/-xmi**] - generates DODS xmi documentation from .doml file.

[**-r/-ptl**] - generates DODS ptl (Rational Rose) documentation from .doml file.

Advanced Custom Compile

In this section you can find information about advanced settings for generation of Java files.

One XML file is generated for every table from DOML file (situated in table folder with other java and sql files). That XML file is used as a base for generating four Java files.

DTD for that file can be found in **<dods_home>/dtd/temporaryXML.dtd** file. Some tags could be changed, i.e. <AUTHOR>.

Important: some tags should not be changed, or otherwise generated code will not be compailable.

Structure

The hierarchy of tags in a XML file is, as follows:

```
<TABLE>
        <PACKAGE>
        <AUTHOR>
        <PROJECT_NAME>
        <TABLE_NAME>
        <CLASS_NAME>
        <DB_VENDOR>
        <TEMPLATE_SET>
        <DO_IS_OID_BASED>
        <IS_ABSTRACT>
        <IS_LAZY_LOADING>
        <DELETE_CASCADES>
        <COLUMN>
                <REFERENCE_OBJECT>
                        <CONSTRAINT>
                        <IS_ABSTRACT>
                        <IS_FOREIGN_KEY>
                        <PACKAGE>
                </REFERENCE_OBJECT>
                <IS_CONSTANT>
                <JAVADOC>
                <DB_TYPE>
                <JAVA_TYPE>
                <JAVA_DEFAULT_VALUE>
                <USED_FOR_QUERY>
                <CAN_BE_NULL>
                <IS_PRIMARY_KEY>
                <IS_FOREIGN_KEY>
                <SIZE>
        </COLUMN>
        <REFERRER>
                <REFATTR/>
        </REFERRER>
        <INDEX>
                <INDEX_COLUMN/>
        </INDEX>
</TABLE>
```

Tag reference

This section contains an alphabetical reference of all the XML tags that DODS can generate using given DOML file. Every tag contains the subsections:

- **Content** - tags that the tag can contain.
- **Attributes** - attributes the tag can have.
- **Context** - tags within which the tag can appear, in other words, the tags that can contain it.

<author>

Author of the Java code (your name).

Content	None
Attributes	None
Context	<table>

<can_be_null>

Can column be null. Possible values for can_be_null are:

- true
- false

Content None

Attributes None

Context <column>

<class_name>

The name of the class which represents table in the database, mostly, it is the TABLE_NAME.

Content None

Attributes None

Context <table>

<column>

Represents one column in the table.

Content <reference_object>

<constraint>

<is_abstract>

<is_foreign_key>

<package>

<is_constant>

<javadoc>

<db_type>

<java_type>

<java_default_value>

<used_for_query>

<can_be_null>

<is_primary_key>

<is_foreign_key>

<size>

Attributes	**name** Name of the column in the table.
Context	<table>

<constraint>

Specifies whether the specified table row must exist. Possible values for constraint are:

- true
- false

Content	None
Attributes	None
Context	<reference_object>

<db_type>

Data type from database that represents column.

Content	None
Attributes	None
Context	<column>

<db_vendor>

The database type. Possible values are Standard, InstantDB, Oracle, Informix, MySQL, Sybase or PostgreSQL.

Content	None
Attributes	None
Context	<table>

<delete_cascades>

Is it possible to delete cascades in database. Possible values for delete_cascades are:

- true
- false

Content	None
Attributes	None

Context <table>

<do_is_oid_based>

Is table based on OID keys. Possible values for do_is_oid_based are:

- true
- false

Content None

Attributes None

Context <table>

<index>

Represents table index.

Content <index_column>

Attributes **id**

Id of index.

unique

True if index is unique, otherwise false.

clustered

True if index is clustered, otherwise false.

Context <table>

<index_column>

Identifies index column.

Content None.

Attributes **id**

Id of index column, same as name of column.

Context <table>

<is_abstract>

Is generated class abstract. Possible values for is_abstract are:

- true
- false

Content	None
Attributes	None
Context	<table> <reference_object>

<is_constant>

Does column have constant value, that is, does it represent constant class attribute (not taken from database). Possible values for is_constant are:

- true
- false

Content	None
Attributes	None
Context	<column>

<is_foreign_key>

Is column used as a foreign key. Possible values for is_foreign_key are:

- true
- false

Content	None
Attributes	None
Context	<column> <reference_object>

<is_lazy_loading>

This attribute specifies whether the DO will use lazy loading. If the DO uses lazy loading, when you supply a known ObjectId to create a DO instance, the DO instance is created but the corresponding row in the table is not retrieved until the first get()or set() method call is made. It delays the hit on the database until the moment the data is actually needed. Possible values for isLazyLoading are:

- true
- false

Content	None
Attributes	None
Context	<table>

<is_primary_key>

Is column a primary key. Possible values for *is_primary_key* are:
- true
- false

Content	None
Attributes	None
Context	<column>

<javadoc>

Text for Javadoc documentation.

Content	None
Attributes	None
Context	<column>

<java_default_value>

Default value for Java data type.

Content	None
Attributes	None
Context	<column>

<java_type>

Data type from Java that represents column.

Content	None
Attributes	None
Context	<column>

<package>

Package that contains Java files.

Content	None
Attributes	None
Context	<table> <reference_object> <referrer>.

<project_name>

The project name.

Content	None
Attributes	None
Context	<table>

<refattr>

Tag that is used like attribute for tag <referrer>. It represents column of table that references generated class.

Content	None.
Attributes	**name** Name of the column that references some DO objects. It is object of generated class, mostly. **do_name** Name of the DO object that is referenced by attribute.
Context	<table>

<reference_object>

If the column is a reference to another table, <reference_object> specifies the table.

Content <constraint>

<is_abstract>

<is_foreign_key>

<package>

Attributes **name**

Name of the reference object class.

Context <column>

<referrer>

Outer table that references generated class.

Content <refattr>

Attributes **name**

Name of the outer table that references generated class.

package

Name of the outer table package that references generated class.

Context <table>

<size>

Specifies the size of data types that are commonly measured in width, like VARCHAR. size must be an integer.

Content None

Attributes None

Context <column>

<table>

Root element of XML files. It contains one table from database.

Content <package>

<author>

	<project_name>
	<table_name>
	<class_name>
	<db_vendor>
	<template_set>
	<do_is_oid_based>
	<is_abstract>
	<is_lazy_loading>
	<delete_cascades>
	<column>
	<referrer>
Attributes	None
Context	None

<table_name>

The name of the table in the database.

Content	None
Attributes	None
Context	<table>

<template_set>

Template set that will be used for java code generation. The possible values for template_set are:

- standard (default)
- multidb
- <any user defined template>

Content	None
Attributes	None
Context	<table>

<used_for_query>

Should column be used for queries. Possible values for used_for_query are:

- true
- false

Content None

Attributes None

Context <column>

Example

This in en example of how the tags mentioned before can be used. Transient XML can look like this:

```
<TABLE>(i)
        <PACKAGE>discRack/data/disc</PACKAGE>
        <AUTHOR>NN</AUTHOR>(2)
        <PROJECT_NAME>discRack</PROJECT_NAME>(2)
        <TABLE_NAME>Disc</TABLE_NAME>
        <CLASS_NAME>Disc</CLASS_NAME>
        <DB_VENDOR>Standard</DB_VENDOR>(2)
        <TEMPLATE_SET>standard</TEMPLATE_SET>4)
        <DO_IS_OID_BASED>true</DO_IS_OID_BASED>
        <IS_ABSTRACT>false</IS_ABSTRACT>
        <IS_LAZY_LOADING>true</IS_LAZY_LOADING>
        <DELETE_CASCADES>false</DELETE_CASCADES>

        <COLUMN name="owner">
                <REFERENCE_OBJECT name="Person">(3)
                        <CONSTRAINT>true</CONSTRAINT>(3)
                        <IS_ABSTRACT>false</IS_ABSTRACT>(3)
                        <IS_FOREIGN_KEY>false</IS_FOREIGN_KEY>(3)
                        <PACKAGE>discRack.data.person</PACKAGE>(3)
                </REFERENCE_OBJECT>(3)
                <IS_CONSTANT>false</IS_CONSTANT>
                <JAVADOC></JAVADOC>(4)
                <DB_TYPE>none</DB_TYPE>
                <JAVA_TYPE>DiscRack.data.person.PersonDO</JAVA_TYPE>
                <JAVA_DEFAULT_VALUE></JAVA_DEFAULT_VALUE>(4)
                <USED_FOR_QUERY>true</USED_FOR_QUERY>
                <CAN_BE_NULL>false</CAN_BE_NULL>
                <IS_PRIMARY_KEY>false</IS_PRIMARY_KEY>
                <IS_FOREIGN_KEY>true</IS_FOREIGN_KEY>
                <SIZE></SIZE>
        </COLUMN>

        <REFERRER name="Storage" package="discRack.data.storage">(3)
                <REFATTR name="disc" do_name="Disc"/>(3)
        </REFERRER>(3)
        <INDEX id="index_owner" unique="true" clustered="false">
                <INDEX_COLUMN id="owner"/>
        </INDEX>
</TABLE>(1)
```

(1) red; (2) green; (3) violet; (4) blue

Different line colors are used to describe tag existence and possibility to change tag values.

Red tags must exist and must NOT be changed.

Green tags must exist and can be changed.

Blue tags are not required.

Violet tags can exist and must NOT be changed.

Black tags must exist and can be changed.

DODS independence

Since this version, DODS is independent from Enhydra. This means that it is possible for user to make any application (it doesn't need to be enhydra application) that can use DODS.

DODS works with DatabaseManagers. DatabaseManager is class that provides facilities for work with databases.

There are two modes of using DODS:

- **non-threading**
  In non-threading mode, only one DatabaseManager is used for the whole application, no matter the application has one or more Threads.

- **threading**
  In threading mode, there is one DatabaseManager for every Thread. User needs, for every Thread, to define DatabaseManager. If, for any Thread, the DatabaseManager is not defined, the default DatabaseManager is used.

In the following text, the DODS independence is explaned for non-threading mode.

- in **main** application,
  add code that makes new DatabaseManager and registers it in DODS:

```
try {
.               . .
                String fileName = discRack.conf";
                        DatabaseManager dbManager =
                        StandardDatabaseManager.newInstance(fileName);
                DODS.register(dbManager);
                        . . .
} catch (Exception e) {
                e.printStackTrace();
}
```

where "discRack.conf" is an example of application's configuration file. This file is the same as the Database Manager section of Enhydra application's configuration file.

This file can look like this:

```
#
# The databases that are used by CSAM.  Each of these databases
# has configuration parameters set under DatabaseManager.DB."databaseName".
#
DatabaseManager.Databases[] = "sid1"
#
# The default database used in this application.
#
DatabaseManager.DefaultDatabase = "sid1"
#
# Turn on/off debugging for transactions or queries. Valid values
# are "true" or "false".
#
DatabaseManager.Debug = "false"
#
# The type of database. Normally this is "Standard".
#
DatabaseManager.DB.sid1.ClassType = "Standard"
# DatabaseManager.DB.sid1.ClassType = "Oracle"


#
# The jdbc driver to use.
#
DatabaseManager.DB.sid1.JdbcDriver = "org.enhydra.instantdb.jdbc.idbDriver"
# DatabaseManager.DB.sid1.JdbcDriver = "oracle.jdbc.driver.OracleDriver"
# DatabaseManager.DB.sid1.JdbcDriver = "sun.jdbc.odbc.JdbcOdbcDriver"

#
# Database url.
#
DatabaseManager.DB.sid1.Connection.Url = "jdbc:idb:@OUTPUT@/discRack.prp"
# DatabaseManager.DB.sid1.Connection.Url = "jdbc:oracle:thin:@MyHost:MyPort:MyDBName"
# DatabaseManager.DB.sid1.Connection.Url = "jdbc:odbc:discRack"

#
# Database user name.  All connection are allocated by this user.
#
DatabaseManager.DB.sid1.Connection.User = "scott"
#DatabaseManager.DB.sid1.Connection.User = "Admin"

# Database user password.
#
DatabaseManager.DB.sid1.Connection.Password = "tiger"
#DatabaseManager.DB.sid1.Connection.Password = ""

#
# The maximum number of connections that a connection
# pool will hold.  If set to zero, then connections
# are allocated indefinitly or until the database
# refuses to allocate any new connections.
#
DatabaseManager.DB.sid1.Connection.MaxPoolSize = 30

#
# Maximum amount of time that a thread will wait for
# a connection from the connection pool before an
# exception is thrown.  This will prevent possible dead
# locks.  The time out is in milliseconds.  If the
# time out is <= zero, the allocation of connections
# will wait indefinitely.
#
DatabaseManager.DB.sid1.Connection.AllocationTimeout = 10000

#
# Used to log database (SQL) activity.
#
DatabaseManager.DB.sid1.Connection.Logging = false

#
# The number of object identifiers that are allocated
# as a group and held in memory.  These identifiers
# are assigned to new data objects that are inserted
# into the database.
#
DatabaseManager.DB.sid1.ObjectId.CacheSize = 20
DatabaseManager.DB.sid1.ObjectId.MinValue = 1000000

#
# User wildcards
#
DatabaseManager.DB.User.userWildcard = "*"
DatabaseManager.DB.User.userSingleWildcard = "_"
DatabaseManager.DB.User.userSingleWildcardEscape = "§"
DatabaseManager.DB.User.userWildcardEscape = "§"

#
```

```
# Cache configuration
#
DatabaseManager.DB.Cache.defaultMaxCacheSize = 5000
DatabaseManager.DB.Cache.Person.maxCacheSize = -50
DatabaseManager.DB.Cache.Disc.maxCacheSize = 3000
DatabaseManager.DB.Cache.Person.readOnlyCache = true
DatabaseManager.DB.Cache.Disc.readOnlyCache = false
DatabaseManager.DB.Cache.Person.defaultMaxSimpleQueryCacheSize = 1000
DatabaseManager.DB.Cache.Person.defaultMaxComplexQueryCacheSize = 200
DatabaseManager.DB.Cache.Person.maxSimpleQueryCacheSize = 150
DatabaseManager.DB.Cache.Person.maxComplexQueryCacheSize = 70
DatabaseManager.DB.Cache.Disc.defaultMaxSimpleQueryCacheSize = 1000
DatabaseManager.DB.Cache.Disc.defaultMaxComplexQueryCacheSize = 200
DatabaseManager.DB.Cache.Disc.maxSimpleQueryCacheSize = 900
DatabaseManager.DB.Cache.Disc.maxComplexQueryCacheSize = 200
DatabaseManager.DB.Cache.Person.initalConditionForCache = "*"
```

The example of non-enhydra application that can use DODS is DiscRack application, explained in next section.

Examples of non-enhydra applications

Examples of DODS non-enhydra applications are included in DODS installation and they are in DODS, in directory:

<DODS_HOME>/examples

Process of running non-enhydra application will be presented in this section on the example Disc Rack. This example application is in *<DODS_HOME>/examples/discrack* directory.

To run example , these steps have to be done in Command Promt:

- first, go to wanted example (directory)

  ```
  cd <DODS_HOME>/examples /discrack
  ```

- second, run ant,by typing:

  ```
  ant
  ```

  ant will build this application in its <output_directory>

- then, go to application's output directory:

  ```
  cd <output_directory>
  ```

- then, example will be run with:

  ```
  run
  ```

The ant, which is used here, must be DODS's ant.bat, which means that path *<DODS_HOME>/bin* must be included at the beginning of the system path.

Caching

Caching affects the behaviour of the DO class. If checked, all DO instances are stored in the cache inside the DO class. Subsequent queries of the table use the Query class for queries. The results of all Queries, are complete.

When you insert new DO into the database, the DO is also (automatically) inserted into the cache.

When you delete DO from the database, the DO is also (automatically) deleted from the cache.

The possible cache types are:

- **None**
  This flag means that there is no caching available.

- **LRU**
  The size of the cache is limited by the maximal number of data objects that can be stored in. When the cache is full, the DOs in it are being replaced by new data objects according to LRU (least recently used) algorithm. This algorithm says that the DO (data object) which had been used the least recently (in the scale of time, the DO to which had been accessed the longest time ago) is replaced with the new data object.

- **Full** (special case of LRU caching)
  This is LRU cache which is unbounded (maximal number of data objects has no limit). The entire table is queried and cached when your application starts. This is appropriate for tables of "static" data which is accessed frequently and which will not change during the execution of your application. In this case, all Queries search the cache, hitting the database is never performed (because all database DOs are stored in the cache).

Default value of caching is "none".

Since this version, beside cache of data object, caching queries is also enabled. Caching now looks like this:

Select clause
- For query by oid, first is checked in the DO cache (cache of data objects) if there is data object with desired oid. If data object is not find in the cache, hitting the database is performed.
- For full caching, for query by oid, first is also checked in the DO cache if there is data object with desired oid. If data object is not find in the cache, hitting the database is not performed (all data objects are in the DO cache, so there is no result of this query).
- For all other queries, first is checked whether the query is already in the QueryCache (cache of queries)
- If query is in the cache, the oids of the results are retrieved from QueryCache
- For every result oid, it is checked whether there is that DO in the cache
- The time needed for performing queries by oid on database for all oids from result that are not in the cache is compared against the time needed for performing the whole query.
- If the time needed for performing queries by oid on database is less, data objects are retrieved from the cache, or, if they are not there, from database (using queries by oid).
- If the time is bigger, or the query is not in the query cache, the query is performed on database. The results are retreved from database. The query and oids of result data objects are put in the QueryCache

Insert clause
- DO is inserted in the database
- DO is added in the DO cache
- All complex queries are removed from QueryCache
- For every simple query from QueryCache it is checked whether inserted DO is the result and if yes, the DO is included in query results

Delete clause
- Delete DO from the database
- Remove DO from the DO cache (if it is there)
- Go through QueryCache and wherever find this DO, remove it from query results

Update clause

- Update DO in the database
- Update DO in the DO cache (if it is there)
- All complex queries are removed from QueryCache
- For every simple query from QueryCache it is checked whether updated DO is the query result
- If yes, the DO is included in query results (if DO is not already in the query results, add it)
- If no, if DO already exists in the query results, it is remove from there

Cache configuration

In application's configuration file, there are following (optional) information about cache:

- default maximal DO cache size
- default maximal simple query cache size
- default maximal complex query cache size
- default read-only
- maximal DO cache size (for every cached table)
- maximal simple query cache size (for every cached table)
- maximal complex query cache size (for every cached table)
- read-only value for every cached table
- initial “where” statement (with witch the cache will be initialized)

Example:
For file **discRack.conf** with type of caching **LRU**, part of code for cache configuration can look like this:

```
#
# Cache configuration
#
DatabaseManager.DB.Cache.defaultMaxCacheSize = 5000
DatabaseManager.DB.Cache.Person.maxCacheSize = -50
DatabaseManager.DB.Cache.Disc.maxCacheSize = 3000
DatabaseManager.DB.Cache.Person.readOnlyCache = true
DatabaseManager.DB.Cache.Disc.readOnlyCache = false
DatabaseManager.DB.Cache.Person.defaultMaxSimpleQueryCacheSize = 1000
DatabaseManager.DB.Cache.Person.defaultMaxComplexQueryCacheSize = 200
DatabaseManager.DB.Cache.Person.maxSimpleQueryCacheSize = 150
DatabaseManager.DB.Cache.Person.maxComplexQueryCacheSize = 70
DatabaseManager.DB.Cache.Disc.defaultMaxSimpleQueryCacheSize = 1000
DatabaseManager.DB.Cache.Disc.defaultMaxComplexQueryCacheSize = 200
DatabaseManager.DB.Cache.Disc.maxSimpleQueryCacheSize = 900
DatabaseManager.DB.Cache.Disc.maxComplexQueryCacheSize = 200
DatabaseManager.DB.Cache.Person.initalConditionForCache = "*"
```

In the following text are explained maximal cache sizes (for DO cache and query caches). The parameters maxCacheSize, maxSimpleQueryCacheSize and maxComplexQueryCacheSize of application's configuration file define these sizes.

1. maxCacheSize > 0

   This cache is limited. The maximal number of elements in the cache is maxCacheSize.

2. maxCacheSize = 0

   This means that there is no cache available. This value excludes cache from use.

3. maxCacheSize < 0

This cache is unlimited.

If any of parameters **maxCacheSize**, **maxSimpleQueryCacheSize** or **maxComplexQueryCacheSize** are not defined in configuration file, the size of the cache is defined by parameter **defaultMaxCacheSize** (or **defaultMaxSimpleQueryCacheSize** or **defaultMaxComplexQueryCacheSize**) of the same configuration file. The rules for these values are the same as for **maxCacheSize**, **maxSimpleQueryCacheSize** and maxComplexQueryCacheSize explained before.

If neither parameter maxCacheSize nor parameter defaultMaxCacheSize are defined, DODS has its own constants:

- **DEFAULT_MAX_CACHE_SIZE**
- **DEFAULT_MAX_SIMPLE_QUERY_CACHE_SIZE** and
- **DEFAULT_MAX_COMPLEX_QUERY_CACHE_SIZE**

that define maximal CACHE sizes.

The read-only value tells whether the cache is read-only or not. The parameter **readOnlyCache** of application's configuration file defines read-only value.If it is read-only, the operations: insert, update or delete on the database are not possible. The default DODS value is false.

In the following text is explained initial "where" statement. The parameter **initalConditionForCache** of application's configuration file defines it. This is "where" part of select clause with which the cache will be initialized.

1. initalConditionForCache = ”*”

   This means that the entire table will be added to the DO cache in DO cache inicialization.

2. initalConditionForCache = ”<where_clause>”

   This means that the DOs from table that satisfy <where_clause> will be added to the DO cache in DO cache inicialization.

If this parameter does not exist in the application's configuration file, by default, no data object will be added to cache during its inicialization.

In previous mentioned example for part of code for cache configuration, for table Person cache type is full, because maxCacheSize is negative, readOnlyCache is true and initalConditionForCache is "*". This combination of values of parameters form special cache of LRU: full cache.

User wildcards

Like cache size, user wildcards are also defined in application’s configuration file.

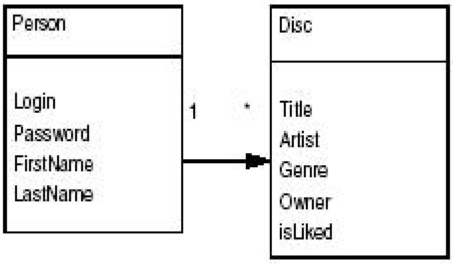
Example:
For file **discRack.conf** part of code for user wildcards can look like:

```
#
# User wildcards
#
DatabaseManager.DB.User.userWildcard = "*"
DatabaseManager.DB.User.userSingleWildcard = "_"
DatabaseManager.DB.User.userSingleWildcardEscape = "§"
DatabaseManager.DB.User.userWildcardEscape = "§"
```

Loading the schema

The picture 4.19 shows the schema of the Disc Rack example (the example is explained detailly in the chapter 5).

Figure 4.19 DiscRack object-model/schema



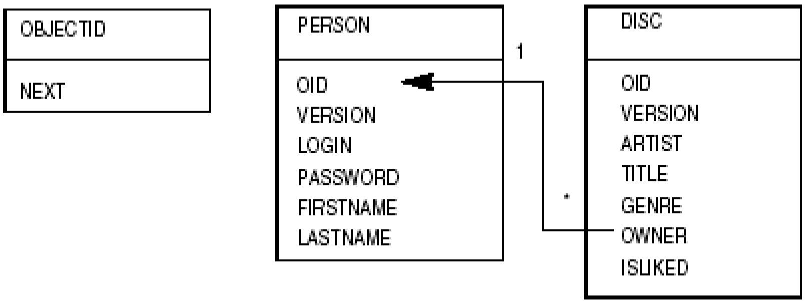
There are two tables:

- Person, which has columns: Login, Password, FirstName and LastName
- Disc, which has columns: Title, Artist, Genre, Owner and isLiked

The complete schema of the DiscRack database generated by DODS is pictured in Figure 4.20. 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 4.20 DiscRack database schema generated by DODS



There are some differences from the original database schema:

- 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 application is updating) is accurate. Because many users can be accessing the database simultaneously, a record can change between the time the application retrieves it and when the application 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.

Generated structure

DODS generates the following files and subdirectories in the **data** directory:

- **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 tables
- **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
- file **build.xml**
- **classes** directory, which is initially empty

Each data object directory contains Java source files to create four classes. For example, the person data object directory contains personDO, personDOI, personQuery, and personDataStruct. The data object and the query classes are the most commonly used classes.

DODS also generates build.xml file for the data layer. This lets you compile the data layer independently or along with the entire project. The empty **classes** directory is used only if you compile the data layer separately.

The next step in the process is to run the SQL script that DODS generated to create the tables in the database. You will do this in the next section.

Running the DODS-generated scripts

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

1 Open the **create_tables.sql** file in the <enhydra_root>/examples/discrack/discRack/data directory.

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

```
/* This SQL was generated for a Standard database. */
create table Person
(
/* class Person */
   login VARCHAR(32) DEFAULT "" NOT NULL ,
   password VARCHAR(32) DEFAULT "" NOT NULL ,
   firstname VARCHAR(32) DEFAULT "" NOT NULL ,
   lastname VARCHAR(32) DEFAULT "" NOT NULL ,
   oid DECIMAL(19,0) NOT NULL PRIMARY KEY,
   version INTEGER NOT NULL
);

/* This SQL was generated for a Standard database. */
create table Disc
(
/* class Disc */
   title VARCHAR(32) DEFAULT "" NOT NULL ,
   artist VARCHAR(32) DEFAULT "" NOT NULL ,
   genre VARCHAR(32) DEFAULT "" NOT NULL ,
   owner DECIMAL(19,0) NOT NULL REFERENCES Person ( oid ) ,
   isLiked INTEGER DEFAULT 0 NOT NULL ,
   oid DECIMAL(19,0) NOT NULL PRIMARY KEY,
   version INTEGER 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.

2 If necessary, edit **create_tables.sql** to work for your database.

The samples directory (<enhydra_root>/doc/getting_started/samples/) contains a **create_tables.sql** file that has been edited to work with InstantDB. The following changes were needed for the file work with InstantDB :

1 Add the following lines to the top of the file to load the JDBC driver and open the database:

```
d org.enhydra.instantdb.jdbc.idbDriver;
o jdbc:idb=<database_path>;
```

For <*database_path*> enter the location of your database properties file. For example, /enhydra/myapps/data/simpleApp.prp.

2 Put an e before each SQL statement. For example, create table Disc becomes e create table Disc.

3 Replace double quotes ("") with single quotes ('').

4 Add the following line at the end of the file to close the database: c close;

You can configure many things about the SQL that DODS generates in the configuration file <*dods_root*>*/conf/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.

3 Load the tables into the database.

For InstantDB, the command to load the SQL file using ScriptTool is:

```
java org.enhydra.instantdb.ScriptTool create_tables.sql
```

For Oracle, the command for SQL*Plus is:

```
SQL> @<simpleApp_root>/simpleApp/data/create_tables.sql
```

4 Add some dummy data to the database for testing purposes.

For InstantDB, use ScriptTool and the supplied tutorial_insert_idb.sql file (located in <*enhydra_root*>*/doc/getting_started/samples*) to add data to the database:

```
java org.enhydra.instantdb.ScriptTool tutorial_insert_idb.sql
```

For Oracle, use SQL*Plus and the supplied **tutorial_insert.sql**:

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

Note The supplied files **tutorial_insert_idb.sql** and **tutorial_insert.sql** contain some sample data, including one person and three discs.

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 object, SimpleDiscQuery.java, you created previously.

1 Replace **SimpleDiscList.java** in your application's business directory with **SimpleDiscList_DODS.java** from the *<enhydra_root>/doc/getting_started/samples* directory. Rename ithe file and call it SimpleDiscList.java.

2 Look at the code in the new SimpleDiscList object and compare it to the code in SimpleDiscList.java in the samples directory.

The main difference between the old and new objects is in the **getDiscList()** method. Here is the heart of the method from the new object:

```
...
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[i][0] = (String)discArray[i].getTitle();
      result[i][1] = (String)discArray[i].getArtist();
      result[i][2] = (String)discArray[i].getGenre();
      result[i][3] = discArray[i].getIsLiked() ? "Yes" : "No";
   }
}
return result;
...
```

The code in the new object 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 previous code only uses the getter methods getTitle(), getArtist(), getGenre(), and getIsLiked(). The getIsLiked() method returns a boolean value, while the other methods return a string. For the sake of consistency, the getIsLiked() method performs some simple logic to translate the boolean value to the appropriate string.

Running the enhydra application

Now we will recompile and run the simpleApp enhydra application.

1 Build the application from the top level, using the ant command. For example:

```
cd /enhydra/myapps/simpleApp
ant
```

2 Start Enhydra, either by starting the Admin Console (see "Launching the Admin Console") or by entering the following commands in the shell window (UNIX), or in command window (WINDOWS):

1. **UNIX**:

```
cd output
```

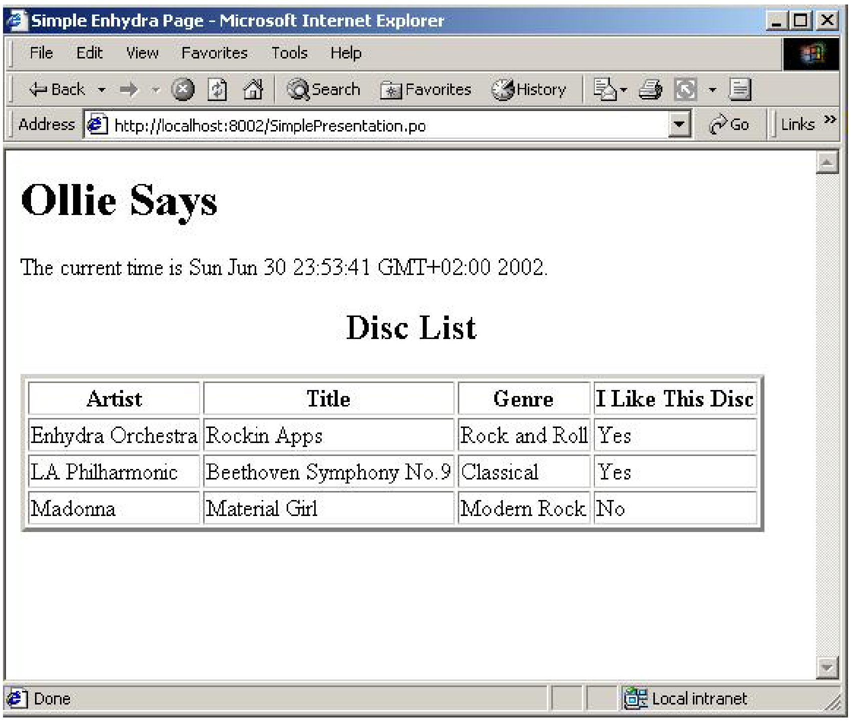
```
./run
```

2. **WINDOWS**:

```
cd output
run
```

3 Test the application by loading http://localhost:8002 in your browser. The Disc List table in your Simple presentation object should now contain data from the Disc table in your database.

Figure 4.21 Simple PO with data from DODS data objects


Simple Enhydra Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost:8002/SimplePresentation.po
Go Links
Done
Local intranet

Ollie Says

The current time is Sun Jun 30 23:53:41 GMT+02:00 2002.

Disc List

Artist	Title	Genre	I Like This Disc
Enhydra Orchestra	Rockin Apps	Rock and Roll	Yes
LA Philharmonic	Beethoven Symphony No.9	Classical	Yes
Madonna	Material Girl	Modern Rock	No

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

1 Look in the **simpleApp.conf** file in the output directory to make sure that the database settings are listed correctly.

2 Check the output displayed in the shell window (UNIX), or in command window (WINDOWS) for errors when you start the Multiserver. If the database settings are in simpleApp.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 For Oracle databases, 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 JVMs running. Sometimes, the class loader can fail to find the correct classes if it picks up an old CLASSPATH from a running **JVM.**

Database Independence

DODS generates java code that is database independent. This means that java code is the same no matter which base you use.

When you want to change the database, the only thing you need to do is to change <App_name>.conf file (update it with information considering new database). This change is necessary for connection to database.

Using multi databases in DODS

Enhydra has the possibility of working with more than one database at the same time. This means that, when the application is started, you don't have to stop it in order to change the database the application uses.

If you want to use this Enhydra option, you must (in <App_name>.conf file) define all logical databases you want to use. For each of these databases you must configurate all needed parameters. If you don't want to use this Enhydra option, the default database will be used.

When you update <App_name>.conf (with information about all databases) and start you application, it uses the default database. The definition of the new (desired) database is being done in the stage of creation of DO and Query objects.

When you create Query object for a database (given or default), the result of this Query are only DOs from that database, not from any other base.

If caching is used, there is only one cache for all <object_class>DOs (<object_class>DOs from all databases are placed in the same cache).

When you create DO for a specific database, this DO can't change database any more. If you want to translate one DO object from its database to another, you must create new DO in that another database, and then copy data of DO you want to translate into new DO (there are copy methods which you can use for this). In this way, new DO gets its own ID of its base.

Query object can change database. When you change database of the Query object, now the result of this Query will be DOs from this new database; you won't be able to get the DOs from the previous database any more.

DODS takes care of referential integrities within the database which means that DODS searches referenced object in the same database in which the object that referenced it is. If you want to use referenced objects from any other database, you must yourself take care of referential integrities.

In the <object_class>DO class public constructors and methods (loadData, createVirgin, createForExisting, createExisting) are now defined and with the parameter database. You can use them with this parameter in which case the object will be created for the given logical database, or you can use these constructors and methods without database parameter. In this case, they will be created for default database.

Tip: Be very careful when you add DO objects in the databases, or when you use Query objects. Now, there is more than one database that is used, and it's more difficult to track in which database, which DO object is placed, and for which base you run Query.

Tip: When you use multi databases with full caching, it would be better to announce in advance which databases will be used (with the method useLogicalDatabase(String database)) so that all needed DOs would be put in the cache at once, not partial.

Conversion of doml file

DODS has the possibility of converting doml file, release 5.*. As described in “Quick

Compile", Generator Wizard has a possibility of converting doml file into four document types: html, pdf, xmi and ptl. The name of the target (html, pdf, xmi, ptl) files will be the same as the name of doml file that is being converted, and they would be located in output directory.

The doml file can also be converted manually. For this purpose are used files that are in <dods _root>/bin folder, and they are:

- **doml2html** - converts doml 5.* file into html file.

  doml2html is used with the following parameters:

  ```
  doml2html [-help] [doml5*-file] [html-file]
  ```

  where:
  help - prints message for usage and exits.
  doml5*-file - doml file relese 5.*.
  html-file - desired target html file.

- **doml2pdf** - converts doml 5.* file into pdf file.

  doml2pdf is used with the following parameters:

  ```
  doml2pdf [-help] [doml5*-file] [pdf-file]
  ```

  where:
  help - prints message for usage and exits.
  doml5*-file - doml file relese 5.*.
  pdf-file - desired target pdf file.

- **doml2xmi** - converts doml 5.* file into xmi file.

  doml2xmi is used with the following parameters:

  ```
  doml2xmi [-help] [doml5*-file] [xmi-file]
  ```

  where:
  help - prints message for usage and exits.
  doml5*-file - doml file relese 5.*.
  xmi-file - desired target xmi file.

- **doml2ptl** - converts doml 5.* file into ptl file.

  doml2ptl is used with the following parameters:

  ```
  doml2ptl [-help] [doml5*-file] [ptl-file]
  ```

  where:
  help - prints message for usage and exits.
  doml5*-file - doml file relese 5.*.
  ptl-file - desired target ptl (Rational Rose) file.

- **doml31_2_doml51** - converting doml 3.1 file into doml 5.1 file.

  doml31_2_doml51 is used with the following parameters:

  ```
  doml31_2_doml51 [-help] [doml31-file] [doml51-file]
  ```

  where:
  help - prints message for usage and exits.
  doml31-file - doml file relese 3.1.
  doml51-file - desired target doml file relese 5.*.

Template sets

User can make its own template sets. All template sets (standard, multidb and users) are placed in one directory. The name and location of this directory can be set (or changed) in **dods.properties** file (in <dods _root> directory). Within this directory, every template set is placed in its own subdirectory.

Custom Configuration

To configure DODS, use **dodsConf.xml** file, located in <dods_root>/build/conf directory.

This file contains the following information:

- location of templates - tag <TemplateDir>

  example:

  ```
  <TemplateDir>C:/DODS/build/template</TemplateDir>
  ```

- for each database vendor, location of its configuration file (in xml format) - tag <Database>

  example:

  ```
  <Database>
      <Vendor name="Standard">C:/DODS/build/conf/StandardConf.xml</Vendor>
      <Vendor name="InstantDB">C:/DODS/build/conf/InstantDBConf.xml</Vendor>
      <Vendor name="Oracle">C:/DODS/build/conf/OracleConf.xml</Vendor>
      <Vendor name="Informix">C:/DODS/build/conf/InformixConf.xml</Vendor>
      <Vendor name="MSQL">C:/DODS/build/conf/MSQLConf.xml</Vendor>
      <Vendor name="Sybase">C:/DODS/build/conf/SybaseConf.xml</Vendor>
      <Vendor name="PostgreSQL">C:/DODS/build/conf/PostgreSQLConf.xml</Vendor>
      <Vendor name="HypersonicSQL">C:/DODS/build/conf/HypersonicSQLConf.xml</Vendor>
      <Vendor name="DB2">C:/DODS/build/conf/DB2Conf.xml</Vendor>
      <Vendor name="QED">C:/DODS/build/conf/QEDConf.xml</Vendor>
      <Vendor name="MySQL">C:/DODS/build/conf/MySQLConf.xml</Vendor>
  </Database>
  ```

Database Vendor's configuration file contains informatio about that database (type of ObjectId, column name for oid and version, information about DeleteCascade, constraints, quotes,comments, characterd for like and wildcard, mapping JDBC types to vendor-specific data types,...).

If tags <ClassPath>, <ClassName> are not mentioned, standard code is generated for that database vendor.

If database is specific, path to jar file for that database vendor is in tag <ClassPath>, and its main class is in tag <ClassName>.

example, for database Informix:

```
<ClassPath>C:/DODS/lib/dbvendors/informix.jar</ClassPath>
<ClassName>com.lutris.appserver.server.sql.informix.InformixLogicalDatabase</ClassName>
```

Chapter 5

DiscRack sample application

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

Building and running DiscRack

Enhydra includes the DiscRack application, which is installed in the <enhydra_root>/examples/DiscRack directory. Throughout this chapter, this top-level directory containing DiscRack is referred to as <DiscRack_root>. To build and run DiscRack, you need to do:

1 Open the application configuration template file, discRack.conf.in, in <DiscRack_root>/discRack and make sure all Database Manager configuration settings are correct

2 Build the application by entering the **ant** command from the <DiscRack_root> directory. Building the application will generate all the classes and packages for the DiscRack application.

Note The DiscRack database and corresponding application data layer are identical to the those created in the chapter 4, “Tutorial: Building Enhydra applications,” with the exception of package naming. This database schema is loaded for you by default, using an InstantDB database. Alternatively, you can use the Microsoft Access database in <DiscRack_root>/discRack/data/discRack.mdb, with the appropriate changes to the Database Manager configuration settings in the application cofiguration file. Refer to the appendix A, “Database configurations,” for additional information about using Microsoft Access with your application.

3 To use the default InstantDB database, copy the InstantDB JAR files, idb.jar and jta-spec1_0_1.jar, into the <DiscRack_root>/lib directory. The InstantDB drivers are loaded by the DiscRack run script.

4 To run DiscRack, enter the following commands:

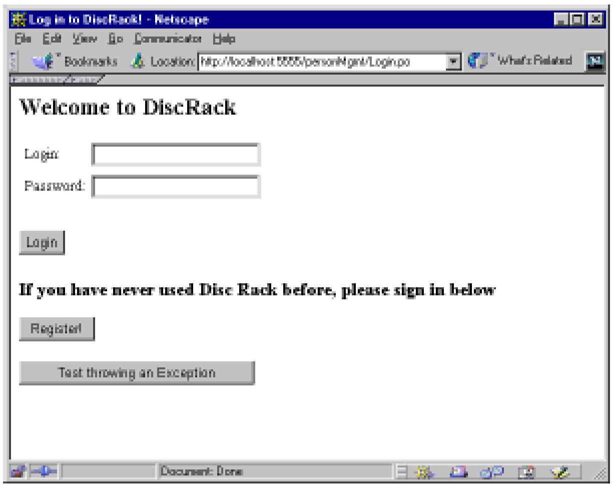
In UNIX:
```
cd <DiscRack_root>/output
./run
```

In WINDOWS:
```
cd <DiscRack_root>/output
run
```

5 To access the application, enter the URL http://Localhost:5555 in your browser location field.

Your browser displays the following screen:

Figure 5.1 Browser displaying the DiscRack Login presentation object



Welcome to DiscRack

If you have never used Disc Rack before, please sign in below

Log in to DiscRack! - Netscape
File Edit View Go Communicator Help
Bookmarks
Location: http://localhost:5555/personMgmt/Login.po
What's Related
Login:
Password:
Login
Register!
Test throwing an Exception
Document: Done

Play around with the application to get a sense for how it works:

- Click the Register button to add yourself as a user, then add some discs to your inventory.
- Try viewing your inventory and editing one of the discs.

Process and preliminaries for developing applications

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 helpful 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.

A simplified Enhydra application development process consists of these steps:

- Requirements definition

  As specifically as possible, create a statement of what the application is supposed to accomplish. This statement essentially defines the high-level goals of the application.

- Functional specification

  Outline how the application solves the problem(s) stated in the requirements definition.

- Design and storyboard

  Design the presentation, data, and business layers of the application, then create the storyboard.

- Development and testing

  Code and test the application.

- Deployment

  Pack the application and install it in its operational environment.

This abbreviated methodology illustrates 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. The following sections illustrate these abbreviated steps.

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 definition:

DiscRack will let each user keep track of his or her individual CD inventory by adding, modifying, and deleting 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

  To access their CD inventory, users must log in with a user name and password.

- Allow new users to sign up by entering their name, user name, and password.

- Once logged in, a user can see his or her CD inventory and:

  - Add new CDs to the inventory.

  - Edit existing CD entries.

  - Delete an existing entry, with a confirmation prompt.

- The information that will be displayed for each CD includes artist, title, genre, and whether or not the user likes the CD.

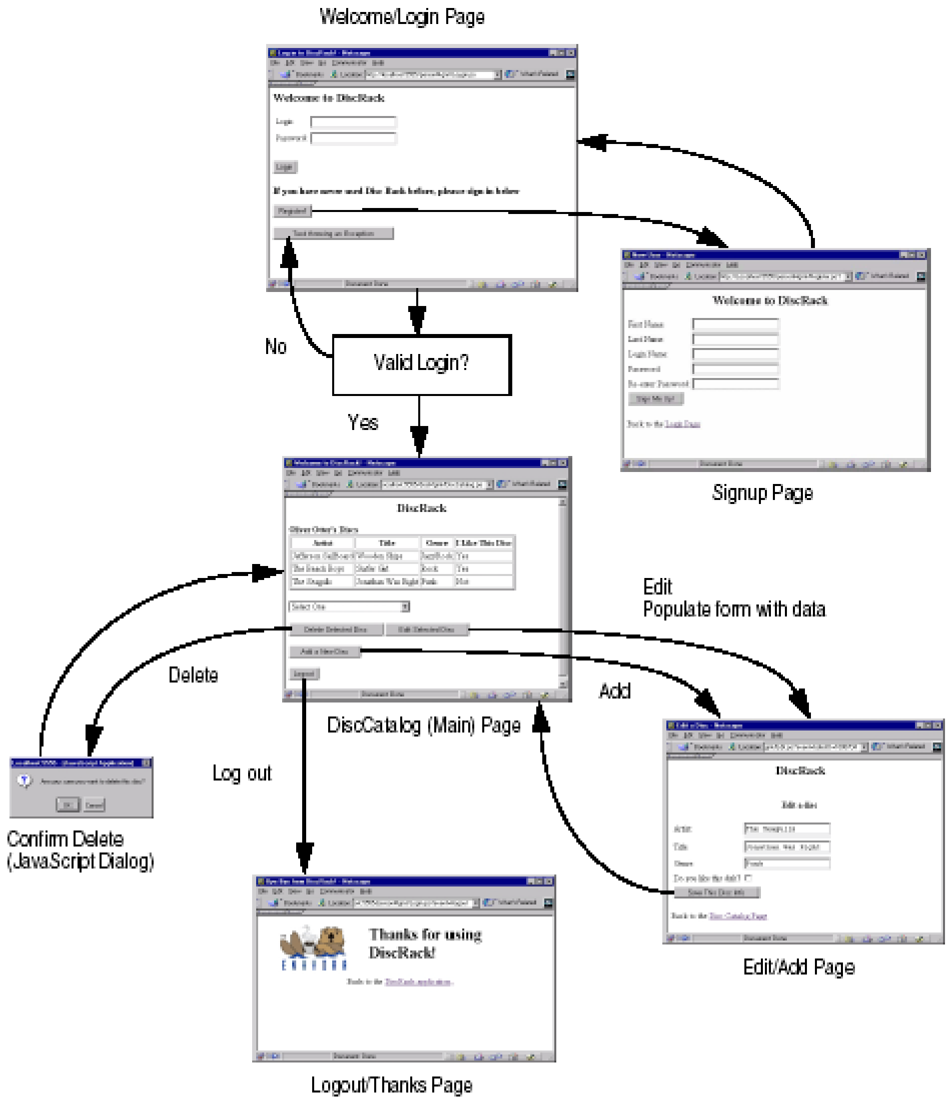
Design and storyboard

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, which is largely an application flowchart, is sometimes referred to as a site *map*, in contrast to a mocked-up HTML storyboard. This book refers to both as a storyboard. The storyboard for DiscRack is shown in Figure 5.2.

Figure 5.2 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 the DiscRack login page.

- Click the Login button to log in and see the disc catalog.
- Click the Sign Up button to display the Signup page.
- Click around on the links to can see the rest of the storyboard.

The flow of the HTML pages follows Figure 5.2. 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.

Developing, testing, and deploying

To finish the application process, the remaining steps include developing, testing, and finally deploying.

When you build an application from the top level, the build.xml files create an output directory containing the configuration files and the run script. Also, there will be a lib directory with a .jar file that contains all the class files for the application, along with any other files (for example, GIFs or stylesheets).

To deploy the application, you 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.

The rest of this chapter describes the DiscRack application itself.

Overview of DiscRack

The basic DiscRack application consists of 23 classes in 9 packages. The fundamental package structure and class functions for DiscRack are described in Table 5.1:

Table 5.1 DiscRack Application Overview

Class or package name	Description
discRack package	
DiscRack	Application object
DiscRackException	Simple base exception class
Presentation layer/package	
BasePO	Abstract base class for all presentation objects
DiscRackSessionData	Container for session data
ErrorHandler	Class to handle exceptions not caught elsewhere in the application
DiscRackPresentationException	Presentation layer exception class
presentation.personMgmt package	Package that contains the Register and Login classes for managing presentation related to the PERSON table
presentation.discMgmt package	Package that contains the Edit and DiscCatalog classes for managing presentation related to the DISC table
Business layer/package	
DiscRackBusinessException	Business layer exception class
business.person package	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

Table 5.1 DiscRack Application Overview (continued)

Class or package name	Description
Person object for a user name	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" in the chapter 4.
WAP layer/package	

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.

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 subclassed. Also, some of its methods are declared abstract, so subclasses must implement them.

BasePO has methods to handle some of the key tasks for DiscRack:

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

Note 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 run(). Its logic is very simple:

- Initialize or get the existing session data by calling **initSessionData()**.
- If this presentation object requires a log in (as determined by **loggedInUserRequired()**, an abstract method implemented by each presentation object), 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 are explained in the following sections.

The **run()** method has one 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" in the chapter 4. In contrast to the way session information was handled in that example, DiscRack stores all its session information in a single **DiscRackSessionData** object and saves that object in the user's session.

DiscRackSessionData is a simple container class containing methods to get and set these member properties:

- A **Person** object that represents the user
- A string, called **userMessage**, for error messages such as "Please choose a valid disc to edit"

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.

  Because **Session.getSessionData()** returns a generic Object, if you store session data separately, you will 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.

initSessionData() method

The first thing each presentation object 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 typecast to DiscRackSessionData; otherwise, the code creates a new DiscRackSessionData object and saves it to the user’s session with set().

loggedInUserRequired() method

BasePO has an abstract method called **loggedInUserRequired()** that returns a boolean value, which indicates whether a user is required to be logged in to access the associated page. Thus, every presentation object is required to implement this method.

In **BasePO.run()**, if this method returns true, then **checkForUserLogin()** is called.

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 returns 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

While you could create a separate presentation object for each task in an application, in many cases it makes sense to have a single presentation object handle multiple events. For example:

- **Edit** presentation object 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.

- **Login** presentation object handles three events -show page, login, and logout.

Note In this context, an “event” refers to the task a user is performing.

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, this URL specifies the event **showAddPage**:

http://Localhost:5555/discMgmt/Edit.po?event=showAddPage

DiscRack illustrates several techniques for setting the event:

- **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”, later in this chapter.

- 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.

- **exit** event is defined in the **DiscCatalog.html** page by the second form’s ACTION attribute:

  ```
  "../personMgmt/Exit.html"
  ```

  At compile time, this URL, as explained in “URL mapping” (later in this chapter), is replaced by:

  ```
  ../personMgmt/Login.po?event=logout'
  ```

  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 presentation object to another. To specify an event, add this string to the URL string passed to the constructor of PageRedirectException:

  ```
  "?event=someEvent"
  ```

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 presentation object corresponding to the current event. Reflection lets you 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 presentation object class that needs to handle that event.

HTML pages

You will find the HTML pages for DiscRack 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 options.xmlc files in the presentation layer (directory resources) controls how the application uses the HTML files.

Maintaining the storyboard

The *storyboard* is initially just a mockup of the application. But with a few simple steps, you can maintain a 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, which may include enhanced graphics, JavaScript special effects, stylesheets, and so on. An example of doing this is illustrated in "Replacing the user interface", later in this chapter.

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

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 build process, you must create an XMLC options file **options.xmlc**. For example:

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 replaces 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 like Sonny and Cher, however, 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.

Note Because this action happens at runtime, it may have a slight affect on performance. If performance is critical, you may want 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 effort 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 DiscRack 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 graphics and a stylesheet.

To replace the original storyboard resources with the “finished” resources:

1 Rename the **resources** directory to **resources_old**.

2 Rename the **resources_finished** directory to **resources**.

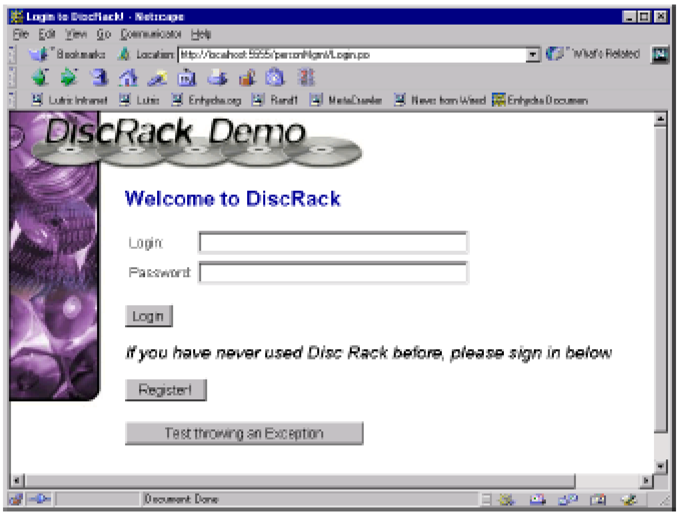
3 Rebuild the **presentation** package by entering the following commands from the directory <DiscRack_root>/discRack/**presentation**:

```
ant clean
ant
```

The **ant clean** command removes all the old classes so that **ant** will completely rebuild the application from scratch.

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

Figure 5.3 Browser displaying the DiscRack Login presentation object with updated graphics



If you have never used Disc Rack before, please sign in below

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 retrieves the DOM object corresponding to the template OPTION tag. The second line calls **getParentNode()** to get the container SELECT tag. Because 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 contain CLASS="discardMe", which causes XMLC to remove those items at build time, as explained before in “Removing dummy data”.

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:

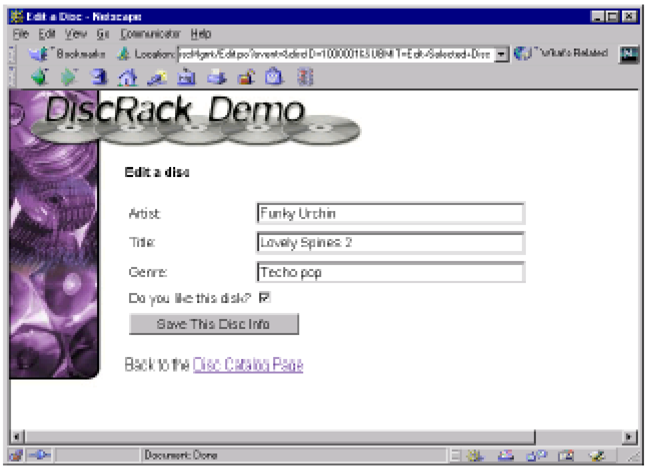
```
<SELECT name='discID' id='TitleList'>
<OPTION value='invalidID' selected>Select One</OPTION>
<OPTION value='1000001'>Funky Urchin: Lovely Spines</OPTION>
<OPTION value='1000021'>The Seagulls: Screaming Fun</OPTION>
</SELECT>
```

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 can 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 the Edit Disc button, the browser displays a form. As shown in Figure 5.3, the edit form is populated with the existing values for that disc. The user can then edit the values and submit them back to the database.

Figure 5.4 DiscRack disc edit form



Edit a Disc - Netscape

DiscRack Demo

Edit a disc

Artist: Funky Urchin

Title: Lovely Spines 2

Genre: Techo pop

Do you like this disk? ☑

Save This Disc Info

Back to the Disc Catalog Page

Document: Done

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);
```

Next, 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 request parameters.

Note **ClientPageRedirectException** is a subclass of java.lang.Error, so it is not caught by the catch clause when that statement 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 deletes some of the values, the edit page redisplays, maintaining all the non-null form element values and restoring the previous values to the null-valued form elements. The page also displays the error string.

Business layer

The DiscRack business layer is simple, consisting primarily of:

- Two packages - **Disc** and **Person**
- Two corresponding factory classes - **DiscFactory** and **PersonFactory**.

A *factory* is an object whose primary role is to create other objects.

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

PersonFactory has one static method, **findPerson()**. It 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 how DiscRack uses the 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 **setQuery*xxx()*** 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 configurations

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 (e.g., simpleApp.conf). Configurable items in the code snippets that you need to specify, such as path names or database identifier, are enclosed in brackets and italicized (for example, <*path_name*> or <*database_id*>).

Driver configuration

Important 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 the JDBC driver in your system CLASSPATH and not in the Enhydra application's CLASSPATH.

A common way to specify the path to the JDBC driver is to save the driver in a lib directory in the project and define the CLASSPATH in the run 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 run file template, start.in, (in the <*appName>/*input directory) to place the driver in your CLASSPATH. For example:

```
...
#
# Build up classpath.
#
CLASSPATH="../lib/idb.jar\;../lib/jta-spec1_0_1.jar"
APPCP="${ENHYDRA_LIB}${PS}../classes"
...
```

3 Build the project with ant, which will copy the run 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, where <*database_id*> is your database identifier.

```
#--------------------------------------------------------------------
# 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>:<port#>:<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, where *<database_id>* is your database identifier.

```
#-----------------------------------------------------------------
# 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>.Connection.Url =
jdbc:informix-sqli://<hostname>:<port#>:INFORMIXSERVER=<db_instance>;
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>.ObjectId.MinValue = 1
```

Sybase

This section presents an example of a Sybase configuration, where *<database_id>* is your database identifier.

```
#-----------------------------------------------------------------
# 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.

Note Although some older versions of MySQL may work with Enhydra without problems, versions 3.22 and earlier do not support transactions. Because of this, you have to make a small patch to the Enhydra code to use MySQL.

Patch

Prior to version 3.23, MySQL does not support transactions, and therefore does not support autocommit. To use MySQL versions 3.22 and earlier, you have to make a small change to the code and rebuild Enhydra. You will 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, where *<database_id>* is your database identifier.

```
#---------------------------------------------------------------------
# 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://<hostname>:<port#>/<db_instance>"
```

PostgreSQL

Note Although other versions are available commercially, **the Together company** supports the open-source version of PostgreSQL for the Linux operating system for use with Enhydra.

PostgreSQL is a popular open-source database used with Enhydra. However, as explained in “Loading the schema” in the chapter 4, 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 your 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.

InstantDB

To use an InstantDB database with an Enhydra application

1 In the application configuration file *<appName>/output/conf/*<appName>.conf (or better, in *<appName>/input/conf/*<appName>.conf.in) set the following line:

```
DatabaseManager.DB.<database_id>.Connection.Url = "jdbc:idb:<propFile>.prp"
```

where *<propFile>* is the full path to the database properties file, and *<database_id>* is the database identifier used in the configuration file.

2 In the same configuration file, identify the JDBC driver with the line:

```
DatabaseManager.DB.<database_id>.JdbcDriver = "org.enhydra.instantdb.jdbc.idbDriver"
```

3 Add the path to idb.jar to the setting for CLASSPATH in the application’s run script, in *<appName>/*run, or better in . *<appName>/*run.in.

Microsoft SQL Server

The exact configuration settings for connecting to MS SQL server depend on the JDBC driver you are using. We do not recommend using the JDBC-ODBC bridge with MS SQL Server.

JTurbo JDBC driver

We certified the JTurbo 2.0 JDBC driver, and the configuration settings for this are:

```
# JTurbo 2.0 JDBC Driver for MS SQL server
DatabaseManager.Databases [] = "my_db"
DatabaseManager.DefaultDatabase = "my_db"
DatabaseManager.DB.my_db.ClassType = "Standard"
DatabaseManager.DB.my_db.JdbcDriver = "com.inet.tds.TdsDriver"
# NOTE: substitute your server's IP address for 10.0.0.18 below
# Substitute the port your DB is listening on for 1433 below
DatabaseManager.DB.my_db.Connection.Url = "jdbc:inetdae:10.0.0.18:1433?database=my_db"
DatabaseManager.DB.my_db.Connection.User = "<user_name>"
DatabaseManager.DB.my_db.Connection.Password = "<password>"
```

If you are using another JDBC driver, you need to determine the driver package, for the DatabaseManager.DB.my_db.JdbcDriver setting, and connection string, for DatabaseManager.DB.my_db.Connection.Url setting.

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.

Because 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" in the chapter 4. 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:

1 Go to Start|Settings|Control Panel and click ODBC Data Sources.

2 Click the Add button in the dialog box that comes up.

3 Select the Microsoft Access Driver in the Create New Datasource dialog box and click Finish.

The ODBC Microsoft Access Setup dialog box appears.

4 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 OK.

2 Place database information in the application's configuration file, as shown in the example below. Replace *<data_source>* with the name you chose for Data Source Name in the preceding step.

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, where *<database_id>* is your database identifier.

```
#-------------------------------------------------------------------
# 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:<data_source>"
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
```

InterBase

InterBase® is an efficient and powerful RDBMS engine. Its vendor, Borland/Inprise, has released InterBase version 6.0 as an open-source product. See http://www.interbase.com for more information and product downloads.

InterClient

The JDBC driver for InterBase is called InterClient™ The InterClient system includes an all-Java thin client, and a server-side daemon (also known as a service on Microsoft Windows NT) called InterServer. This daemon accepts JDBC connection requests and in turn connects to the InterBase RDBMS daemon. The three processes (JDBC client, InterServer daemon, InterBase daemon) can run all on separate hosts, all on the same host, or in any other combination.

InterClient is a class 3 JDBC driver in that it has a separate daemon on the server to serve JDBC connections; however, it also matches the definition of a class 4 driver because the client component can connect only to one DBMS back-end, InterBase.

InterClient is installed separately from InterBase. On Windows, InterClient is commonly installed in:

```
C:\Program Files\Borland\InterClient\interclient.jar
```

Depending on the version of InterClient, it might instead be installed in:

```
C:\Program Files\InterBase Corp\InterClient\interclient.jar
```

Find the JAR file and append its location to your system CLASSPATH environment variable on the client host where you run Java applications.

Different versions of InterClient are available.

- InterClient version 1.50x works only with JDK 1.1x.
- InterClient version 1.51x works only with JDK 1.2.x.

Whichever version of InterClient you use, you must use the matching version of InterServer.

Configuration

You need to configure both the dods.conf and your <*application*>.conf to support InterClient.

DODS configuration

You should apply the following configuration edits to dods.conf to make the Standard_JDBC database class match InterBase features. This is necessary because there is not yet a specific com.lutris.appserver.server.sql.interbase package in the Enhydra sources.

```
Database.OidDbType.Standard_JDBC= "DECIMAL(9,0)"
Database.BitType.Standard_JDBC= "SMALLINT"
Database.TimeType.Standard_JDBC= "DATE"
Database.TimestampType.Standard_JDBC= "DATE"
Database.OnCascadeDelete.Standard_JDBC= true
Database.StringQuoteCharacter.Standard_JDBC= '
Database.StringMatch.Standard_JDBC= "LIKE"
Database.StringWildcard.Standard_JDBC= "%"
```

Application configuration

This section presents an example of an Interbase configuration, where *<database_id>* is your database identifier.

```
#-------------------------------------------------------------------
# Database Manager Configuration
# InterBase / InterClient
#-------------------------------------------------------------------
DatabaseManager.Databases[] = "<database_id>"
DatabaseManager.DefaultDatabase = "<database_id>"
DatabaseManager.Debug = "false"
DatabaseManager.DB.<database_id>.ClassType = "Standard"
DatabaseManager.DB.<database_id>.JdbcDriver = "interbase.interclient.Driver"
DatabaseManager.DB.<database_id>.Connection.Url =
"jdbc:interbase://loopback/<path_to_database>"
DatabaseManager.DB.<database_id>.Connection.User = "sysdba"
DatabaseManager.DB.<database_id>.Connection.Password = "masterkey"
```

Configuration notes

The JDBC driver class is interbase.interclient.Driver.

Server name

The general URL format for InterClient JDBC connections is as follows:

```
jdbc:interbase://servername/<path_to_database>
```

where *<path_to_database>* is the full path to the database file, including the name of the database (for example, /usr/local/data/inventory.gdb).

The *servername* is the hostname or IP address of the server running InterServer, the server-side daemon that accepts JDBC connection requests. If your Enhydra application runs on the same host where InterServer runs, you can use the special servername loopback.

Pathnames

The *<path_to_database>* is an absolute path to the InterBase database file on the server where the InterBase RDBMS server runs. InterBase does not have abstract handles to databases, like some database products do (for example, Oracle SIDs or BDE aliases). You must specify the real path to the database. You cannot use mapped drives or NFS filesystems in this path.

Notice the literal slash character (/) following the server name. If the absolute path starts with a slash character (/), then you should have a pair of slash characters (//) together. For example:

```
jdbc:interbase://servername//usr/local/data/inventory.gdb
```

If the server is a Windows host, the path starts with a drive letter identifier:

```
jdbc:interbase://servername/C:/data/inventory.gdb
```

If InterServer runs on a different host than the InterBase RDBMS server, you must specify this host in the path to database, with the following syntax:

jdbc:interbase://<*interserver_host*>/<*interbase_host*>:<*path_to_database*>

Tip Slash (/) and backslash (\) characters within path names are interchangeable to InterBase; the InterBase daemon translates these characters as needed to match the convention on the server platform. It is easier to use slashes in code, however, because escape sequences are required to represent backslashes in code.

Ports

InterBase does not take a port number argument in connection strings. InterClient and InterServer always communicate using the TCP/IP service named interserver, which defaults to port 3060. InterServer and InterBase always communicate using the TCP/IP service named gds_db , which defaults to port 3050. These services and port numbers are registered with IANA.

Username and password

The username sysdba and its default password masterkey are used in the example configuration above, but for security reasons it is recommended that you: (a) change the default sysdba password on your InterBase server, and (b) create a non-superuser login in the InterBase password database, and use that login for general database access.

Appendix B

Multiserver Administration Console

The Enhydra Multiserver Administration Console, also called the Admin Console, is a built-in component of the Multiserver itself. The Multiserver is a servlet runner that runs Enhydra applications, Java servlets, and JavaServer Pages (JSPs). The Multiserver can accept direct HTTP requests or requests forwarded from a Web server by Enhydra Director. It has controls that allow you to add, remove, start, stop, configure, and monitor your applications.

An Enhydra Multiserver installation typically consists of a directory containing applications, a configuration file called multiserver.conf, and a script to start the Multiserver. The Admin Console, which is actually just another Enhydra application running on the Multiserver, gives you access to the server and allows you to set properties defined in multiserver.conf.

Launching the Admin Console

The Admin Console is an Enhydra application that allows you to add, remove, and configure applications to run with Enhydra.

Starting the Admin Console

To launch the Admin Console, follow these steps:

Important: In Windows, directory <*enhydra_root*>\bin must be included in the CLASSPATH.

1 Type the following command at the command prompt to start the Multiserver:

In UNIX, type:

```
<enhydra_root>/bin/multiserver
```

where <*enhyra_root*> is the root of your Enhydra installation.

In WINDOWS, type

```
multiserver
```

Invoking the multiserver command without giving it a multiserver.conf as an argument brings up the server in the default installation.

Note If the Admin Console does not start, the path environment variable is not set correctly. The Enhydra installation instructions provide information about setting your path environment variable. The installation instructions are available in HTML format only on the Enhydra CD (refer to the top-level index.html)

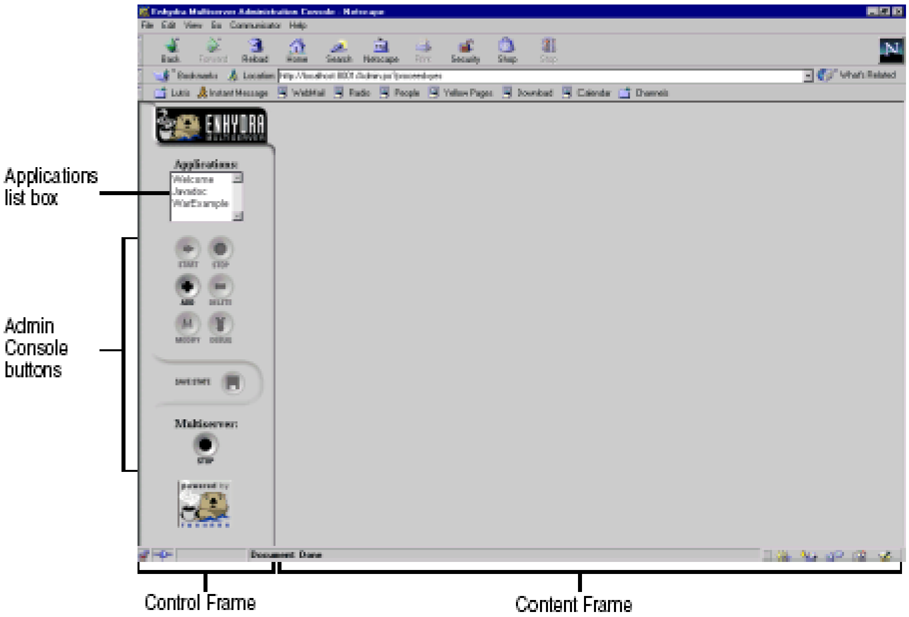
2 In your browser, display the console by entering this URL:

http://localhost:8001/

3 The console displays a password dialog box, as shown in Figure B.1. To get started, enter the default user name, **admin**, and password, **enhydra**.

The Admin Console appears in your Web browser as shown in Figure B.1, “Admin Console display.”

Figure B.1 Admin Console display


Enhydra Multiserver Administration Console - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
ENHYDRA
Applications:
Welcome
Javadoc
WarExample
Multiserver:
Document: Done
Applications list box
Admin Console buttons
Control Frame
Content Frame

Admin Console display

As shown in Figure B.1, “Admin Console display,” the Admin Console is divided into two frames:

- The control frame on the left contains the console buttons and the Applications list box (or window).
  The Applications window contains a list of all the applications, servlets, and Web archives (WARs) available in the Enhydra Multiserver. The console buttons below the Applications window initiate operations on the selected application.

- The content frame on the right displays information about the application that is currently selected in the Applications window. Some Admin Console functions are accessed from the content frame to display information or to request input.

Note There are two kinds of Web applications:

- Enhydra super-servlet applications, created with Enhydra development tools

- Servlet applications, otherwise known as WAR files (see “Creating a WAR file” on 113)

Control frame

As shown in Figure B.1, “Admin Console display,” the control frame has two components, an Applications window and the console buttons. Both components are described in the following sections.

Applications window

The Applications window contains a list of all applications currently available in the Enhydra Multiserver. You will see three sample applications that are initially available:

- Welcome application
- Javadoc servlet
- WarExample Web archive.

Admin Console buttons

Table B.1 describes the function of each button in the control frame.

Table B.1 Admin Console buttons

Button	Description
Start	Starts the application currently selected in the Applications window Unavailable when the selected application is already running
Stop	Stops the application that is selected in the Applications window. If the application has active users, you are prompted to verify that you want the application stopped.
Add	Adds an application to the Enhydra Multiserver.
Delete	Removes the selected application from the Enhydra Multiserver
Modify	Modifies the configurable attributes of the selected application
Debug	Invokes the debugging utility for the selected application. When you click this button, the debugging control panel displays.
Save State	Saves the state of the Enhydra Multiserver.
Multiserver Stop	Stops the Enhydra Multiserver.

Content frame

As shown in Figure B.1, the content frame displays information relevant to the current activity of the Multiserver, typically showing Status, Modify, or Debug windows with their various tab sections. The information you see in these screens varies with the type of application (Enhydra application, servlet, or Web archive),and with the specifics of the current task.

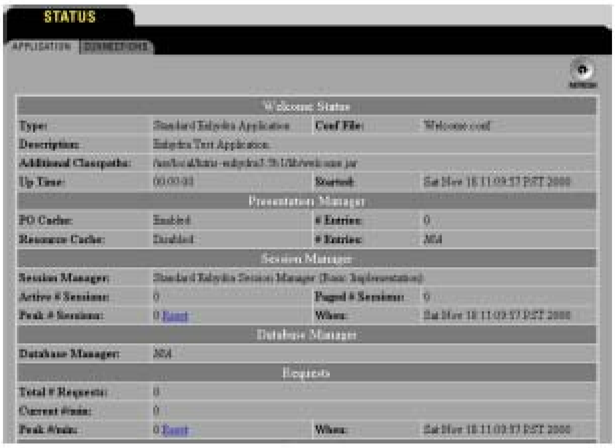
Viewing status information

To display status information about an application, select its name in the Applications list box (See Figure B.1). The Status window appears in the content frame, containing an Applications tab and a Connections tab. Figure B.2 shows a portion of the Status window for the sample Welcome application.

Note The elements of the screen will be somewhat different for a servlet or WAR.

The Application tab contains information about the application including its CLASSPATH, session manager status, database manager status, and traffic statistics.

Figure B.2 Status display for an application



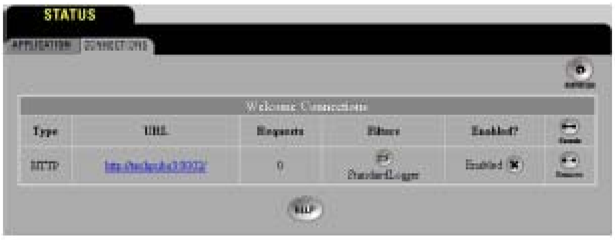
Welcome Status			
Type:	Standard Enhydra Application	**Conf File:**	Welcome.conf
Description:	Enhydra Test Application		
Additional Classpaths:	[illegible]		
Up Time:	00:00:00	**Started:**	Sat Nov 18 11:03:57 PST 2000
Presentation Manager			
PO Cache:	Enabled	**# Entries:**	0
Resource Cache:	Disabled	**# Entries:**	N/A
Session Manager			
Session Manager:	Standard Enhydra Session Manager (Basic Implementation)		
Active # Sessions:	0	**Paged # Sessions:**	0
Peak # Sessions:	0 Reset	**When:**	Sat Nov 18 11:03:57 PST 2000
Database Manager			
Database Manager:	N/A		
Requests			
Total # Requests:	0		
Current #/min:	0		
Peak #/min:	0 Reset	**When:**	Sat Nov 18 11:03:57 PST 2000

Viewing connections status information

Click the Connections tab to display connection status information for the application. Connections represent channels for requests coming into the application.

For example, an application could be receiving direct HTTP requests on port 8000 and requests coming from a Web server via an Enhydra Director connection on port 8020. Figure B.3 shows an example of the connection status display for an application receiving direct HTTP requests.

Figure B.3 Connection status display for a running application



Welcome Connections					
Type	**URL**	**Requests**	**Filters**	**Enabled?**	
HTTP	[illegible]	0	StandardLogger	Enabled	

Using the Admin Console

This section describes how to use the Admin Console to work with your Enhydra applications and servlets. This section provides information on:

- Adding an application
- Stopping an application
- Deleting an application
- Modifying the configuration of an application
- Debugging an application
- Saving the state of the Multiserver

Adding an application

This section describes how to add an Enhydra super-servlet application, a single servlet, or a servlet application set up as a Web Archive (WAR) to the list of applications running on the server.

Note A WAR is a collection of servlets bound together for convenient administration. Using the Admin Console, you can add a WAR to the Multiserver in just a few steps.

For convenience, we recommend that you bundle your servlets into a WAR and add them all in one process - see “Adding a servlet application configured as a WAR” (later in this chapter).

Adding an Enhydra super-servlet application

Follow these steps to add an Enhydra application to the Multiserver:

1 Copy the application’s configuration file from the application’s root directory to *<enhydra_root>*/apps/. For example:

```
cp simpleApp.conf /usr/local/lutris-enhydra3.5b1/apps/
```

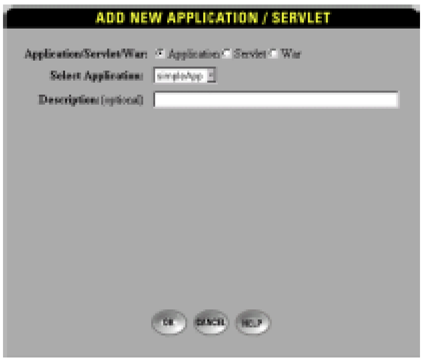
2 In the new simpleApp.conf file, locate the server.Classpath variable. Comment out the first line, and uncomment the second line. Set the server.Classpath variable equal to the new absolute CLASSPATH.

```
#server.Classpath[] = ../classes
Server.Classpath[] = "/enhydra/myapps/simpleApp/output/lib/simpleApp.jar"
```

Then save and close the configuration file.

3 Click the Add button to display the Add New Application/Servlet dialog box, and select the Application radio button if it’s not already selected.

Figure B.4 Add New Application/Servlet dialog box



Application/Servlet/War: ● Application ○ Servlet ○ War
Select Application: simpleApp
Description (optional):

4 Select the name of your application from the pull-down list.

An application’s name only appears in the list if its configuration file is in the *<enhydra_root>/*apps directory, and if it has not already been added to the Multiserver.

Optionally, enter a description.

5 Click OK to add your application to the Multiserver.

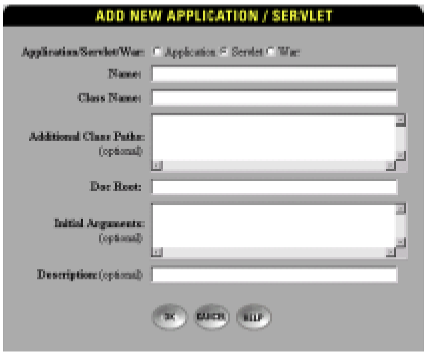
Adding a single servlet

This section describes how to use the Admin Console to add a single servlet to the Multiserver.

Note We recommend that you set up your servlet application as a WAR file. The following instructions, however, explain how to add a servlet not set up as a WAR file.

1 Click the Add button to display the Add New Application/Servlet dialog box, and select the Servlet radio button.

Figure B.5 Add New Servlet dialog box



2 In the fields provided, enter:

- Name of the servlet
- Name of the class to instantiate for the servlet
- Any additional classpaths required for the servlet
- Root of the servlet’s file system on disk
- Any initial arguments for the servlet (optional)
- Description of the servlet (optional)

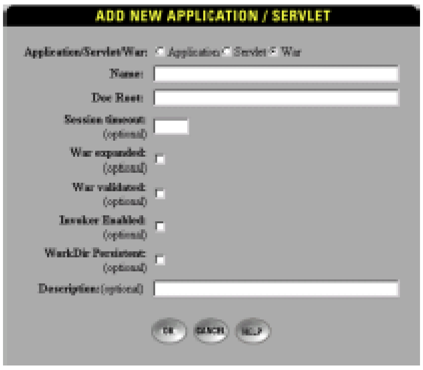
3 Click OK to add your application to the Multiserver.

Adding a servlet application configured as a WAR

This section describes how to use the Admin Console to add a WAR file to the Multiserver. For information on creating a WAR, see “Creating a WAR file” (later in this chapter).

1 Click the Add button to display the Add New Application/Servlet dialog box, and select the WAR radio button.

Figure B.6 Add New WAR dialog box


ADD NEW APPLICATION / SERVLET
Application/Servlet/War: ○ Application ○ Servlet ● War
Name:
Doc Root:
Session timeout: (optional)
War expanded: (optional) ☐
War validated: (optional) ☐
Invoker Enabled: (optional) ☐
WorkDir Persistent: (optional) ☐
Description: (optional)
OK
CANCEL
HELP

2 In the fields provided, enter:

- Name of the archive, not necessarily the file name.
- Doc Root - path to the root directory of the Web archive after it's unzipped.
- Session Timeout - period of time, in seconds, for which the session may remain idle before timing out.
- War Expanded - leave selected; all WARs must be expanded in this release.
- War Validated - leave selected; see Tomcat documentation for details.
- Invoker Enabled - select if you want to use /servlets/* syntax.
- WarDir Persistent - select if you want to save the work directory after the termination of the current Multiserver session. The work directory is where the Multiserver saves compiled JSPs. If you stop the Multiserver with Ctrl-C, you may want to manually remove the work directory before restarting the server to avoid reuse of previously compiled JSPs.
- Description - plain English description of this archive.

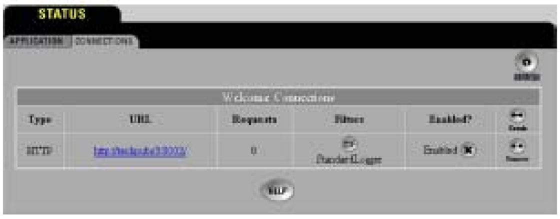
3 Click OK to finish adding the WAR to the Multiserver.

Specifying a connection method

Once you have added your application, you must establish a connection method for the item you've just added. To do that, follow these steps:

1 With the application selected in the Applications list box, click the Connections tab in the content frame.

Figure B.7 Live connection in the Connections tab

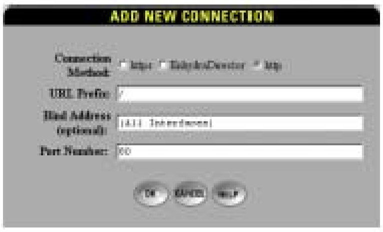


Welcome Connections

Type	URL	Requests	Filters	Enabled?
HTTP	[illegible]	0	StandardLogger	Enabled

2 Click Create in the Connections tab section to display the Add New Connection dialog box.

Figure B.8 Add New Connection dialog box



Connection Method: ○ https ○ EnhydraDirector ● http
URL Prefix: /
Bind Address (optional): (All Interfaces)
Port Number: 80

3 For Connection Method, choose

- **HTTP** for a standard Web connection, typically in a development environment.
- **HTTPS** for a secure Web connection, also in a development environment. This option is only available if you have configured your Enhydra installation with Sun's Java Secure Socket Extension Kit.
- **Enhydra Director** for connection via a Web server.

4 Enter a URL Prefix to define the portion of the URL that precedes the application. For example, the demonstration uses an URL prefix of /examples, making the full URL to the application

http://localhost/examples.

5 For Bind Address, enter an IP address if you want to bind to only one of the available IP addresses on a a given machine. If no IP address is specified, the server binds the given port for all IP addresses on the machine.

6 For Port Number enter an unused port number. Note that you must be root to bind to ports numbered below 1024 on UNIX systems. You may want to check that a given port is not already in use with the UNIX command netstat. The highest valid port number is 65535.

7 If your connection method is EnhydraDirector, three new configuration options are available: Session Affinity, Authentication Key, and HTTP Server URL. Session Affinity specifies whether Director will attempt to preserve session affinity in your application. Authentication Key, which specifies the password for the application if it was configured to use authentication in enhydra_director.conf, and HTTP Server URL, which specifies the URL of the application as set in enhydra_director.conf, are optional.

8 Click OK to return to the Status window.

9 To start the application, make sure it is selected in the Applications window, and click the Start button.

10 Click Save State to add the item permanently to the Multiserver. This overwrites the Multiserver’s configuration file, multiserver.conf.

Stopping an application

To modify or delete an application, you must first stop it. To do so, select the application’s name in the Applications list box, and click the Stop button.

In some situations, such as when users are still connected, you will be prompted to confirm your decision.

Deleting an application

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

Follow these steps to delete an application from the Multiserver:

1 Select the application in the Applications window.

2 Click the Stop button to stop the application.

3 Click the Delete button to delete it from the current session of the Multiserver.

4 Click the Save State button to overwrite the Multiserver configuration file and make the change permanent.

Modifying the configuration of an application

To modify an application, you must first stop it by using the Stop button. You can then edit parameters for the application. In the case of an Enhydra application, you can edit the application's configuration file using the Admin Console.

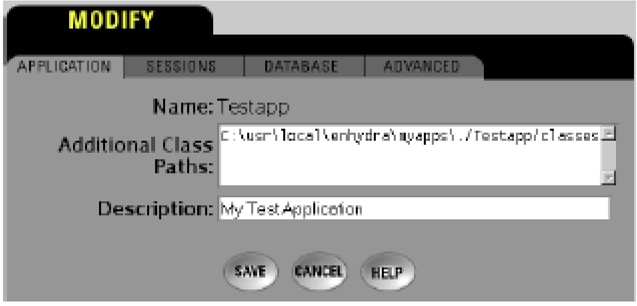
Use the following steps to modify the configuration of an application in Multiserver:

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.9.

Figure B.9 Modify Configuration window



Name: Testapp

Additional Class Paths: C:\usr\local\enhydra\myapps\./Testapp/classes

Description: My Test Application

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 CLASSPATHs for the application or servlet
- Sessions tab to modify Session Manager parameters
- Database tab to modify the database connection.
- Advanced tab to modify the application's default URL.

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

Debugging an application

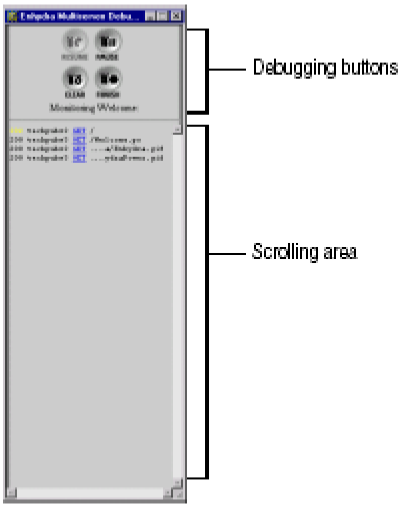
The debugging tool that comes with the Enhydra Multiserver is not a debugger in the classic sense of allowing you set breakpoints and step through your code. It is actually a traffic “snooper” that lets you see the requests going into your application and the responses being sent back. This capability can be very helpful for debugging HTTP-related issues when it is not always clear what is in the request coming from the client.

Use the following steps to debug an application running in the Multiserver:

1 Select the desired application in the Applications window.

2 Click the Debug button to display the debugging control panel, as shown in Figure B.10

Figure B.10 Debugging control panel



As shown in Figure B.10, 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.

3 Make three to five requests to your application using a browser. The active event list in the scrolling lower portion of the debugger window will list the requests as they come in. The method name is captured for each request.

4 Click the name of the response type - GET in the above example - to display the Debug window in the content frame, with its tab sections of Request, Trace, Sessions, and Response. Use this information as needed in monitoring, debugging, or modifying your application.

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 OK in the Confirmation dialog box.

Stopping and restarting the Multiserver

Stopping the Multiserver terminates all running applications as well as the Admin Console itself.

- Click the Multiserver Stop button in the control frame, then click OK to confirm.
- To restart the Multiserver, return to the Enhydra shell and proceed as directed in “Launching the Admin Console” explained at the beginning of this appendix.

Creating a WAR file

This section describes how to set up the directory structure of a WAR file to deploy Java servlets, JSPs, and static content. Under the Servlet 2.2 API, JSPs and servlets are assembled into a directory structure referred to as a Web application archive, or WAR. When the WAR is finished, you can compress it into a file with a .war extension. Once set up, it can be moved from server to server without further configuration. A good way to understand how to construct a WAR is to look at a simple example.

A simple WAR example

Suppose you want to deploy a JSP called Hello.jsp and a servlet called Hello.java. Set up the following directory structure:

```
/tmp/webApp
   myJspDir
      Hello.jsp
   WEB-INF
      classes
         Hello.class
      web.xml
```

The directory structure begins with a document root directory that has an arbitrary name, in this case, webApp. The JSP page can be placed either in the document root directory or in any of its subdirectories. In this example, it is placed in an arbitrary subdirectory named myJspDir. WEB-INF, the one required subdirectory, contains the configuration file web.xml and a directory called classes which in turn contains the compiled servlet classes. If you are only using JSPs, you do not need a classes subdirectory. The Multiserver adds the classes directory to its CLASSPATH, so the server automatically finds servlets placed in that directory.

The last required file, web.xml, contains deployment information like name mappings, parameters, and default file mappings. The web.xml file in this simple example contains the following essentially empty file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
        "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">
<web-app>
  <!-- configuration options would go here -->
</web-app>
```

Once you register your Web application with the Multiserver, you access the above pages with the following URLs:

- http://<*your_host*>/myPrefix/myJspDir/Hello.jsp
- http://<*your_host*>/myPrefix/servlet/Hello

The http://<*your_host*> section of the URL represents your host machine. When you configure the Multiserver to run the servlet, you tell it that the path to the document root is /tmp/webApp, and the URL prefix is some arbitrary string like /myPrefix.

Therefore, every request with the prefix myPrefix is forwarded to the Multiserver which in turn runs your application.

The remainder of the URL for the JSP page corresponds to the directory structure.

You can put HTML files in the same directory and request them in a similar manner.

Calls to the servlet require the reserved word “servlet” in the URL. When the servlet server sees it, it knows to look for the corresponding class in the classes subdirectory of the WEB-INF directory.

For more information

For a somewhat more complex example, see the WarExample application that ships with Enhydra. Beyond that, Sun’s Servlet 2.2 specification provides more information about configuring a WAR, containing both instructions and examples. You can download it from: http://java.sun.com/products/servlet/download.html.
