Getting Started with Lutris Enhydra
Contents

Chapter 1
Introduction
What you should already know 1
Conventions used in this book 1
Lutris Enhydra document set 3
Getting Started 3
Developer’s Guide 4
Wireless Application Developer’s Guide 4
Lutris documentation updates available online 4
Contacting Lutris Technical Publications 4
Where to find support and training for Lutris Enhydra 5
Registering your product online 5
Contacting Lutris Technical Support 5
Submitting bug reports to Lutris Technical Support 5
Lutris training 6
Available training courses 6
Contacting Lutris Education Services 6
Additional Enhydra information available online
Enhydra.org 6
Enhydra.org mailing lists 7
Enhydra.org working groups 7
Enhydra.org community documentation 8
Open-source software downloads 8
Acknowledgments 8

Chapter 2
Installation
Installation instructions also freely available online 9
Lutris Enhydra installation instructions freely available to Enhydra.org users 9
Why are installation instructions available in HTML format only? 9

Chapter 3
Overview
What is Enhydra? 11
What’s new in Lutris Enhydra 12
Anatomy of an Enhydra application 13
Enhydra super-servlet applications 14
Application objects 14
Presentation objects 15
Servlet applications 15
Servlet versus super-servlet applications 16
Application layers 17
Multiserver runtime component 17
Enhydra Director 18
Multiserver Administration Console 19
Enhydra application framework 19
Presentation Manager 20
Session Manager 20
Database Manager 21
Enhydra tools 21
Enhydra Application Wizard 21
Extensible Markup Language Compiler (XMLC) 22
Dynamic recompilation 22
Data Object Design Studio (DODS) 22
Kelp tools 24
Enhydra Application wizard 24
XMLC Compiler wizard 24
XMLC property pages 25
Enhydra sample project 25

Chapter 4
Tutorial: Building Enhydra applications
Creating your first application 27
Building the application 29
How it works 30
Directories and files in SimpleApp 31
Configuration files 32
Launching the Admin Console 33
Adding simpleApp to the Admin Console 34
Specifying a connection method 35
Starting and stopping an application 36
Using XMLC 37
Adding a hit counter 37
Understanding the Document Object Model 39
SPAN and DIV tags 40
Using XMLC from the command line 40
-dump option 41
-class and -keep options 41
Enhydra programming 42
Maintaining session state 42
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adding a servlet application configured as a WAR.</td>
<td>107</td>
</tr>
<tr>
<td>Specifying a connection method</td>
<td>108</td>
</tr>
<tr>
<td>Stopping an application</td>
<td>109</td>
</tr>
<tr>
<td>Deleting an application</td>
<td>109</td>
</tr>
<tr>
<td>Modifying the configuration of an application.</td>
<td>110</td>
</tr>
<tr>
<td>Debugging an application</td>
<td>110</td>
</tr>
<tr>
<td>Saving the state of the Multiserver</td>
<td>111</td>
</tr>
<tr>
<td>Stopping and restarting the Multiserver</td>
<td>112</td>
</tr>
<tr>
<td>Creating a WAR file</td>
<td>112</td>
</tr>
<tr>
<td>A simple WAR example</td>
<td>112</td>
</tr>
<tr>
<td>For more information</td>
<td>113</td>
</tr>
</tbody>
</table>

Index 115
Chapter 1, Introduction

This book introduces the Lutris® Enhydra™ application server and the Enhydra development environment. It provides an introductory overview of Enhydra and explains how to develop an application by using an example 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.
- Knowledge of basic UNIX commands and the UNIX make utility. This is not necessary if you are developing your application with the Kelp toolset in an IDE such as JBuilder.
- 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>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Italics</td>
<td>Indicates variables, new terms and concepts, and book titles. For example,</td>
</tr>
<tr>
<td></td>
<td>• A servlet is a Java class that dynamically extends the functionality of a</td>
</tr>
<tr>
<td></td>
<td>Web server.</td>
</tr>
</tbody>
</table>
### Conventions used in this book

#### Table 1.1  Typographical conventions (continued)

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fixed-width</strong></td>
<td>Used to indicate several types of items. These include:</td>
</tr>
<tr>
<td></td>
<td>• Commands that you enter directly, code examples, utility programs,</td>
</tr>
<tr>
<td></td>
<td>and options. For example,</td>
</tr>
<tr>
<td></td>
<td>• cd mydir</td>
</tr>
<tr>
<td></td>
<td>• System.out.println(&quot;Hello World&quot;);</td>
</tr>
<tr>
<td></td>
<td>• make utility</td>
</tr>
<tr>
<td></td>
<td>• -keep option</td>
</tr>
<tr>
<td></td>
<td>• Java packages, classes, methods, objects, and other identifiers. For</td>
</tr>
<tr>
<td></td>
<td>example,</td>
</tr>
<tr>
<td></td>
<td>• ErrorHandler class</td>
</tr>
<tr>
<td></td>
<td>• run() method</td>
</tr>
<tr>
<td></td>
<td>• Session object</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> Method names are suffixed with empty parentheses, even if the</td>
</tr>
<tr>
<td></td>
<td>method takes parameters.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> Only specific references to object names are in fixed-width;</td>
</tr>
<tr>
<td></td>
<td>generic references to objects are shown in plain text.</td>
</tr>
<tr>
<td></td>
<td>• File and directory names. For example:</td>
</tr>
<tr>
<td></td>
<td>• /usr/local/bin</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> UNIX path names are used throughout and are indicated with a</td>
</tr>
<tr>
<td></td>
<td>forward slash (/). If you are using the Windows platform, substitute</td>
</tr>
<tr>
<td></td>
<td>backslashes () for the forward slashes (/).</td>
</tr>
<tr>
<td><strong>Fixed-width italic</strong></td>
<td>Indicates variables in commands and code. For example,</td>
</tr>
<tr>
<td>and</td>
<td>xmlc [options</td>
</tr>
<tr>
<td>&lt;Fixed-width italic&gt;</td>
<td><strong>Note:</strong> Angle brackets (&lt; &gt;) are used to indicate variables in directory</td>
</tr>
<tr>
<td></td>
<td>paths and command options. For example,</td>
</tr>
<tr>
<td></td>
<td>• -class &lt;class&gt;</td>
</tr>
<tr>
<td><strong>Boldface</strong></td>
<td>Used for the words <strong>Note</strong>, <strong>Tip</strong>, <strong>Important</strong>, and <strong>Warning</strong> when</td>
</tr>
<tr>
<td></td>
<td>they are used as headings that draw your eye to essential or useful</td>
</tr>
<tr>
<td></td>
<td>information.</td>
</tr>
<tr>
<td><strong>Keycaps</strong></td>
<td>Used to indicate keys on the keyboard that you press to implement an</td>
</tr>
<tr>
<td></td>
<td>action. If you must press two or more keys simultaneously, keycaps are</td>
</tr>
<tr>
<td></td>
<td>joined with a hyphen. For example,</td>
</tr>
<tr>
<td></td>
<td>• &quot;Ctrl-C.</td>
</tr>
<tr>
<td>**</td>
<td>(pipe)**</td>
</tr>
<tr>
<td></td>
<td>interface (GUI), and to separate choices in a syntax line. For example,</td>
</tr>
<tr>
<td></td>
<td>• File</td>
</tr>
<tr>
<td></td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>• [a</td>
</tr>
<tr>
<td><strong>{} (braces)</strong></td>
<td>Indicates a set of required choices in a syntax line. For example,</td>
</tr>
<tr>
<td></td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>means you must choose a, b, or c.</td>
</tr>
<tr>
<td><strong>[] (brackets)</strong></td>
<td>Indicates optional items in a syntax line. For example,</td>
</tr>
<tr>
<td></td>
<td>• [a</td>
</tr>
<tr>
<td></td>
<td>means you can choose a, b, c, or nothing.</td>
</tr>
</tbody>
</table>
Table 1.2 lists additional conventions used in this book, including the convention used to describe the Enhydra root directory, platform-related conventions, and so on.

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

Table 1.2 Additional conventions

The Lutris Enhydra documentation set is an excellent resource for information about Enhydra. The documentation set includes the following printed guides.

Note

Online versions of these books in both PDF and HTML formats are provided with the purchase of Lutris Enhydra. These online books, along with additional Enhydra online documentation, are located in the `doc` subdirectory of the directory in which you installed Lutris Enhydra. You can also view the online books and installation instructions directly from the product CD.

Getting Started

*Getting Started with Lutris Enhydra* introduces the fundamentals of Enhydra. The purpose of this book is to introduce Lutris 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.

**Note**
As part of our commitment to support the Enhydra and open-source communities, Lutris Technologies has made the latest online version of *Getting Started with Lutris Enhydra* available for free viewing and download from the Lutris Documentation home page at http://www.lutris.com/documentation/index.html.

**Developer’s Guide**

The Lutris Enhydra *Developer’s Guide* introduces advanced topics and explores key features of Enhydra in detail. The purpose of the *Developer’s Guide* is to provide developers with the information they need to create and debug sophisticated Enhydra applications. This guide provides in-depth information on the Lutris Enhydra development tools:

- Application Wizard
- Multiserver Administration Console
- Kelp tools
- Enhydra™ XMLC
- Data Object Design Studio (DODS)
- InstantDB
- Enhydra Director

**Note**
The *Developer’s Guide* is available only with the purchase of Lutris Enhydra.

**Wireless Application Developer’s Guide**

The Lutris Enhydra *Wireless Application Developer’s Guide* presents information on wireless technologies and describes how to develop wireless applications with Enhydra. It includes a detailed tutorial and an explanation of the Enhydra AirSent wireless sample application.

**Note**
The *Wireless Application Developer’s Guide* is available only with the purchase of Lutris Enhydra.

**Lutris documentation updates available online**

The latest product documentation updates and release notes are available to registered users from the Lutris Documentation home page at http://www.lutris.com/documentation/index.html.

**Contacting Lutris Technical Publications**

We strongly encourage you to send us your feedback because it helps us understand your needs and makes our documentation even better. You can submit feedback from the Lutris website at
Where to find support and training for Lutris Enhydra

Lutris Enhydra includes a package of products for developing Enhydra applications, including open-source products. Lutris Technologies, Inc. provides support and services for Lutris Enhydra.

Note
Open-source communities or commercial entities support the other products. For detailed information on the available support options for those products, please refer to the appropriate group or company website.

Lutris support

Lutris offers a variety of support programs designed to assist you with your technical support needs. We can help with installing and using your Lutris product, developing and debugging your code, maintaining your deployed applications, providing onsite consulting services, and delivering enterprise-level support. For more information about any of the Lutris technical support programs, see the Lutris Support home page at http://www.lutris.com/support/index.html or call Lutris Customer Service toll-free at 1-877-688-3724, Monday–Friday, 8 a.m.–6 p.m. Pacific Time (outside of North America, please call 1-831-460-7590).

Registering your product online

Lutris strongly encourages you to register your product online. Registering your product entitles you to 15 days of free installation support and provides you with the option of purchasing Lutris Support Services.

To register online, browse to the product registration form that is available at http://www.lutris.com/register.html.

Contacting Lutris Technical Support

For more information about any of Lutris’ technical support programs, see the Lutris Support home page at http://www.lutris.com/support/index.html or call Lutris Customer Service toll-free at 1-877-688-3724, Monday–Friday, 8 a.m.–6 p.m. Pacific Time (outside of North America, please call 1-831-460-7590). You can also send email to support@lutris.com.

Submitting bug reports to Lutris Technical Support

To report suspected Lutris Enhydra bugs, fill out the Bug Report form available at http://lutrisbugs.custhelp.com/cgi-bin/lutrisbugs/people. We recommend that you choose the Search Bugs link before submitting a bug report so that you can see if your bug has already been reported. Be sure to include steps-to-reproduce, exact error messages, and code snippets, if applicable, to help us better evaluate your report.
Lutris training

Lutris wants to ensure your success. Our expert trainers guide participants through hands-on labs designed to provide an intensive learning environment where participants quickly learn how to maximize Lutris Enhydra in development and deployment environments.

Available training courses

The following courses are currently offered by Lutris Technologies. For more information about training offerings, see the Lutris Training home page at http://www.lutris.com/training/index.html.

Lutris Enhydra Fundamentals

Lutris Technologies currently offers a five-day, instructor-led course titled *Lutris Enhydra Fundamentals*. This course is intended primarily for developers, architects, project managers, IT staff, and consultants who will be using Lutris Enhydra or are evaluating it for future projects.

Building Wireless Applications

Lutris Technologies currently offers a two-day, instructor-led course titled *Building Wireless Applications with Lutris Enhydra*. This course is intended primarily for Enhydra developers who want to create applications that serve content to cellphones or other wireless devices.

Lutris Enhydra Database Techniques

Lutris Technologies currently offers a two-day, instructor-led course titled *Database Techniques with Lutris Enhydra*. This course is intended primarily for Enhydra developers working on applications that require existing database platform support, Java database specialists and DBAs responsible for maintaining the data layer of an Enhydra application, and evaluators interested in seeing the database capability available through Enhydra.

Contacting Lutris Education Services

For more information about training offerings, see the Lutris Training home page at http://www.lutris.com/training/index.html or call Lutris Customer Service toll-free at 1-877-688-3724, Monday–Friday, 8 a.m.–6 p.m. Pacific Time (outside of North America, please call 1-831-460-7590). You can also send email to training@lutris.com.

Additional Enhydra 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 website is the home of the Enhydra 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.

**Enhydra.org mailing lists**

The Enhydra.org website includes archives of the various electronic mailing lists that serve as the backbone of the Enhydra community, as well as instructions on how to subscribe to the mailing lists.

Lutris encourages you to join one or more of the following Enhydra email lists:

- **Enhydra@enhydra.org**
  The Enhydra mailing list for developer interaction. The Enhydra project team monitors this list. It is the ideal place to get answers to your questions from fellow Enhydra developers.

- **Enhydra-daily@enhydra.org**
  A daily collection of all mail sent to enhydra@enhydra.org is sent to subscribers of this list.

- **Enhydra-digest@enhydra.org**
  A weekly digest of all mail sent to enhydra@enhydra.org.

- **EnhydraEnterprise@enhydra.org**
  The Enhydra Enterprise mailing list is tailored for those who are developing and deploying Enhydra applications on a large scale. Here you can find answers to the more detailed Enhydra questions, such as those on Enterprise Java Beans (EJB) and the Common Object Request Broker Architecture (CORBA).

- **EnhydraEnterprise-digest@enhydra.org**
  A weekly digest of all mail sent to EnhydraEnterprise@enhydra.org.

- **Enhydra-announce@enhydra.org**
  The mailing list for receiving Enhydra announcements.

For information and instructions on joining one or more of these lists, go to http://www.enhydra.org/community/mailingLists/index.html.

**Mailing list archives**

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

**Enhydra.org working groups**

Enhydra working groups bring together developers interested in creating new Enhydra applications and contributing new technologies or bug fixes for Enhydra.
Each working group provides access to the current project source code and to the project email list. This lets you communicate with the project leaders and other developers.

For information and instructions on joining one or more of these groups, go to http://www.enhydra.org/project/workingGroups/index.html.

**Documentation working group**

The documentation working group is focused on facilitating developer-created documentation for open-source Enhydra and related technologies. The working group also provides a central point for documentation discussions and proposals.

Community members are encouraged to submit and collaborate on articles of any length, on topics of general interest to all Enhydra developers—from beginning to advanced.

For information and instructions on joining this group, go to http://www.enhydra.org/project/workingGroups/index.html.

**Enhydra.org community documentation**

The Enhydra website also has documentation provided by members of the community. For more information on this documentation, see http://www.enhydra.org/software/documentation/enhydra/index.html.

**Open-source software downloads**

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

**Acknowledgments**

As an open-source product, Enhydra benefits from the contributions of many developers around the world. In particular, Lutris would like to thank the following people who have contributed information used in some form in this book: Robert Cadena, G. W. Estep, Rohan Oberoi, Dan Rosner, Peter Speck, and David Trisna.
Complete step-by-step installation instructions for Lutris Enhydra and related software (including bundled third-party software) are available on the Lutris Enhydra CD. To begin, refer to the top-level index.html file on the Enhydra CD.

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

**Installation instructions also freely available online**


**Lutris Enhydra installation instructions freely available to Enhydra.org users**

In the spirit of supporting the Enhydra and open-source communities, Lutris Technologies has made the latest Lutris Enhydra installation instructions freely available from the Lutris Documentation home page at http://www.lutris.com/documentation/index.html.

**Note** The Lutris Enhydra installation instructions are tailored to the installation of Lutris Enhydra. Open-source Enhydra users will need to substitute filenames and pathnames appropriately.

**Why are installation instructions available in HTML format only?**

Previous versions of this book included printed installation instructions. Due to book-printing schedules, this installation information was usually not as accurate as the information provided on the Lutris Enhydra CD, which is able to be updated until just prior to release. Therefore, to ensure that you always receive the latest installation information, the installation instructions are now available only on the CD or online in an easy-to-print HTML format.
Why are installation instructions available in HTML format only?

It is our hope that by single-sourcing the installation instructions in print-friendly HTML and making the latest versions always available online, we are serving our Lutris Enhydra customers and the Enhydra community in the best possible way.

We hope you agree. Please feel free to send your comments to documentation@lutris.com.

Thank you,

Lutris Documentation Team
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?
- Anatomy of an Enhydra application
- Multiserver runtime component
- Enhydra application framework
- Enhydra tools

What is Enhydra?

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

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

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,” illustrates the basic elements of an Enhydra application within the context of the Enhydra architecture.
What is Enhydra?

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.

<table>
<thead>
<tr>
<th>For more information on</th>
<th>See this topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enhydra application architecture</td>
<td>“Anatomy of an Enhydra application” on page 13</td>
</tr>
<tr>
<td>Multiserver</td>
<td>“Multiserver runtime component” on page 17</td>
</tr>
<tr>
<td>Application framework</td>
<td>“Enhydra application framework” on page 19</td>
</tr>
<tr>
<td>Enhydra tools</td>
<td>“Enhydra tools” on page 21</td>
</tr>
</tbody>
</table>

What’s new in Lutris Enhydra

Enhydra 3.5 has a number of new features and enhancements, including:

- Improved application performance with Java 2 v1.3 support (JDK 1.3)
- Support for server-side redirects
- Wireless support enhancements, including:
  - XMLC support for VoiceXML, cHTML, and XHTML
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.
Anatomy of an Enhydra application

For a detailed comparison of the two application models, see “Servlet versus super-servlet applications” on page 16.

Enhydra super-servlet applications

An Enhydra application has, at a minimum:

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

These objects run in the context of the Enhydra application framework, as described in “Enhydra application framework” on page 19.

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” on page 19)

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.

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.
Anatomy of an Enhydra application

**Figure 3.2** Servlet application model

---

**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” on page 20). Likewise, the servlet API provides session management, so there is no need for the Enhydra Session Manager (see “Session Manager” on page 20).

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)
Additionally, Enhydra has a sophisticated make system, especially tailored to building XMLC-based applications. You cannot use this system when building servlet applications.

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 enhydra.jar).

### 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 graphically creates data objects to populate the data layer, and creates both Java code and SQL code to create the corresponding tables in the database. For more on DODS, see “Data Object Design Studio (DODS)” on page 22.

### 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 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.
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” on page 101.

**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.”
**Enhydra application 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” on page 27.

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.
Enhydra tools

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).

For additional information about the Enhydra Application Wizard, refer to Chapter 2, “Using the Application Wizard to create Enhydra applications,” in the Developer’s Guide.

**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” on page 27.

**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), as shown in Figure 3.4, is a graphical tool you can use to define your data model.
Data objects in the DODS data model 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).

Once you have defined your data model, the 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 data model 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.

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

For an example using DODS, see “Using DODS” on page 58.
Enhydra tools

Kelp tools

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

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

Figure 3.5 shows a general outline of Kelp, as features vary depending on the IDE. See Chapter 4, “Using Enhydra Kelp,” of the Developer’s Guide for the documentation included with the Kelp tools for specific features supported by your Java IDE.

![Figure 3.5 JBuilder project generated with Kelp](image)

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” on page 27. 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.
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 Chapter 5, “DiscRack sample application,” for a look at an application with more advanced features.

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

Creating your first application

The 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:
   ```bash
   mkdir myapps
   ```
2. Open a shell window and make the new directory the current directory. For example:
   ```bash
   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
Creating your first application

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 Lutris Enhydra installation instructions provide information about setting your path environment variable. The installation instructions are available in HTML format only on the Lutris Enhydra CD (refer to the top-level index.html) or the Lutris Documentation home page: http://www.lutris.com/documentation/index.html.

Figure 4.1  Application Wizard GUI

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 Generator.
   Select SuperServlet from the Generator 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 /enhydra/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 Create Makefiles and Create Shell Scripts and click Finish.
Creating your first application

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:
   ```bash
cd simpleApp
   ```

6 Browse the application root directory and note the following items created by the Application Wizard:
   - Two make files: config.mk and Makefile
   - A readme.html file that contains some simple instructions to build and run the application
   - A source directory, src, containing all the source code for the application
   - An input directory, input, containing templates of the configuration files and start scripts for the application.

The contents of this application root directory are explained in “Directories and files in SimpleApp” on page 31. In the next section, you will finish building your simple application.

Building the application

To build the application:

1 In the shell window, enter the make command from the application root directory:
   ```bash
cd /enhydra/myapps/simpleApp
make
   ```
   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

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

   When you build the application, make compiles the files 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/archive directory, along with the configuration files needed to run the application.

2 To start the application, enter the following commands in the Enhydra shell window:
   ```bash
cd output
./start
   ```

   Note: The Multiserver Administration Console provides a GUI for managing applications. Among other functions, the Admin Console can be used to start and
Creating your first application

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

http://localhost:9000

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

You have just built and run your first Enhydra application.

Now press Ctrl-C in the shell window to stop the Enhydra process.

How it works

The application created by the Application Wizard provides a simple example of how Enhydra works.

Look at the file myapps/simpleApp/src/simpleapp/presentation/Welcome.html, which contains a few dozen HTML tags. Notice tags such as these:

```html
<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 same directory. In particular, notice these lines of code:

```java
WelcomeHTML welcome;
String now;
...
welcome = (WelcomeHTML)comms.xmlcFactory.create(WelcomeHTML.class);
now = DateFormat.getTimeInstance(DateFormat.SHORT).format(new Date());
welcome.getElementTime().getFirstChild().setNodeValue(now);
```

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

The first couple lines of 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:

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

For a more detailed explanation of XMLC, see “Using XMLC” on page 37. For more information about the Enhydra Application Wizard, see “Enhydra Application Wizard” on page 21.

### 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 Application Wizard. The `simpleapp` directory is divided into three subdirectories for the business, data, and presentation layers. The `business` and `data` directories are empty in the new application. The `presentation` directory contains Java, HTML, and media files.

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

- The output subdirectory contains the application’s configuration files, `servlet.conf` and `simpleApp.conf`. It also has an archive directory 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 start 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 make program also copies the Multiserver configuration file, `servlet.conf`, and the start script to the output directory to make it easier to run the application.

Configuration files

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

- Multiserver configuration file, `servlet.conf`, located in the servlet 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 start 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 `make` 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
Launching the Admin Console

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 start script is that this one applies to the application’s own class loader, while the latter is global in scope for all applications run by that Multiserver instance.

- 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, add the simpleApp application to the console, and then use the Admin Console to start and stop simpleApp. Using the Admin Console to modify and debug Enhydra applications—as well as Java servlets and Web archives—is discussed in Chapter 3, “Using the Multiserver Administration Console,” of the Developer’s Guide.

To launch the Admin Console:

1. Enter the following command in the Enhydra shell at the prompt:

   `multiserver`

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

   `http://localhost:8001/`
Launching the Admin Console

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

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 with the `cp` command, or by copy-and-pasting from one folder to another in Windows. For example:

   ```bash
   cp simpleApp.conf /usr/local/lutris-enhydra3.5/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/archive/simpleApp.jar`, like this:

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

![Adding simpleApp using the Admin Console](image)

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

![Creating a connection for simpleApp using the Admin Console](image)

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.
Launching the Admin Console

- 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. For more information, see Appendix A, “Using SSL with Enhydra” of the Developer’s Guide.

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.

   For detailed information about the connection options, refer to Chapter 3, “Using the Multiserver Administration Console,” in the Developer’s Guide.

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 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 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.

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 an HTML file and creates a Java object that enables an application to change the HTML file’s content at runtime, without regard for its formatting. The Java objects that XMLC creates have interfaces defined by the Document Object Model (DOM) standard from the World Wide Web Consortium (W3C).

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 and WelcomePresentation.java in the presentation directory.

2. Add the following line of HTML to Welcome.html before the closing <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.

   ```java
   public void run(HttpPresentationComms comms)
   throws HttpPresentationException, IOException {
      HttpPresentationOutputStream out;
      WelcomeHTML welcome;
      String now;
      byte[] buffer;

      welcome = (WelcomeHTML)comms.xmlcFactory.create(WelcomeHTML.class);
      now = DateFormat.getTimeInstance(DateFormat.SHORT).format(new Date());
      welcome.getElementTime().getFirstChild().setNodeValue(now);
      // Increment the count and write into the html.
      welcome.setTextHitCount(String.valueOf( ++hitCount ));
      buffer = welcome.toDocument().getBytes();
      comms.response.setContentType( "text/html" );
      comms.response.setContentLength( buffer.length );
      out = comms.response.getOutputStream();
      out.write(buffer);
      out.flush();
   }
   ```
Getting Started with Lutris Enhydra

Using XMLC

4. Build the application by running `make` from the top-level `simpleApp` directory. Then restart Enhydra, either by starting the Admin Console (see “Launching the Admin Console” on page 33) or by entering the following commands in the shell window.

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

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

5. Test the application by loading `http://localhost:9000` 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

![Browser displaying the simpleApp Welcome page with a hit counter](image)

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:

```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:

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

In this example, the class `HTMLTableCellElement` and the method `setBgColor()` come from the W3C packages; `getElementById()` 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 `make`. To run XMLC from the command line, you must have the `enhydra.jar` file in your `CLASSPATH`. The make files set the `CLASSPATH` when they run. You must include `enhydra.jar` in the `CLASSPATH` yourself, before running XMLC from the command line.

Set your `CLASSPATH` with one of the following commands:

**UNIX**

```bash
export CLASSPATH=<{enhydra_root}/lib/enhydra.jar};
```

**Windows**

```bash
export CLASSPATH=<{enhydra_root}/lib/enhydra.jar};
```

For more information on configuring Enhydra, see the installation and configuration instructions in the CD documentation.

The basic command-line syntax of XMLC is

```
xmcl -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.


2. Add the following HTML to it:

   ```html
   <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 presentation directory and enter this command:

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

   XMLC displays the following in the shell window:

   ```
   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
     - LazyText: 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.
Enhydra programming

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

```bash
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” on page 20.

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 the
keyword strings that you use to get and set each object you want to save to the session.

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

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

   \(<P>Number of hits from you: \<SPAN ID="PersonalHitCount">no count</SPAN>\</P>\)

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

   \(\text{import com.lutris.util.*;}\)

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

   \(\text{final String hits = "HITS";}\)

   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/books/getting-started/samples/ directory.

   \(\text{try \{ \}
   \text{Integer personalHits = (Integer)comms.session.getSessionData().get(hits);} \)
   \(\text{if(personalHits == null) \{ \}
   \text{personalHits = new Integer(1);} \)
   \(\text{} \}
   \text{comms.session.getSessionData().set(hits, personalHits);} \)
   \(\text{\} \}
   \text{welcome.setTextPersonalHitCount( personalHits.toString() );} \)
   \(\text{welcome.setTextSessionID( comms.session.getSessionKey() );} \)
   \(\text{\} \}
   \text{catch (KeywordValueException e) \{ \}
   \text{comms.response.writeHTML("Session access error" + e.getMessage());} \)

   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.Object, 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

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 were running on a “real” server, you would see different numbers depending on how many times you had accessed the page versus the total number of hits. Notice also that the session ID string always stays the same.

Adding a new page to the application

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

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 a “stripped down” presentation object. The `run()` method should look like this:

```java
public void run(HttpPresentationComms comms) throws HttpPresentationException, IOException {
    HttpPresentationOutputStream out;
    SimpleHTML simple;
    String now;
    byte[] buffer;

    simple = (SimpleHTML)comms.xmlFactory.create(SimpleHTML.class);
    now = DateFormat.getTimeInstance(DateFormat.SHORT).format(new Date());
    simple.getElementById().getFirstChild().setNodeValue(now);
    buffer = simple.toDocument().getBytes();
    comms.response.setContentType( "text/html" );
    comms.response.setContentLength( buffer.length );
    out = comms.response.getOutputStream();
    out.write(buffer);
    out.flush();
}
```

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

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

7 Add the following lines just before the last statement in the `run()` method:

```java
HTMLHeadingElement heading = simple.getElementById("MyHeading");
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.

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

```makefile
CLASSES = \
    RedirectPresentation \n    WelcomePresentation \n    SimplePresentation
HTML_CLASSES = \
    WelcomeHTML \n    SimpleHTML
```
When you run `make` in the presentation directory, the HTML_CLASSES variable is passed to the XMLC -class option to create the specified classes. This is what you did when you ran XMLC from the command line, using the -class option. The Lutris convention is to create `fooHTML` class for a file `foo.html`. This convention is defined in the following file:

```markdown
<enhydra_root>/lib/stdrules.mk
```

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

By default, packages are generated in the classes subdirectory off of the application root directory, `simpleApp` in this example. Look in the classes/simpleapp/presentation directory to see the generated class files.

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

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

   ```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, and build the application from the top level:

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

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
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 \(<\text{BODY}\)\) tag.

   \(<\text{H2 align=center}>\text{Disc List}\</\text{H2}>\)
   \(<\text{TABLE border=3}\)\)
   \(<\text{TR}\)\)
   \(<\text{TH}>\text{Artist}\</\text{TH}>\text{Title}\</\text{TH}>\text{Genre}\</\text{TH}>\text{I Like This Disc}\</\text{TH}>\)
   \(<\text{/TR}\)\)
   \(<\text{TR id=TemplateRow}\>\)
   \(<\text{TD}>\text{Van Halen}\</\text{TD}>\text{Fair Warning}\</\text{TD}>\text{Good Stuff}\</\text{TD}>\text{Yes}\</\text{TD}>\)
   \(<\text{/TR}\>\)
   \(<\text{/TABLE}\)\)

This HTML contains a table with a single “template” row (in the second \(<\text{TR}\)\) tag). Notice that both this row and the \(<\text{SPAN}\)\) 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 \(<\text{enhydra_root}>/\text{doc/books/getting-started/samples/TableCode.java}\) into your application’s presentation directory and rename it SimplePresentation.java.

   \(<\text{Note}>\) If you like, you can save your old SimplePresentation.java to SimplePresentation.sav for future reference.\(<\text{/Note}>\)
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:

```java
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:

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

The next section of code removes the ID attributes from these objects. The reason for this is that the DOM requires that each ID in the document to be unique. When you make a copy of the table row, you would otherwise have duplicate IDs.

```java
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:

```java
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:

```java
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.

```java
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, and build the application from the top level:

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

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

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.
Enhydra programming

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:

```java
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:

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

```make
CLASSES =   
SimpleDiscList
```

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

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

1. Import the new class:

```java
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.

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

5. 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, and InstantDB.

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

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

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 Lutris 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/books/getting-started/samples`, into your data directory. For example, the copy command might look like the following:
   
   ```
   cp /usr/local/lutris-enhydra3.5/doc/books/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.
Connecting the application to a database

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/books/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/books/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

52 Getting Started with Lutris Enhydra
Connecting the application to a database

- 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 Appendix A, “Database configurations,” for configuration information for other databases.

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

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:

   ```bash
   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,

   ```bash
   cp usr/local/lutris-enhydra3.5/doc/books/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.

   ```java
   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());
       }
     }
   }
   ```
Connecting the application to a database

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:
   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 that
   should help you isolate the problem. Refer to your database JDBC documentation,
   and confirm your database driver and connection string information. The Enhydra
   mailing list is also a useful resource. See “Enhydra.org mailing lists” on page 7 for
   additional information about mailing lists.

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 start 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 start script: the template start.in is located in the
application’s input directory, and start is in the output directory. As with
configuration files, building the application overwrites the start 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/start.in and add the following lines at the beginning
   of the section titled “Build up classpath:”
   CLASSPATH="<JDBC_LIB>"
   export CLASSPATH

   where <JDBC_LIB> is the JDBC driver library (generally a .jar or .zip file), including
   the file path. For example:
   CLASSPATH=/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.
Connecting the application to a database

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 make utility copies this file to the output/conf directory after every build.


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

```plaintext
# Database Manager Configuration
#-------------------------------------------------------------
DatabaseManager.Databases[] = "simpleApp"
DatabaseManager.DefaultDatabase = "simpleApp"
DatabaseManager.Debug = "false"
DatabaseManager.DB.simpleApp.ClassType = "Standard"
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
```

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 Appendix A, “Database configurations,” for additional 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 This is an example of the configuration file for InstantDB. For information on using your application with other databases, see Appendix A, “Database configurations.”
Getting Started with Lutris Enhydra

Connecting the application to a database

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 `make` from the application root directory, `simpleApp`. For example:
   ```shell
cd /enhydra/myapps/simpleApp
make
```

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” on page 47. 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/books/getting-started/samples` to your data layer (that is, the `simpleApp/src/simpleapp/data` directory).

2 To ensure that `SimpleDiscQuery.java` gets compiled when you build the project, edit the make file in the `data` directory and add the name of the file to the `CLASSES` variable:
   ```
   CLASSES =   \
   SimpleDiscQuery
   ```
   There should not be spaces after the slash (`\`), just a carriage return.

3 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:
   ```java
   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:
   ```java
   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`. 

...
Connecting the application to a database

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:

4 Find the file `<enhydra_root>/doc/books/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:
  ```java
  import simpleapp.data.SimpleDiscQuery;
  import java.util.*;
  ```
- Add a member variable corresponding to the data object:
  ```java
  SimpleDiscQuery _sdq;
  ```
- 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.

5 Now, build the application from the top level, using the make command.

```bash
cd /enhydra/myapps/simpleApp
make
```

6 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
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 graphical object-relational mapping tool that generates SQL code to create tables in a database and the corresponding application code to access the tables. DODS creates code that is specific to different databases, so you don’t have to learn the nuances of each database. DODS also handles common issues such as transactions and data integrity. DODS is most useful when you are creating a database from scratch, or when you are free to modify the database schema, as explained in “Loading the schema” on page 65.

In the previous section, we created a table, LE_TUTORIAL_DISCS, for information about music selections (discs). In this section, we will build on this theme, using DODS to create a database schema and corresponding data objects that associate an “owner” with each “disc.” This theme will be further developed in Chapter 5, “DiscRack sample application.”

**Note**

You are not required to use DODS when you develop an Enhydra database application, but it does significantly simplify the process.

**Running DODS**

This section introduces DODS and gives you a preview of the database schema that we will create in the following section.

1. Start DODS by entering the following command:

   ```
   dods
   ```

   You will see the DODS graphical interface.

2. Open the DODS file for the DiscRack project by choosing File|Open and selecting `<enhydra_root>/examples/DiscRack/discRack.doml`.

   As shown in Figure 4.12, you can expand the directories in the lower left panel to see the package hierarchy for the data model.
The DODS window has three subpanels:

- At the top, the *object panel* shows the entity-relationship model. Each data object (corresponding to a table in the database) is represented by a box with a name inside. Relationships between data objects are shown by red lines between the boxes.

- In the lower left, the *package panel* displays the package hierarchy in the current data model, with a directory for each object/table in the data model, and a blue dot signifying objects in each package.

- In the lower right, the *attribute panel* shows the attributes (properties or columns) of the selected data object. Each attribute is represented by a row in the grid, with various information associated with the attribute, such as the data type or whether null values are allowed, indicated in columns of the grid.

The schema of the DiscRack database is pictured in Figure 4.12. 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 4.12** DODS with the DiscRack.doml loaded

**Figure 4.13** DiscRack object-model/schema
Creating the data layer

Now you are going to create the data layer for your application from scratch. DODS provides a means to design a data model and then generate the corresponding data-layer classes for an application that uses the Enhydra application framework. DODS also generates the SQL code to create the requisite tables in your database.

The data model in this section will be used for the DiscRack application data layer (see Chapter 5, “DiscRack sample application”) with a slight change in the package naming.

Creating the data layer with DODS consists of the following tasks:

- Defining the package hierarchy
- Defining data objects
- Generating the data layer code

Defining the package hierarchy

Defining the package hierarchy within DODS establishes the file structure for the source files that DODS generates.

1. Create a new data model by choosing File | New.
   DODS closes any open data model file and starts a new one containing a single package, named “root” by default.

2. Using the Database menu, select your database type.
   For example, if you are using Oracle, choose Database | Oracle. This ensures that the SQL statements that DODS generates have the correct syntax for your database. For this example, select Standard JDBC for InstantDB.

3. Now create the package hierarchy:
   - Change the name of the root package.
     In the lower left panel, select the root folder, then choose Edit | Package. Type the name of this application, simpleApp, in the dialog box, then press Enter.
   - Add the data package by selecting the simpleApp folder and choosing Insert | Package. Type data, then press Enter.

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

   **Figure 4.14** Package hierarchy
Defining data objects

Data objects in the DODS data model correspond to tables in the database. The attributes defined for data objects describe database columns.

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

![Data Object Editor dialog box](image)

---

2. Change the data object defaults as follows:

   - In the **Class** tab, type **Person** in the Name field.
   - In the **Package** tab, select the person package.
   - In the **DataBase** tab, type **Person** in the db Table Name field.

   Note that the class name starts with an uppercase letter (Person), while the package name is lowercase (person).

3. Leave the rest of the default values and click OK. You now have a Person data object in your data model. Later, when you generate the data layer with the Build All command, DODS will generate the code to create a Person table in the database. DODS will also create a Person object that you can incorporate into your application.

4. Select the disc package in the lower left pane, then choose **Insert | Data Object**.

5. In the Data Object Editor dialog box, change the defaults as follows:

   - In the **Class** tab, type **Disc** in the Name field.
   - In the **Package** tab, select the disc package.
   - In the **DataBase** tab, type **Disc** in the db Table Name field.

6. Leave the rest of the default values and click OK. You now have a Disc data object in your data model.

7. Add the login attribute to the Person data object:

   1. Select the Person data object (indicated by a blue dot in the simpleApp data model), and click **Insert | Attribute** to open the Attribute Editor dialog box, as shown in Figure 4.16.
2 In the General tab, type login in the Name field.

3 In the Java tab, select String from the Java Type pull-down list to set the data type. String is the default Java data type.

Note
When a Java data type is selected from the Java Type pull-down list in the Java tab, the corresponding database type is assigned for the db Type field in the Database tab. For example, when you selected String in the Java tab, the db Type field was set to VARCHAR.

4 In the Database tab, leave the db Type as VARCHAR and select Can Be Queried.

5 Leave the rest of the default values and click OK.

Notice that the login attribute appears in the attribute panel in the lower right of the DODS window (see Figure 4.12 on page 59). The fields show the different options you can set in the Attribute Editor. For example, the yellow Q symbol means the data object can be queried. Move the cursor over the other symbols to see what they represent.

8 Repeat the previous step to add the following three more attributes (with Java Type String and db Type VARCHAR) to the Person data object:

- password
- firstname
- lastname

Tip You can use the Add Attribute button on the toolbar to open the Attribute Editor dialog box.
Add the following three attributes (with Java Type String and db Type VARCHAR) to the Disc data object, as described in the previous steps:

- title
- artist
- genre

Define the owner attribute for the Disc data object.

Adding the owner attribute is a little more complicated than the previous attribute additions because there is a one-to-many relationship between Person and Disc based on the owner attribute.

1. Select the Disc data object and choose Insert | Attribute to open the Attribute Editor dialog box.
2. In the General tab, type owner in the Name field.
3. In the Java tab, select simpleapp.data.person.PersonDO from the Java Type pull-down list.
4. In the Database tab, select Can Be Queried and Referenced DO Must Exist options.
5. Click OK to close the Attribute Editor dialog box.

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

Define the isLiked attribute for the Disc data object.

1. Select the Disc data object and choose Insert | Attribute to open the Attribute Editor dialog box.
2. In the General tab, type isLiked in the Name field.
3. In the Java tab, select boolean from the Java Type pull-down list.
4. In the Database tab, select Can Be Queried.
5. Click OK to close the Attribute Editor dialog box.

You have now defined the data objects Person and Disc.

Generating the data layer code

To finish this exercise, save and build the data model.

1. Select File | Save to save the data model file.

   Save the data model as simpleApp.doml in the top-level application directory (for example, /enhydra/myapps/simpleApp).

   **Note**

   DODS stores the data model specification in a .doml file. DOML stands for Data Object Meta Language. The specification has an XML-like syntax.

   Now that you have defined all the data objects, DODS is ready to generate and compile the code. The Build All command causes DODS to overwrite the existing
Using DODS

Data directory with all the new information from the data model, so if you want to
save your old data layer code, copy it to another directory.

2 Select File|Build All to open the Destroy and Create New DODS Directory dialog box.

1 Select the data directory of your simpleApp project (for example,

Important Navigate to the directory that contains the business, data, and presentation
directories. For example, navigate to /enhydra/myapps/simpleApp/src/simpleapp
(simpleapp will be displayed in the Look In field). Single-click the data directory
to select it; do not double-click the data directory.

2 Choose the Re/Create Directory button.

You will see the following warning:

Figure 4.17 DODS Re/Create Directory warning box

3 Click Yes, and begin the build.

DODS aligns the directory structure in the GUI with the directory structure in the project, generates Java and SQL files, and then runs make to build the data layer. As
it proceeds, DODS displays messages in a dialog box. For example:

Figure 4.18 DODS Build All message dialog box

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

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 table
• create_tables.sql and drop_tables.sql files, which contain standard SQL statements to create and remove the disc and person tables from a database, respectively

• Two make files—Makefile and config.mk

• 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 make files 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.

**Loading the schema**

Figure 4.19 illustrates the complete schema generated by DODS:

![Figure 4.19 DiscRack database schema generated by DODS](image)

Notice 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 an application is updating is accurate. Because many users can be accessing the database simultaneously, a record can change between the time an application retrieves it 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.

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 `<simpleApp_root>/simpleApp/data` directory.

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

   ```sql
   /* 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/books/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:

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

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 `<enhydra_root>/dods/dods.conf`. For example, by default DODS generates C-style comments, but you can change the style of comments if your database requires a different format.

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
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/books/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
SQL> @<enhydra_root>/doc/books/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 DODDS 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/books/getting-started/samples` directory. Rename the file `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.
Using DODS

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:

```java
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.dic` 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 application

Now we will recompile and run the simpleApp application.

1. Build the application from the top level, using the make command. For example:
   ```
cd /enhydra/myapps/simpleApp
make
```

2. Start Enhydra, either by starting the Admin Console (see “Launching the Admin Console”) or by entering the following commands in the shell window.
   ```
cd output
./start
```

3. Test the application by loading http://localhost:9000 in your browser.
   The Disc List table in your Simple presentation object should now contain data from the Disc table in your database.
Figure 4.20  Simple PO with data from DODS data objects

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 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.
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 make the following modifications to the installed files:

1. Open config.local.mk.in in <DiscRack_root> and set the ENHYDRA_DIR and JDKDIR variables to your Enhydra root directory and JDK directory respectively.

2. Copy config.local.mk.in to config.local.mk in the <DiscRack_root> directory.

3. Open the application configuration template file, discRack.conf.in, in <DiscRack_root>/discRack and make sure all Database Manager configuration settings are correct. See “Configuring the Database Manager” on page 55 and Appendix A, “Database configurations.”

4. Build the application by entering the make 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 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 configuration file. Refer to Appendix A, “Database configurations,” for additional information about using Microsoft Access with your application.

5. 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 start script.
To run DiscRack, enter the following commands:

cd <DiscRack_root>/output
./start

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

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

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**
  Package 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, a user name, and a 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.

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.
Figure 5.2  DiscRack Storyboard

Welcome/Login Page

Valid Login?

Yes

No

Signup Page

Edit/Populate form with data

Edit/Add Page

Edit

Add

DiscCatalog (Main) Page

Delete

Log out

Confirm Delete (JavaScript Dialog)

Logout/Thanks Page

Welcome to DiscRack

Login

Password

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

Yes

Logout/Thanks Page

Invalid Login?

Welcome/Login Page
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

<table>
<thead>
<tr>
<th>Class or package name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>discRack package</td>
<td></td>
</tr>
<tr>
<td>DiscRack</td>
<td>Application object</td>
</tr>
<tr>
<td>DiscRackException</td>
<td>Simple base exception class</td>
</tr>
<tr>
<td><strong>Presentation layer/package</strong></td>
<td></td>
</tr>
<tr>
<td>BasePO</td>
<td>Abstract base class for all presentation objects</td>
</tr>
<tr>
<td>DiscRackSessionData</td>
<td>Container for session data</td>
</tr>
<tr>
<td>ErrorHandler</td>
<td>Class to handle exceptions not caught elsewhere in the application</td>
</tr>
<tr>
<td>DiscRackPresentationException</td>
<td>Presentation layer exception class</td>
</tr>
<tr>
<td>presentation.personMgmt</td>
<td>Package that contains the Register and Login classes for managing presentation related to the PERSON table</td>
</tr>
<tr>
<td>presentation.discMgmt</td>
<td>Package that contains the Edit and DiscCatalog classes for managing presentation related to the DISC table</td>
</tr>
<tr>
<td><strong>Business layer/package</strong></td>
<td></td>
</tr>
<tr>
<td>DiscRackBusinessException</td>
<td>Business layer exception class</td>
</tr>
<tr>
<td>business.person</td>
<td>Package that contains two classes:</td>
</tr>
<tr>
<td></td>
<td>• Person, which represents a person</td>
</tr>
<tr>
<td></td>
<td>• PersonFactory, which has a single method that returns the Person object for a user name</td>
</tr>
</tbody>
</table>

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 make files create an output directory containing the configuration files and the start script. Also, there will be a lib directory with a .jar file that contains all the 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.
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 is 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” on page 42. 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.
Chapter 5, DiscRack sample application

Presentation layer

- 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:

```java
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:

```java
...  
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`:

```
```

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:

  ```javascript
  document.location='Edit.po?event=showAddPage'
  ```

  This function is defined in `presentation/discMgmt/DiscCatalogScript.html`, not the DiscCatalog page, as explained in “Replacing JavaScript” on page 84.

- `add` event (to add a disc to the database) is defined in the `Edit.html` page by a hidden form field:

  ```html
  <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” on page 83, 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:

```java
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:

```java
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 make files in the presentation layer control how the application uses the HTML files. There are a total of three make files in the presentation layer—one in the top level and one in each subdirectory.

To keep the HTML files in a directory separate from the presentation classes, the make files use the `HTML_DIR` directive, which specifies the relative path to the directory containing the HTML files. For example, in `presentation/Makefile`, you’ll see:

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

And in `presentation/discMgmt/Makefile`:

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

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

The `HTML_CLASSES` directive indicates the names of the class files that XMLC creates, as explained in “Adding a new page to the application” on page 44.

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

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

Maintaining the storyboard

The storyboard is initially just a mockup of the application. But with a few simple steps, you can maintain 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” on page 84.

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

```
XMLC_HTML_OPTS_FILE = options.xmlc
```

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

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

When XMLC compiles the files in this directory, it 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:

```html
<SCRIPT id="DummyScript">
<!--
function doDelete()
{
    document.EditForm.action='DiscCatalog.html';
    if(confirm('Are you 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:

```html
... 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:

```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. Edit the JAR_INSTALL directive in `<DiscRack_root>/discRack/presentation/media/Makefile` by removing the two comment symbols (\#) and adding a continuation character (\) after the first line so that it looks like this:

   ```
   JAR_INSTALL = \n   ../resources/media/*.gif \n   ../resources/media/*.css \n   ../resources/media/*.jpg
   ```

   This ensures that the new .jpeg graphics files and the stylesheet file are included in the packaged application .jar file.
4. Rebuild the presentation package by entering the following commands from the directory `<DiscRack_root>/discRack/presentation`:

   ```
   make clean
   make
   ```

   The make clean command removes all the old classes so that make will completely rebuild the application from scratch.
5. 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

---

**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:

---

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

```java
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:

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

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

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

The other `OPTION` tags contain `CLASS="discardMe"`, which causes XMLC to remove those items at build time, as explained in “Removing dummy data” on page 83.

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

```java
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:

```html
<SELECT id="TitleList" Name="discID">
  <OPTION selected VALUE="invalidID">Select One</OPTION>
  <OPTION id="templateOption">Van Halen: Van Halen One</OPTION>
  <OPTION id="discardMe">Sonny and Cher: Greatest Hits</OPTION>
  <OPTION id="discardMe">Sublime: 40 oz. to Freedom</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

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

```html
<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:

```java
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):

```java
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:

```java
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):

```java
page.getElementDiscID().setValue(disc.getHandle());
page.getElementTitle().setValue(disc.getTitle());
page.getElementArtist().setValue(disc.getArtist());
page.getElementGenre().setValue(disc.getGenre());
page.getElementByIdBox().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.

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

The result is that when a user tries to edit a disc and delete some of the values, the edit page redisplay, 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.

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

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

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

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>).

Note
In general, Lutris supports the latest releases of the following databases. A comprehensive list of certified platforms is provided on the Lutris website at http://www.lutris.com/support/platform.html.

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 start script. To do this, follow these steps:

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

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

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

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

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

```java
# 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.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.

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

```java
#-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
# 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.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:

```java
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, Lutris 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” on page 65, 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>/<appName>.conf` (sometimes, by convention, `<appName>/<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"
   ```
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:

```java
# 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” on page 52. 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.DblDbType.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.

```java
# 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`.

For additional information, see Chapter 3, “Using the Multiserver Administration Console,” of the Developer’s Guide.

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:

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

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

   where `enhydra_root` is the root of your Enhydra installation. 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 Lutris Enhydra installation instructions provide information about setting your path environment variable. The installation instructions are available in HTML format only on the Lutris 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*. See Chapter 3, “Using the Multiserver Administration Console,” of the *Developer’s Guide* for information on changing these settings.

The Admin Console appears in your Web browser as shown in Figure B.1, “Admin Console display.”

**Figure B.1** 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 page 112)

**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 (see Figure B.1 on page 102).

<table>
<thead>
<tr>
<th>Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>Starts the application currently selected in the Applications window.</td>
</tr>
<tr>
<td></td>
<td>Unavailable when the selected application is already running.</td>
</tr>
<tr>
<td>Stop</td>
<td>Stops the application that is selected in the Applications window.</td>
</tr>
<tr>
<td></td>
<td>If the application has active users, you are prompted to verify that you</td>
</tr>
<tr>
<td></td>
<td>want the application stopped.</td>
</tr>
<tr>
<td>Add</td>
<td>Adds an application to the Enhydra Multiserver.</td>
</tr>
<tr>
<td>Delete</td>
<td>Removes the selected application from the Enhydra Multiserver.</td>
</tr>
<tr>
<td>Modify</td>
<td>Modifies the configurable attributes of the selected application.</td>
</tr>
<tr>
<td>Debug</td>
<td>Invokes the debugging utility for the selected application.</td>
</tr>
<tr>
<td></td>
<td>When you click this button, the debugging control panel displays.</td>
</tr>
<tr>
<td>Save State</td>
<td>Saves the state of the Enhydra Multiserver.</td>
</tr>
<tr>
<td>Multiserver Stop</td>
<td>Stops the Enhydra Multiserver.</td>
</tr>
</tbody>
</table>
Viewing status information

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 on page 102). 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

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.
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” on page 107.

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:
   ```bash
   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.
   ```bash
   #server.Classpath[] = ../classes
   server.Classpath[] = "/enhydra/myapps/simpleApp/output/archive/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

![Add New Application/Servlet dialog box](image)

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

![Add New Servlet dialog box](image)
Using the Admin Console

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” on page 112.

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

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

2 Click Create in the Connections tab section to display the Add New Connection dialog box.

Figure B.8 Add New Connection dialog box

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. For more information, see Appendix A, “Using SSL with Enhydra” of the Developer’s Guide.

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.
For Bind Address, enter an IP address if you want to bind to only one of the available IP addresses on a given machine. If no IP address is specified, the server binds the given port for all IP addresses on the machine.

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.

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.

Click OK to return to the Status window.

To start the application, make sure it is selected in the Applications window, and click the Start button.

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

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.

For more information about the available configuration parameters, see Chapter 3, “Using the Multiserver Administration Console,” of the Developer’s Guide.

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.
Creating a WAR file

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” on page 101.

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:
Creating a WAR 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 class 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

For a more in-depth discussion on using the Admin Console with WAR files, see
Chapter 3, “Using the Multiserver Administration Console” in the Developer’s Guide.
Symbols

# sign in code 85
\ characters in code 85

A

Access databases 95
accessing data 56
dynamically 47
in database tables 58
accessing DiscRack 72
Add New Application/Servlet dialog box 106, 107
adding
business objects 49
data objects 56, 61
dates 30
links 46
new pages to applications 44
presentation objects 44, 80
Admin Console. See Multiserver Administration Console
allocateConnection() method 56
allocating database connections 56
Apache servers 18
application architecture (Enhydra) 13
application configuration file 32, 33
application development tools 11
application files
generating 29
application flowchart 74
application framework 12, 17
described 19
application objects 14
application root directory 29
application server 11
See also Enhydra
application URLs 33
Application Wizard 24
creating applications with 27
overview 21, 30
running 27
applications 11
accessing database tables 58
adding pages 44
adding to Multiserver console 105
building 76, 85
configuring for JDBC 54
connecting to databases 51
counting user hits 37, 43
creating 21, 27, 58
designing 73, 74
developing 17, 24, 42, 72
installing 18
maintaining state 20
managing 19
optimizing 17
reconfiguring 110
removing dummy data from 83
removing from Multiserver 109
running 29, 32, 103
securing 20
support for large scale 7
testing 76
updating 18
viewing default URL 110
viewing information about 103
appwizard utility 21
archives (Enhydra.org mailing list) 7
arrays 47, 57
Attribute Editor dialog box 61
attribute panel (DODS) 59
attributes 23, 40
changing 103
displaying data object 59
removing 48

B

base classes 77
BasePO class 76, 77, 79
event handling with 81
bug reports 5
Build All command 63
building DiscRack 71
building Enhydra applications 27, 76, 85
builds, failing 64
business layer 17
DiscRack application 88
business logic 74
business objects 17, 88
adding 49
button events 80

C

cache 20
cascading style sheets 40
centering headings 45
changing
  attributes 103
database connections 110
HTML pages at runtime 22
checkForUserLogin() method 78, 79
class loaders 18
CLASSPATH for 33
class option 41
classes 19
  deriving for sample applications 77
  DiscRack application 76
  generating Java 22
  package mapper for 24
classes111.zip 91
CLASSPATH 33
  adding JDBC driver to 54
database connections and 91
  setting 32, 40
cloneNode() method 48
columns 59
command-line tools 21
comment symbols 85
Common Gateway Interface (CGI) 18
compiler 22
  running XML 37, 40
compiling 24
configuration files 32
  application 32, 33
  deployment and 76
  editing 32
  Multiserver configuration 32
  multiserver.conf 32
  multiserverAdmin.conf 32
  servlet.conf 32
configurations
  Web archives 112, 113
configuring
  Database Manager 55
databases 91-99
  DiscRack 71
  JDBC drivers 54
connection strings 55
connections 18, 21, 51
  allocating 56
  changing 110
  displaying status 104
  establishing JDBC 52
  setting JDBC driver for database 91
  setting parameters for 55
  support for ODBC 95
Connections tab (Multiserver) 104
contacting
  Lutris Documentation 4
  Lutris Technical Support 5
Lutris Training 6
Content Frame (Multiserver)
  described 104
continuation characters 85
Control Frame (Multiserver)
  buttons described 103
conventions
  documentation 1
  for Enhydra root directory 3
  for screen shots 3
  for URLs 3
  UNIX pathnames 3
creating
  applications 21, 27
  business objects 49
data objects 56, 61
database applications 58
database tables 47, 51, 58
  links 46
  object models 60
  presentation objects 44, 80
D
data 14
  accessing 56
  accessing dynamically 47
  getting session 78
  initializing for Session objects 14
  removing dummy 83
  replacing at runtime 40
data integrity 58
data layer 17
  designing 74
  DiscRack application 65, 89
  example for creating 60
data layer classes 88
Data Object Design Studio. See DODS
Data Object Editor dialog box 61
data object models 59
  creating 60
data objects 17, 23, 89
  associating with database schema 58
  creating 56, 61
  getting 79
data sources 17, 96
database applications 58
database configurations 91-99
  Access-specific 95
  Informix-specific 92
  InstantDB-specific 94
  InterBase-specific 96
  MS SQL Server-specific 95
  MySQL-specific 93
Oracle-specific 92
PostgreSQL-specific 94
Sybase-specific 93
Database Manager 21
  configuring 55
database schema 58
designing 74
generated for DiscRack 65
databases 23
  allocating connections for 56
  changing connections 110
  configuring 91-99
  connecting to 21, 51, 55, 91
  creating tables for 47, 51, 58
  mapping tables to 58
  populating tables for 47
dates
  adding 30
  debugging 19, 110
  debugging tool (Multiserver) 103, 110
DefaultUrl setting 33
delete-class option 83
deleting
  applications in Multiserver console 109
  attributes 48
  dummy data 83
deployment process
  defined 73
  described 76
design and storyboard process 73
designing applications 73, 74
designing HTML pages 22
developing applications 17, 24, 42, 72
development and testing process
  defined 73
  described 76
development tools 11
Director 18
directories
  needed for deployment 76
  relative paths to 82
  setting up for Web archives 112
directory mapper 24
Disc class 88
disc package 77
discardMe option 83, 86
DiscCatalog.java 86
DiscCatalogScript.html 84
DiscFactory class 89
discMgmt package 76
DiscRack application 25
  business layer 88
  data layer 60, 65, 89
  deployment process for 76
development and testing process for 76
  event handling in 80, 81
  functional specification for 74
HTML pages for 82
logging in 78, 79
overview 76
populating 85
populating forms for 87
presentation layer 77
removing dummy data 83
replacing JavaScript 84
replacing the user interface 84
requirements definition for 73
running 68, 72
schema generated for 65
setting up 71
storyboard for 75, 82
URL mapping 83
welcome page 72
DiscRack class 76
DiscRack directory 71
DiscRack login page 74
discRack package 76
discRack.mdb 71
DiscRackBusinessException class 76
DiscRackException class 76
DiscRackPresentationException class 76
DiscRackSessionData class 76, 78
displaying
  administration console 33, 102
  connection status 104
  data object models 59
  default URLs 110
  DiscRack storyboard 74
  package hierarchies 59
DIV tag 40
Document Object Model
  defined 22
  displaying 41
  overview 39
Document Object Model classes
  building 25
  mapping packages to 24
documentation 4
  conventions 1
  updates and release notes 4
documentation set 3
DODS 17
  accessing data-specific objects 67
  main window described 59
  overview 22, 58
  running generated SQL scripts 66
  starting 58
DOM. See Document Object Model
Getting Started with Lutris Enhydra

.dml files 63
downloading
   Enhydra open-source software 8
drivers 91
dump option 41
duplicate IDs 48
dynamic recompilation 22

E
edit forms 87
Edit presentation object 80
Edit.html 87
Edit.java 87
editing configuration files 32
Enhydra
   accessing source code for 7
detailed installation instructions (on CD) 9
documentation set 3
downloading open-source software 8
dynamic recompilation feature 22
HTML installation instructions 9
mailing lists 7
new features and enhancements 12
online documentation 4
online installation instructions (HTML file) 9
overview 11
prerequisites for using 1
product registration 5
reporting bugs 5
root directory conventions 3
runtime component 17
sample project for 25
support for large scale applications 7
technical support 5
training courses 6
website (Enhydra.org) 6
working groups (Enhydra.org) 7
Enhydra applications
   adding pages 44
   adding to Multiserver console 105
   architecture described 13
   building 76, 85
   configuring for JDBC 54
   connecting to databases 51
   counting user hits 37, 43
   creating 21
   defined 11
   designing 73, 74
   developing 17, 24, 42, 72
   framework described 19
   installing 18
   maintaining state 20
   managing 19
   optimizing 17
   reconfiguring 110
   removing dummy data from 83
   removing from Multiserver 109
   required objects 17
   running 29, 32, 103
   securing 20
   setting default URL for 33
testing 76
tutorial for building 27
updating 18
viewing default URL 110
viewing information about 103
Enhydra CD
detailed installation instructions 9
Enhydra Database Manager 21
Enhydra Director 18
Enhydra Presentation Manager 20
Enhydra Session Manager 20
Enhydra tools 12, 21
enhydra.jar 40
Enhydra.org 6
   community documentation 8
   free online installation instructions 9
   mailing list archives 7
   mailing lists 7
   working groups 7
   error messages (DiscRack) 78
ErrorHandler class 76
events 80, 81
executeQuery() method 56
executing generated SQL scripts 66
executing queries 56
Extensible Markup Language Compiler 24
command-line syntax for 40
overview 22
running 37, 40
eXtensible Markup Language. See XML

F
factory 88
failover 18
files 15, 23
generating application 29
needed for deployment 76
findPerson() method 89
flowchart 74
forms 87
functional specification
defined 73
described 74
functions. See methods
G

generating
  application files 29
  Java classes 22
  SQL code 58
get() method 42
getElementTime() method 31
getParentNode() method 48
getSessionKey() method 43
graphical user interfaces 74, 84
graphics files 76, 82
  including in applications 85

H

handleDefault() method 87
handleEvent() method 78, 81
headings 45
hidden form fields 80
hit counts 37
  accessing information for 43
hitCount property 38
hits property 43
HTML
  online Enhydra installation instructions 9
HTML forms 87
  manipulating content at runtime 39
HTML pages 15, 82
  adding 44
  adding dates 30
  adding links 46
  centering headings 45
  changing at runtime 22
  designing 22
  timestamping 31
HTML tables 47, 48
HTML tags
  overview 30
  removing attributes 48
HTML templates 22
HTML_CLASSES variable 46
HTML_DIR directive 82
HttpPresentation interface 15
httpPresentation package 20
HttpPresentationManager class 20
hyperlinks 83
  adding to HTML pages 46

images 76, 82
  including in applications 85
Informix databases 21
  configuring 92
initializing Session objects 14
initSessionData() method 79
inline tags 40
installation
  online instructions (HTML file) 9
installing Enhydra applications 18
instance variables 14
InstantDB databases 21
  configuring 94
InterBase databases 21
  configuring 96
InterClient JDBC driver 97
InterClient (InterBase JDBC driver) 97
interfaces 74, 84
invalid user sessions 20

J

Java class generation 22
Java Database Connectivity. See JDBC
Java Servlet API 18
Java source files 41
Java2 Enterprise Edition (J2EE) 18
JavaScript 84
JBuilder
  developing applications with 24
JDBC 21
  JTurbo driver 95
JDBC classes 54
JDBC compatibility 51
JDBC configurations 54, 91
JDBC connections 52
JDBC-ODBC bridge 95
JDeveloper
  developing applications with 24
JServ 18
JTurbo JDBC driver 95

K

keep option 41
Kelp tools 22, 24

L

large scale applications 7
launching
  Multiserver Administration Console 101
lib directory 76, 91
libraries 76
Lifetime setting 33
links 83
  - adding to HTML pages 46
list boxes 85
load balancing 18
load distribution 18
loading SQL scripts 66
locating sessions 20
log files 32
loggedInUserRequired() method 78, 79
logical database (defined) 21
login attributes 61
Login presentation object 80
logins 20, 78
  - checking for valid 79
    DiscRack application 78, 79
Lutris documentation 4
Lutris technical support 5

M
mailing lists (Enhydra.org) 7
mailing lists Enhydra.org
  - archives 7
maintaining storyboards 82, 84
maintaining user sessions 43
maintenance 17
make 32
make clean command 85
make command 29
make files 24, 29, 65
  - usage example 82
managing applications 19
managing presentation objects 20
mapping
  - packages to directories 24
  - tables to databases 58
  - URLs to presentation objects 20, 83
mapping table 24
mapping tool 58
MaxIdleTime setting 33
methods 14, 40
Microsoft Access databases 95
Microsoft servers 18
Microsoft SQL Server databases 21, 95
Modify Configuration window 110
modifying
  - attributes 103
  - database connections 110
mouse events 80
MS SQL Server databases 95
Multiserver
  - load distribution with Director 18

overview 17
stopping 46
Multiserver Administration Console 19
  - adding/deleting applications 105, 109
  - button descriptions 103
  - configuration file 32
  - debugging with 110
  - display described 103, 104
  - launching 101
  - overview 101
  - reconfiguring applications 110
  - saving state 111
  - starting 101
  - stopping 103
multiserver command 101
Multiserver configuration file 32
multiserver.conf 32
multiserverAdmin.conf 32
MySQL databases
  - configuring 93

N
Netscape servers 18
new features 12
newapp. See appwizard

O
object models 59
  - creating 60
object panel (DODS) 59
objects 17, 88
ODBC connections 95
online
  - documentation 3, 4
  - installation instructions (HTML file) 9
  - registration 5
open source databases 93, 94
options 25
options.xmlc 83
Oracle databases 21
  - configuring 92
Oracle JDBC driver 91
output directory 32, 76
owner attribute 63

P
package names 60
package panel (DODS) 59
packages
  - DiscRack application 76
mapping 24
setting up hierarchy for 60
viewing hierarchy 59
parameters 78
passwords 20
Multiserver 34, 102
pathnames
UNIX 3
paths 33
persistent data sources 17
Person class 88
person package 76
PersonFactory class 89
personMgmt package 76
.po files 15
populating
forms 87
list boxes 85
tables 47
ports 32
PostgreSQL databases 21
configuring 94
presentation layer 17
DiscRack application 77
presentation logic 74
Presentation Manager 20
presentation objects 15
accessing same session 78
creating 44, 80
deriving class names for 33
managing 20
mapping URLs to 83
PresentationPrefix setting 33
product registration 5
programming tutorial 42
properties 14
displaying data object 59
property pages 25
queries 21, 23, 89
    generating statements for SQL 58
    loading SQL scripts for 66
    running 56
query() method 56

rebuilding applications 85
reconfiguring applications/servlets 110
reference attributes 23
reflect package 81
reflection 81
registering Enhydra 5
relative paths 82
release notes 4
removeAttribute() method 48
removeChild() method 48
removing
    applications in Multiserver console 109
    attributes 48
dummy data 83
replacing
    JavaScript 84
    URLs at runtime 84
    user interfaces 84
reporting bugs 5
requestPreprocessor() method 14
requests 15, 18, 20
    identifying user sessions for 42
requirements definition
    defined 73
    described 73
resources 85
resources directory 74, 82
resources_finished directory 84
response objects 15
result sets 56
round-robin distribution 18
run() method 15, 20
    usage example for 77
running
    applications 29, 32, 103
    DiscRack 72
generated SQL scripts 66
Multiserver Administration Console 101
queries 56
sample applications 68
runtime component (Enhydra) 17

S
sample project 25
running 68
saving Multiserver state 111
scalability 18
schema. See database schema
screen shot conventions 3
security 20
SELECT tag 85
server class 19
servers 11, 17, 18
    establishing connections to 52
Servlet API 18
servlet runners 18
servlet.conf 32
servlets
  adding to Multiserver console 105
defined 17
  presentation objects vs. 15
reconfiguring from Multiserver 109
viewing information about 103
session affinity 18
session data 78
session idle time 33
session IDs 20
session keys 42
gaining 43
Session Manager 20
  changing parameters for 110
Session objects 20
  initializing data structure 14
session package 20
SessionData class 42
sessions
  defined 20
  getting data objects for 79
  getting data specific to 78
  locating 20
  maintaining 43
  maintaining state 42
  storing information for 78
  timing out 20
set() method 42
setTextHitCount() 39
setting connection parameters 55
shell scripts 24
simpleApp directory 31
SimpleDiscQuery.java 56
site map 74
Sonny and Cher 83
source code, accessing 8
source directory 29
source files 23, 41
SPAN tag 30, 40
sql package 21
SQL queries 23, 89
  generating statements for 58
  running 56
SQL scripts 66
StandardApplication class 14
StandardDatabaseManager class 21
StandardSessionManager class 20
start script 32, 76
starting
  Application Wizard 27
  Data Object Design Studio 58
  Enhydra applications 29, 32, 103
  Multiserver Administration Console 101
startup functions 14
startup() method 14
state 20, 21
  maintaining session 42
  saving Multiserver 111
stdrules.mk 46
stopping applications in Multiserver 103
stopping Multiserver 46, 103
storyboards 73
  maintaining 82, 84
Structured Delivery Process (SDP) 73
stylesheets 76
  example 84
  including in applications 85
submitting bug reports 5
subscribing to Enhydra mailing lists 7
support 5
Sybase databases 21
  configuring 93
system CLASSPATH 91

T

tables 23
  accessing 58
  creating 47, 51, 58
  getting document object references for 48
  mapping to databases 58
  populating 47
TCP ports 32
technical support 5
template directory 29
templates 22
testing applications 76
text, replacing at runtime 40
timestamps 31
timing out 20
tools 12, 21
training courses 6
transaction patch 93
transactions 21, 58
tutorial 27
typographical conventions 1

U

Uniform Resource Locators. See URLs
UNIX pathnames 3
updating applications 3
URL
  conventions 3
urlmapping option 83
URLs 18
  displaying default 110
mapping to presentation objects 20, 83
obtaining paths from 33
replacing at runtime 84
setting default application 33
user interface 74, 84
user names (Multiserver) 34, 102
user sessions
See also sessions
   accessing 42
   viewing length of 33
users 20
   getting number of hits for 37, 43

V
valid login scripts 79
Vectors 56
viewing
   administration console 33, 102
   connection status 104
   data object models 59
   default URLs 110
   DiscRack storyboard 74
   package hierarchies 59

W
WAR. See Web archives
Web applications 11, 19
   adding pages 44
   building 76, 85
   counting user hits 37, 43
   creating 21, 27
   designing 73, 74
   developing 17, 24, 42, 72
   installing 18
   maintaining state 20
   managing 19
   optimizing 17
   reconfiguring 110
   removing from Multiserver 109
   running 29, 32, 103
   securing 20
   setting default URL for 33
   testing 76
   updating 18
   viewing default URL 110
   viewing information about 103
Web archives
   configuring 112, 113
Web browsers 19
Web servers 17, 18
website (Enhydra) 6
weighted round-robin distribution 18
working groups
   Enhydra.org 7
World Wide Web. See Web
wrapper classes 88

X
XML compiler
   command-line syntax for 40
   overview 22
   running 37, 40
XML Compiler wizard 24
XML documents 22
XMLC option files 25
XMLC property pages 25