Enhydra 5.1 Application Architecture

Table of Contents

List of Examples

Chapter 1. Introduction

This document describes Enhydra applications. The intended audience is software developers who are writing their own Enhydra application. This document does not cover the internal workings of the Enhydra Multiserver or the Enhydra 5.1 Application Framework. This document describes the architecture of the classes you must write to create an Enhydra application.

The Enhydra Application Framework provides all the infrastructure of a Web application, but none of the content. You simply add an application object and a set of presentation objects, and you have a complete Web application. The Enhydra tools help you to create and compile your application. After compiling, the result is a single jar file that is a Servlet. This Servlet may be served to the Net by the Enhydra Multiserver, or it may be installed in any Servlet-capable Web server.

Chapter 2. What is an Enhydra Application?

A good analogy is to think of the Enhydra Application Framework as an abstract class in Java. An abstract class specifies a list of methods, defines variables, sets constants, and provides default implementations of some functions. But you cannot instantiate it. In order to use it you must create a new class that extends the abstract class. This class that you write has certain responsibilities (the methods you must write). Once your class provides the missing pieces, you have a usable Java class.

The Enhydra Application Framework is like an abstract class in that it does most of the work, and it specifies what pieces you must provide. But rather than being a single class, it is a collection of cooperating classes. These classes are implemented into a Servlet which is an almost complete Web application. Similar to how an abstract class can not be instantiated, the Enhydra Application Framework does not implement a complete, usable application. It has two places where you need to provide Java classes. Your classes fill in the gaps, resulting in a complete application.

You must provide one application object, and a set of presentation objects (one for each URL you want in your application).

You are strongly urged to run the Application Wizard (bin/appwizard) to create a sample Enhydra application so you can see how applications are laid out. AppWizard creates Java code that implements an application object and two presentation objects. Throughout this document examples will refer to the files AppWizard creates. See the AppWizard documentation (html [../developer/app_wizard.html] , pdf [../developer/app_wizard.pdf]) for details on how to use it.

Chapter 3. The Application Object

The application object is the central hub of your application. It is responsible for holding all application-wide data needed by the framework. Some examples of this data are the

- current running status (started/stopped/dead), the
- config file used to initialize the application, the
- name of the application, the
- log channel to use for logging,
- pointers to parts of the framework (session manager, database manager), and to provide a
- preprocessor function used on all requests.

Applications typically do three things to the application object:

- read settings from the config file at startup,
- extend the preprocessor (to check for HTTP basic authorization, for example), and
- store application-wide data structures.

Aside from the preprocessor function, which is optional, application objects do not deal with HTML, handle requests or otherwise talk to the net. This is the job of the presentation objects.

Note: your application object must implement the interface com.lutris.appserver.server.Application. We recommend you extend the class com.lutris.appserver.server.StandardApplication.

In your application's config file, the key Server.AppClass tells the framework which class to use as the application object.

Most Web-based applications consist of three fundamental types of components: presentation objects, business objects and data objects.

Chapter 4. The Presentation Object

Presentation Object contains the logic that presents information to an external source and obtains input from that source. The presentation logic generally provides menus of options to allow the user to navigate through the different parts of the application, and it manipulates the input and output fields on the display device. Frequently the presentation component also performs a limited amount of input data validation.

You must provide a presentation object for each URL (page) in your application. The presentation object is given the request in object form and is responsible for servicing the request. It is also given a response object to use to write data in response (very similar to the Servlet API's service() method). Almost all presentation objects handle GET requests and respond by writing HTML to the Net, but they may also, for example, read in files sent by a POST request, or send Java serialized objects to an Applet.

Presentation objects were given that name because they are in charge of how your application's data and results are presented to the client (the Web browser). See the section "Recommends" below for more details.

When a request is sent to your application, the framework receives it (through the Servlet interface), and determines which presentation object to pass the request on to. The presentation object is then located (via the class loader), instantiated and called. All this happens automatically. Presentation objects do not need to notify the framework that they exist. The framework, driven by the URLs it is asked for, will attempt to locate the corresponding Java classes.

Only requests for URLs that end in ".po" (presentation object) will result in calling a presentation object. All other types of requests are handled by serving a static file, just like a normal Web server. This lets you mix dynamic content pages (.po) in with normal html files (.html) and images (.gif, .jpeg). Rather than being files on a disk, these are typically archived into the application's jar file. The class loader is used to read the files (getResourceAsStream()). This lets you archive your entire application, pictures and all, into a single jar file. If you run "bin/AppWizard MyApp", an example of this is the file myApp/presentation/media/enhydra.gif. When a request for the URL media/enhydra.gif arrives, the framework automatically reads the file and writes it to the net, without bothering the application.

If you run "bin/appwizard MyApp", then the files myApp/presentation/Welcome.html and myApp/presentation/Redirect.java are the two presentation objects for the application. The application only has two presentation objects: the URLs Welcome.po and Redirect.po are the only URLs the application responds to. All other URLs result in the standard file not found response (an optional special case is the URL /. See the config file MyApp.conf for an explanation).

In the application's config file, the key Server.PresentationPrefix specifies the root of the Java packages containing the presentation objects (with / instead of . separators). Suppose that the presentation prefix is "application/foo/bar". Then suppose a request arrives for the URL "login/remote/Login.po". The framework will put these two together resulting in the class name "application.foo.bar.login.remote.Login". The framework will then attempt to load this class. If it is found, the request is sent to it. If it is not found a file not found response is returned.

Note: presentation objects must implement the interface com.lutris.appserver.server.httpPresentation.HttpPresentation.

Because most presentation objects emit dynamic HTML, they can be written using Document Object Model (DOM) access to the HTML (see Writing Presentation Objects with XMLC below) or they can be created by the Enhydra JDDI compiler. They can also be written by hand, by simply writing a Java class that implements HttpPresentation.

Chapter 5. Writing Presentation Objects with XMLC

The XML Compiler (XMLC) reads normal HTML files and creates Java classes that contain and represent the exact same HTML content. The Document Object Model (DOM) is used to provide access methods to read and modify the content. You would then write a "by hand" presentation object which uses the XMLC generated classes as a library.

In the HTML, you add "ID=Name" to whatever tag you want access to.

Example 5.1.

```
<B ID=FirstName>John</B>

<I ID=LastName>Doe</I>
```

The resulting class will have getFirstName(), setFirstName(), getLastName() and setLastName() classes. You simply call the set method, then call the class's toHtml() method to get the HTML for the entire page (which you would then write out to the Net). If you want DOM access to the whole page, simply add an ID field to the <BODY> tag.

We have found that for large projects is it very important to minimize the interference between the software engineers and the graphic designers, who use HTML tools that have a tendency to reformat entire files. The small amount of extra work to coordinate with designers in the beginning of the project will be paid back many times over at the end of the project when the engineers can fix bugs while the graphic artists redo the entire style of the site, and the two changes will not interfere with each other.

DOM allows access to XML files. So the XML Compiler will also allow for presentation objects to serve dynamic content XML.

Chapter 6. Writing Presentation Objects Using JDDI

Again, most presentation objects are created using Enhydra JDDI. This is because most presentation objects need to emit dynamic content HTML, and Enhydra JDDI is the easiest way to accomplish this.

You create a ".jhtml" file. You may start by simply renaming a plain HTML file. Then, by adding special <JDDI> tags, you add Java code to the file. As a timesaving shortcut you may also use "JDDI fields" in the HTML. You set their values in Java code, then in the HTML you simply put (@fieldName@), and the Enhydra JDDI field will be replaced with it's value. It's a way to avoid having to use <JDDI> tags every place you want dynamic content.

Most JDDI fields are set in your Java code, however a few are automatically created for you. For each cgi-bin argument to your page, an Enhydra JDDI field is initialized. Suppose the query string "?name=Andy" is appended to the URL to your presentation object. Then you could access this value by using the Enhydra JDDI field (@cgiArgs.name:Smith@). All the names will start with cgiArgs. Only the values are created, nothing happens if you do not add the Enhydra JDDI field to your HTML. In the previous example, the ":Smith" is used to provide default text. This prevents an error from occuring if no query string is sent to the URL.

Please refer to the Enhydra JDDI documentation (html [../legacy_documentation/jddi_syntax.html] , pdf [../legacy_documentation/jddi_syntax.pdf]) for more details.

At compile time, Enhydra JDDI reads in the .jhtml file and outputs a temporary Java file. This Java file is then compiled, resulting in a class file. The temporary Java file is then deleted. The temporary file is a Java class that implements HttpPresentation: you have simply used JDDI to do the work of coding up a presentation object for you. But the resulting presentation objects are not special: the framework does not know how the presentation object was created.

Chapter 7. Writing Presentation Objects Using Java

At times you will have a presentation object that does not emit HTML, or needs to do a lot of computation in Java. An example of this is if you have a login screen (with an HTML form) that sends the results to a presentation object LoginProcessor.po. This presentation object examines the username and password and decides if the user is allowed to log in. If they are, the user's session object is marked as logged in, and an HTTP redirect to Main.po is returned. If they aren't, an HTTP redirect back to Login.po is returned. So either way the presentation object does not need to handle any HTML. In this case you would simply write a Java class to do the work. By providing a run() method, it meets the requirements to implement HttpPresentation.

A good rule of thumb is: if you have a lot more code than content, you may want to simply write the class yourself. The framework does not differentiate between presentation objects: anything an Enhydra JDDI presentation object can do a "by hand" presentation object can do (i.e., the framework doesn't know how the class was made).

Chapter 8. The Business Object

Business Objects contain the application logic that governs the business function and process. Business objects are invoked either by a presentation component when a user requests an option, or by another business function.

The business functions generally perform some type of data manipulation.

Chapter 9. The Data Object

Data Objects contain the logic that interfaces with a data storage system, such as database systems or hierarchical file systems, or with some other type of external data source such as a data feed or an external application system.

Business objects invoke Data Objects to save persistent state.

Chapter 10. Recommends

Through years of experience, we strongly recommends that you break your application up into three categories of objects: presentation objects, business objects, and data objects. Presentation objects handle how the application data is presented to the clients (Web browsers). Any and all HTML is kept in the presentation objects. Data objects get and set the data your application manipulates, from a file, a database or even a hard-coded list. All database code, or file reading code, is kept in the data objects. Business objects handle all the "business logic". All the policy decisions, algorithms and data manipulation. Essentially everything left over after you quarantine all the HTML (and HTTP) to the presentation objects, and quarantine all the database/file access code to the data objects.

Designing your application this way minimizes the impact on your application when you switch databases, file formats or URL layouts. The Enhydra Application Framework only requires that you use presentation objects. The business and data classes you create are up to you. When you run AppWizard, it creates directories business and data for you to add these objects.

An additional benefit of designing your application this way is that you will be able to use the Data Object Design Studio (DODS) (html [../developer/index_dods.html] , pdf [../developer/index_dods.pdf]) to create your data objects for you. This tool will be released very soon (it is in final testing). You graphically create your objects, and it creates both Java code that implements them (with get and set methods that talk SQL to the database), and the SQL code to create the required tables in your database.

Chapter 11. Presentation Manager

The presentation manager handles the loading and execution of presentation objects in the context of an Enhydra application. The presentation manager transforms the name of the presentation object into a URL, uses the specified class loader to load the presentation object, and then executes the presentation object by executing its run() method. There is one instance of a presentation manager per Enhydra application. The presentation manager can also cache presentation objects in memory and any associated files that are part of the application.

It uses the Enhydra class loader to get these other files.

The presentation manager is contained within an instance of httpPresentationServlet class. This design allows a servlet to have multiple presentation objects managed by the presentation manager. The presentation manager also manages the resources necessary to execute the presentation objects.

The presentation manager also provides the key with which the session manager uses to locate a session. The key either is a cookie or generated by URL rewriting.

Chapter 12. The Session Manager

The SessionManager [../user-doc/com/lutris/appserver/server/session/SessionManager.html] is one part of the Enhydra Application Framework. The framework automatically issues users a cookie and creates a Session [../user-doc/com/lutris/appserver/server/session/Session.html] object. The cookie is a secure, opaque identifier. All the application data is kept safely inside the application. Every request the user makes, when it is given to the presentation objects, contains the user's session object. Inside the session object is a SessionData [../user-doc/com/lutris/appserver/server/session/SessionData.html] object. This is a totally flexible container for whatever the application wants to store.

The session manager and session objects provide a flexible, lightweight mechanism that enables stateful programming on the Web. Enhydra provides a general implementation of session management that serves as a basis for more sophisticated state models (a session is a series of requests from the same user that occur during a time-period). Session management gives servlets and other server-side applications the ability to keep state about a user as the user moves through the application. Enhydra maintains user state by creating a Session object for each user. These Session objects are stored and maintained on the server. When a user first makes a request to an application, the user the session manager assigns a new Session object and a unique session ID. The session ID matches the user with the Session object in subsequent requests. The Session object is then passed as part of the request to the servlets that handle the request. Servlets can add information to Session objects or read information from them. After the user has been idle for more than a certain period, the user's session becomes invalid and the Session Manager destroys the corresponding Session object.

Any data that is application-wide (shared across all pages and all users) should be kept in the application object. Any data that is user-wide (needed across all URLs a user goes to, but one copy per user) should be kept in SessionData. Any data that is page-specific (only needed by one page, but shared across all accesses to the page) should be stored as static fields of the presentation object.

Chapter 13. The Database Manager

The Database Manager is another (optional) part of the Enhydra Application Framework. An application may create an instance of theStandardDatabaseManager class by specifying keys in the application config file.

The database is the component that manages a pool of connections across any number of logical databases, allowing for much faster database access. SQL queries are also cached, again allowing for faster database access.

Logical databases hide the nuances across different JDBC and database implementation and contain a set of connections. These connections are shared across thread boundaries. The connections are responsible for maintaining a connection to a JDBC driver and the state of a database connection, including the current statement(s) and result set(s) that are in progress.

Chapter 14. The Administration Manager

The administration manager is graphical tool that allows a system manager to configure and monitor an instance of Enhydra and associated applications. All configuration information for Enhydra and Enhydra applications is stored in configuration files. When Enhydra starts, it reads these configurations files and starts the server process and any specified applications. Once the instance of Enhydra is running, the administration manager is able to perform management operations. All management operations work in the same manner; the active state (including resource parameters) of Enhydra, an application, or servlet is changed and the change may be saved in the configuration file.

The following are the management operations performed by the administration manager.

- Start/Stop applications or servlets that are currently executing in an Enhydra installation.
- Add/Remove an application or servlet from an Enhydra installation.
- Modify the operational attributes for Enhydra, an application or servlet.
- Check Status on the active state of Enhydra, an application or servlet. This includes properties like the number of active applications, the default time for a session, and the size of the database connection pool.