Custom SiteSearch Access Authorization

System Integration at the University of Chicago Library

Tod A. Olson <tod@uchicago.edu>

1. Introduction

The University of Chicago Library uses the the WebZ component of SiteSearch to access the Committee on Institutional Cooperation (CIC) Virtual Electronic Library (VEL). Its primary use at our site is to facilitate patron-initiated interlibary loan. In a previous version of this product, we replaced the vendor-supplied access mechanism, a flat-file database, using WebZ's API to authenticate against our production library system. With the release of SiteSearch 4.0, the API had changed completely; it was available only to Java (we had no Java programmers at the time), was more complicated (requiring more work to understand and use), and the default access database was now a full featured relational database. Even so, we again replaced the default access mechanism, authenticating directly against our production library system.

Information on all University of Chicago SiteSearch users (all faculty, students and staff) exists and is maintained in the Library's integrated library system, Ameritech Library Systems' Horizon. Maintaining the default SiteSearch access database would require double storage of the patron data, building an export/import mechanism, and adding regular reloading to the production schedule. Authentication against Horizon directly is preferable to maintaining a second database. Since Horizon runs under Sybase Adaptive Enterprise server and we have the freedom to add tables and otherwise customize Horizon, additional SiteSearch support can be added if needed.

Arranging for SiteSearch to authenticate against our local Horizon database requires writing Java classes to implement the AccessServerConnect and AccessClient interfaces. This paper assumes familiarity with a SiteSearch 4.0 installation and access to the accompanying javadoc.

2. Principles

The following principles guided implementation:

- *Separate customizations from the distribution as much as possible.* Use local .ini files, HTML files and Java classes and configure the JaSSI to use them. This should make upgrades easier, as the upgrade will overwrite fewer customizations. Ideally, only the JaSSIServer.ini would need to be manually edited after an upgrade. From the outset, one of the most important principles was to separate local customizations from the SiteSearch distribution files as much as possible. After an upgrade, limiting the manual editing to as few files as possible is ideal.
- *Hide database interaction under an abstraction layer.* The AccessClient code is sufficiently complicated without littering it with JDBC code. A decent abstraction will permit reuse on other projects. A pleasant side effect is that this veneer gives compile time type checking on the column types.
- *Test the pieces individually.* With a `main()` method in each class to exercise its methods, the debugging cycle is much shorter. The implementor can do much debugging without having to install the class, bounce JaSSI, probably log in and then slog through competing data in the logs.
- *Avoid touching the user interface.* Though the AccessClient supplies information, particularly patron information, to the user interface, it should not force changes to the user interface.
- *Code should support policy, not set it.* Policy decisions should be reflected in the configuration files to the greatest possible extent.

3. Implementation

What follows is an outline of how the AccessClient was implemented. It is something of a revisionist history, in that some events have been reordered to read more easily. This may also serve as a model for sites interested in similar development.

3.1. Preparation

Before implementing, decisions were made about the order in which to implement the Java classes, for what exactly each class would be resposible, where the custom classes and .ini files would be installed, and how to manage ongoing changes. A review of the vendor-supplied javadoc, especially for the AccessServerConnect and the AccessClient interfaces, and of the .ini file informed these decisions.

The database abstraction was implemented first. It is reponsible for hiding the particulars of JDBC interacting with our database, yet gives general access to the database. The HorizonAccessServer class, which implements AccessServerConnect, was second. It reads configuration information from the .ini files and passes it to HorizonAccessClient. Last was the AccessClient, which retrieves from HorizonAccessServer various configuration data and performs the actual database interaction: it loads the JDBC driver, connects to the database, performs user authorization and stores patron information .

In planning to keep local customizations separate from the distribution, one must consider local Java classes, .ini files and (though not in this paper) HTML files. Local classes go in their own packages. The database abstraction classes are installed outside of SiteSearch entirely, as is the JDBC distribution, since they may be used by other projects. The AccessServerConnect and AccessClient live in the `edu.uchicago.lib.webz` package, which is installed under the SiteSearch classes directory. .ini files and HTML files have less flexible lookup mechanisms, so local files are distinguished by a standard prefix, "UChi".

Currently, all changes are managed by the RCS revision control package. In the future CVS will replace RCS in this project.

3.2. Database abstraction

Since SiteSearch customization is done in Java and Horizon runs on top of Sybase, the interaction with Horizon is performed via JDBC using Sybase's jConnect product. A database abstraction is used to keep the details of the JDBC interaction out of the custom SiteSearch source code. Such an abstraction may be tailored to the application or be quite general. A general approach was adopted, with an eye towards code reuse in other projects. The abstraction as implemented is no more than a thin veneer over JDBC, hiding the details of JDBC but exposing the table organization at the row and column level. A commercial code-generator would likely have automatically produced a suitable abstraction layer, but the small amount of abstraction code required by this project would not have justified the expense.

Horizon table borrower_phone:

Column:	Type:
`borrower#`	`int`
`ord`	`tinyint`
`phone_no`	`varchar`
`phone_type`	`varchar`

Index name:	Index Keys:
`borrower_phone_index`	`borrower#, ord`

Java class BorrowerPhone:

```
public int getBorrowerNum()
public byte getOrd()
public String getPhoneNo()
public String getPhoneType()

public boolean getByBorrowerNumAndOrd(
                  int borrowerNum, byte ord)
public boolean next()
```

Figure 1

The database itself is represented by the Horizon class, which stores the JDBC Connection and hides the process of connecting to the database behind an `open()` method. A Horizon object is passed to other abstraction objects at object instantiation time so the abstraction objects have access to the Connection. Stored procedures that return minimal results are coded as methods.

Each table in the database is represented by a class. *Figure 1* shows the Horizon borrower_phone table and the Java BorrowerPhone class that gives it an object representation. Every table class has a method to query by primary key, a `next()` method to grab a row of the result set, and one method per column to return that column's value for the current row. There is a query method for each index which takes index keys for arguments. The user uses `next()` to iterate through the rows of the result set.

This is an extremely simple, general query-only veneer over the database. It could easily be extended to permit updates. It could be extended to permit the user to specify which columns are of interest or to provide access to SQL "where" clauses, probably with the help of a column object. Currently, and perhaps even with the above extensions, it would be straightforward to implement a code-generator for this abstraction. It would be interesting to compare these ideas with abstractions provided by commercial code-generators.

3.3. AccessServerConnect

HorizonAccessServer implements the AccessServerConnect interface. It reads the .ini files and stores the information for HorizonAccessClient. The .ini files mimic OCLC's AccessServer organization.

Within the JaSSI, the access server .ini files (see *Figure 2*) are proccessed as follows: JaSSI reads the ServerMngr section of JaSSIServer.ini. It sees UChiAccessServer listed and looks for a UChiAccessServer section. In that section, it looks for an InitClass, which it will instantiate. It will then call that object's `init()` method with the server name, UChiAccessServer, and the JaSSIServer.ini (as an IniFile object). `init()` looks up two .ini files. First it looks up UChiAccessServer.ini (the server name followed by ".ini") and reads in JDBC connection information. Then it looks in the JaSSIServer.ini UChiAccessServer section for "inifile", here listed as "server/UChiAccess.ini". `init()` looks here for information that the JaSSI will want. (See the javadoc for the AccessServerConnect interface.) Note that this information is split into two files to mimic the OCLC AccessServer.ini and servers/Access.ini organization; it could have all gone into one .ini file just as easily. The data that HorizonAccessServer expects to find in its .ini files is documented in the class's javadoc.

Implementation started with a skeleton class in which all methods were stubs which returned default values (null, 0, false). Included was a `main()` method to set up the log, etc., and a `toString()` method to assist in debugging (see *Listing 1*). The initial stub methods logged entering and exiting the methods. Once methods do much more, it may be prudent to catch NullPointerException and log such errors. The `toString()` method uses reflection to report on all fields. This is particularly useful when the fields contain simple data, rather than objects or arrays. `main()` mimics the environment this class will run under: it sets up ServerLog.out (used as the default log by HorizonAccessServer) and sets IniFile.rootDir so the IniFile class can find JaSSIServer.ini. `main()` then calls `init()` as described earlier and outputs the resulting state via `toString()`, which is called implicitly.

`init()` is the heart of HorizonAccessServer (see *Listing 2*). It understands the organization of the .ini files and makes that information available to HorizonAccessClient objects. `init()` expects the `name` argument to be the server name and the IniFile `ini` argument to be JaSSIServer.ini. It looks in `name+".ini"` for JDBC information and it looks up JaSSI information in the file named in `ini`'s section `name`, key "inifile" (again, see *Figure 2*). If "inifile" is not present, `init()` assumes that the JaSSI data will be found in `name+".ini"`. `init()` also uses helper methods (not listed) to actually extract data from the .ini files.

```
JaSSIServer.ini:

[ServerMngr]
server1 = ZBase
server2 = UChiAccessServer
server3 = ILLServer

[UChiAccessServer]
inifile        = servers/UChiAccess.ini
InitClass      = edu.uchicago.lib.webz.HorizonAccessServer
sectionName    = JaSSI

UChiAccessServer.ini:

[JDBC]
SQLDriver=com.sybase.jdbc.SybDriver
URL=jdbc:sybase:Tds:horizon-prod-1.uchicago.edu:2025/horizon
Login=[ . . . ]
Password=[ . . . ]

servers/UChiAccess.ini:

[JaSSI]
CheckNameAuthos = true
CheckIPAuthos   = false
CheckDNSAuthos  = false
clientClass=edu.uchicago.lib.webz.HorizonAccessClient
```

Figure 2

The AccessServerConnect interface specifies methods that make available the JaSSI information in servers/UChiAccess.ini, the `authorizationBy{Name|IP|DNS}Required()` methods. It does not specify such methods for the JDBC data in UChiAccessServer.ini. The interface does inherit `hostName()`, `serverName()`, `serverPort()` and `serverType()` methods from the ServerConnect interface. This being the case, HorizonAccessServer only implements these ServerConnect methods as stubs, and provides the JDBC data through `serverSQLDriver()`, `serverURL()`, `serverLogin()` and `serverPassword()` methods. There is nothing Horizon-specific about HorizonAccessServer; it could be used by any AccessClient that needs the same JDBC data.

3.4. AccessClient

An AccessClient implements a site's access policy. It authenticates a user and determines what databases are available to the user. A new AccessClient is instantiated for each SiteSearch session.

Setting JaSSI session logging to TRACE_ENTERS_AND_EXITS and walking through a short session in the default AccessClient revealed the order in which the AccessClient methods are called. The first method called is `initialize()`, followed by `setLog()` and `setDbList()`. After a successful `authorizeByName()`, JaSSI calls `getDbList()`. This information helped guide the implementation.

Implementation of HorizonAccessServer began with most methods implemented as stubs only, a `toString()` to report state, and a `main()` to test the other methods (see *Listing 3*). `main()` has to emulate more of the JaSSI in order to run an AccessClient. It sets up `ServerLog.out`, as before. Conveniently, once this is done `UserLog.out` automatically writes to the same file. This is useful for debugging, as all information can be logged to the same file when testing. `main()` then generates a UserStateObject. Notice that the static method `UserStateObject.setCapacity()` is called before instantiating the UserStateObject. This requirement is not documented, but if omitted, `new UserStateObject()` will throw an IllegalArgumentException. `main()` then properly initializes a HorizonAccessServer object. The HorizonAccessServer and UserStateObject are then used as arguments to `initialize()`.

`setLog()` was trivial: it stores its Log argument in a private Log field called `log`, which defaults to `ServerLog.out`. All logging in the entire class uses `log`.

`initialize()` was the next method to implement (see *Listing 4*). `initialize()` loads the JDBC driver and opens a connection to the database using the JDBC information from HorizonAccessServer. Its type signature expects two Objects. SiteSearch technical support said that `initialize()` receives an AccessServerConnect object (an instance of HorizonAccessServer) and a UserStateObject. More precisely, the first argument will be an instance of the class listed in InitClass (see *Figure 1*), but requiring it to implement AccessServerConnect would be quite reasonable: a class that doesn’t implement AccessServer-Connect would likely generate an IllegalArgument-Exception in the JaSSI soon after instantiantion.

Before continuing, a basic understanding of `get()`, `set()` and `delete()` is useful. These methods are very much like the Hashtable methods `get()`, `put()` and `remove()` to look up, store and remove information, except that the HorizonAccessClient methods take an additional "resource object" argument. This suggests that the resource objects in fact be Hashtables. To implement these, it is useful to know what "resource objects" are expected. illformscreen.java and isoillscreen.java demonstrate this, with uses like these:

```
accessClient.get("illauthos", "inst")
accessClient.get("illpwd", "inst")
accessClient.get("clientname", "authos")
accessClient.get("clientid", "authos")
```

This suggests that an AccessClient might want to use two hashtables, one of patron data ("authos") and one of institutional data ("inst"). HorizonAccessClient uses this strategy.

Default institutional data is already provided by IllService.ini. The AccessClient lookup mechanism provides a hook for a site to override these institutional defaults on a user-by-user basis, if desired. For example, an ILL department might have a $20 maximum authorized cost per item, but raise that to $30 for faculty.

The authorization methods were implemented next. The only one of interest is `authorizeByName()` (see *Listing 5*). At the University of Chicago, every patron has a barcode on his school ID card and a PIN associated with it in Horizon (this is a local extension). These are passed to `authorizeByName()` as the authos and password arguments. `authorizeByName()` checks Horizon to see whether the barcode and PIN match, whether the patron’s privileges are current, and whether the patron has ILL privileges. If all are true, `authorizeByName()` will return true, else false. This value is cached in a private variable to be reported by `authorizedByName()` as needed. If authorization was successful, `authorizeByName()` gets the relevant patron information from Horizon: name, email address and phone number. This data must be stored in such a way that the user interface code can retrieve it. illformscreen.java and isoillscreen.java show how the information will be retrieved. As shown above, the patron information will be retrieved from the resource object named "authos", and the keys that will be used are listed in the source code for these classes. A site may wish to customize these classes if the site wishes to provide more or different patron data than is allowed for. These classes are referred to only by the <oclc-app> elements of the SiteSearch HTML files. An effective way to customize these classes is to create a site-specific class by copying and editing the Java source and to refer to the new classes in the <oclc-app> elements.

All development up to this point had been done using `main()` to test the code. It was time to run the HorizonAccessServer and HorizonAccessClient classes under the JaSSI for the first time. After bouncing the JaSSI and looking at the logs, everything seemed to start up fine. The complication came when trying to authorize a patron. JaSSI could not find the JDBC driver specified and generated a ClassNotFoundException. Analyzing the java.class.path system resource revealed that none of the local additions to the CLASSPATH were in effect. It turns out that CLASSPATH is overridden in the script ssmgr.[hostname], which includes only the classes installed with Java and the classes installed with SiteSearch:

```
    CLASSPATH="$JAVAHOME/lib/classes.zip:
              $ROOTDIR/classes:$ROOTDIR/classes-
/lib/SS$SSVERSION.jar"
```

After editing this script to include our JDBC installation location and our local Java classes, the HorizonAccessClient could load the JDBC drivers and authenticate the user. This is somewhat unsatisfactory as this modification could easily be overwritten during an upgrade. This would mean retaining (or rediscovering) knowledge of an additional file to edit after an upgrade.

In a failed attempt to avoid this problem, a solution was designed to add CLASSPATH components to the UChiAccessServer.ini. These components were added to the java.class.path system property. This had no discernable effect on the class loading mechanism. It appeared that the class loading mechanism retrieved the java.class.path system property early on and cached it, not looking it up again.

This settled, `setDbList()` and `getDbList()` were next (see *Listing 6*). Their type signatures do not help guide the implementation: one accepts an Object, the other returns an Object, with no other hints in the javadoc.

Java's reflection facilities showed that the Object JaSSI passes to `setDbList()` is really a Hashtable of Zdb objects, keyed on String `ZDb.name()`. Looking at both Authorize.java and Topics.java reveals that `getDbList()` should return a Vector of String `ZDb.name()`, naming only those databases to which the user is allowed access. In HorizonAccessClient, `setDbList()` accepts the incoming Hashtable and creates a Vector populated with the `ZDb.name()` of each Hashtable element and stores the Vector. `getDbList()` merely returns the Vector dbList.

Clearly the database list is available to the AccessClient so that the implementor can edit the database list, presumably editing the list based on the type of authentication the user has performed. For example, many of the electronic resources licensed by the University of Chicago Library are for use only by faculty, students and staff. To handle such a requirement, such a resource would not be in the default database list, but would be added during `authorizeByName()`, if the user were eligible. A list of database names, organized by authorization required, could be maintained in the .ini files and read at runtime for ease of maintenance.

4. Conclusions

In a situation like that at the University of Chicago Library, writing a custom access component is well worth the effort. A custom access component obviates the necessity of maintaining access information in an additional production database.

Testing each class of the project outside of the JaSSI is a tremendous help when writing debugging code. The coding/debugging cycle was significantly longer once it had to be done within the context of the JaSSI. Hiding the database behind an abstraction helps keep the access code a bit cleaner and frees the programmer from some of the tedious detail. Keeping the access code separate from the interface code was easier than initially expected. It was also fairly straightforward to keep local customizations separate from the SiteSearch distribution files, though the local files are often in the same directory as distributed files, distinguished by a "UChi" prefix. Whether this makes an upgrade easier remains to be seen. Keeping policy in the configuration files can be difficult in places, if the policy is not easily table-driven.

5. Acknowledgements

I'd like to thank, from OCLC, Lisa Cox for illuminating dark corners of the SiteSearch API, Collette Mak and Dena Bovee for explaining OCLC extensions to ISO 10161, and Michael Burkett for answering similar questions for the previous version of WebZ. Thanks to Rose Smith from the University of Wisconsin for sharing her early code. And from the University of Chicago, thanks to Julie Stauffer, Keith Waclena and Janet Fox for their comments.

6. Code Listings

Listing 1: `HorizonAccessServer` skeleton

```
public class HorizonAccessServer implements AccessServerConnect {

    // Sample stub method...
    public void init(String name, IniFile ini)
    {
       final String method = "HorizonAccessServer.init";
       log.println(Log.TRACE_ENTERS_AND_EXITS, method, "Enter method");
       try {
           /* . . . */
       }
       catch (NullPointerException e) {
           log.println(method, e);
           e.printStackTrace(log.getLogfile());
       }
       log.println(Log.TRACE_ENTERS_AND_EXITS, method, "Exit method");
    }
    /* . . . */

    public String toString()
    {
       String method = className + ".toString";
       Field[] field = this.getClass.getDeclaredFields();
       StringBuffer buf = new StringBuffer(className + ":\n");

       for (int i = 0; i < field.length; i++) {
           Field f = field[i];
           try {
              buf.append(f.getName()).append(" = ");
              buf.append(f.get(this)).append('\n');
           }
           catch (IllegalArgumentException e) {
              log.println(method, e);
              e.printStackTrace(log.getLogfile());
           }
           catch (IllegalAccessException e) {
              log.println(method, e);
              e.printStackTrace(log.getLogfile());
           }
       }
       return buf.toString();
    }

    public static void main(String[] args)
    {
       String iniDir = "/projects/webz-dev/ini/";
       String logDir = "/tmp/";
       String logName = "HorizonAccessServer";
       String jassiIniName = "JaSSIServer.ini";
       String name = "UChiAccessServer";

       try {
           ServerLog log = new ServerLog();
           log.setName("test_user", logName, logDir);
           log.open();
           ServerLog.out = log;
           System.out.println("logging to file " + logName +
                              ", directory " + logDir);
           log.putTraceLevel(Log.TRACE_ALL);

           IniFile.rootDir = iniDir;
           IniFile jassiIni = new IniFile(jassiIniName);

           HorizonAccessServer server = new HorizonAccessServer();
           server.init(name, jassiIni);
           System.out.println(server);
       }
       catch (IOException e) {
           System.err.println(e);
       }
    }
}
```

Listing 2: `HorizonAccessServer.init()`

```
public void init(String name, IniFile ini)
{
    final String method = "HorizonAccessServer.init";
    log.println(Log.TRACE_ENTERS_AND_EXITS, method, "Enter method");

    this.serverName = name;
    try {
        String serverFile = this.serverName + ".ini";
        String accessFile = ini.getValue(name, "inifile");
        if (accessFile == null)
       accessFile = serverFile;
        String sectionName = ini.getValue(name, "sectionName");
        initAccessData(accessFile, sectionName);
        try {
       IniFile serverIni = new IniFile(serverFile);
       initServerData(serverIni);
        }
        catch (IOException e) {
       log.println(method,
                   "Problem processing " + serverFile + ":\n" + e);
        }
    }
    catch (NullPointerException e) {
        this.log.println(className, method +
                    ": " + e +
                    "probably missing data in .ini files");
    }
    log.println(Log.TRACE_ENTERS_AND_EXITS, method, "Exit method");
}

private void initAccessData(String accessFile, String sectionName)
{
    String method = className + ".initAccessData";
    log.println(Log.TRACE_ENTERS_AND_EXITS, method, "Enter method");
    try {
        log.println(Log.TRACE_MSGS, method,
               "access ini file name =  " + accessFile);
        IniFile accessIni = new IniFile(accessFile);
        IniFileSection iniSection = accessIni.getSection(sectionName);

        clientClassName = iniSection.getValue("clientClass");
        checkName = iniSection.getValue("CheckNameAuthos");
        checkIP = iniSection.getValue("CheckIPAuthos");
        checkDNS = iniSection.getValue("CheckDNSAuthos");

        if (clientClassName == null)
       log.println(method,
                   "Could not set clientClassName from [" +
                   sectionName + "] of " +
                   IniFile.rootDir + accessFile);
    }
    catch (IOException e) {
        log.println(method,
               "Problem processing " + className +
               " ini file:\n" + e);
    }
    log.println(Log.TRACE_ENTERS_AND_EXITS, method, "Exit method");
}

private void initServerData(IniFile serverIni)
{
    String method = className + ".initServerData";
    log.println(Log.TRACE_ENTERS_AND_EXITS, method, "Enter method");

    this.serverType = serverIni.getValue("Operations", "ServerType");
    IniFileSection jdbc = serverIni.getSection("JDBC");
    if (jdbc == null) {
        this.log.println(method,
                    "Server .ini file has no [JDBC] section");
    }
    else {
        this.SQLDriver = jdbc.getValue("SQLDriver");
        this.URL = jdbc.getValue("URL");
        this.Login = jdbc.getValue("Login");
        this.Password = jdbc.getValue("Password");
    }
    log.println(Log.TRACE_ENTERS_AND_EXITS, method, "Exit method");
}
```

Listing 3: **`HorizonAccessClient.main()`**

```
public static void main(String[] args)
{
    String iniDir = "/projects/webz-dev/ini/";
    String logDir = "/tmp/";
    String logName = "HorizonAccessClient";
    String jassiIniName = "JaSSIServer.ini";
    String name = "UChiAccessServer";

    String barcode = args[0], pin = args[1];

    try {
        try {
           ServerLog.out = new ServerLog();
           ServerLog.out.setName("test_server", logName, logDir);
           ServerLog.out.open();
           ServerLog.out.putTraceLevel(Log.TRACE_ALL);
           ServerLog.out.println("HorizonAccessClient.main()",
                                 "ServerLog is open and writable.");
           System.out.println("logging to file " + logName +
                              ", directory " + logDir);
        }
        catch (Exception e) {
           System.err.println("Failed to set up the server log");
           e.printStackTrace(System.err);
        }
        // UserLog.out now writes to the same file as ServerLog.out!
        UserLog.out.println("HorizonAccessClient.main()",
                           "UserLog.out is open and writable.");

        UserStateObject.setCapacity(10);
        UserStateObject user = new UserStateObject("1");

        IniFile.rootDir = iniDir;
        IniFile jassiIni = new IniFile(jassiIniName);

        HorizonAccessServer server = new HorizonAccessServer();
        server.init(name, jassiIni);
        System.out.println("Server information:");
        System.out.println(server);

        // To really test, should pull name out of ini files...
        HorizonAccessClient client = new HorizonAccessClient();
        client.initialize(server, user);
        if (!client.authorizeByName(barcode, pin)) {
           System.out.println("Authorization failed for barcode "
                              + barcode + ", pin " + pin + ":");
           System.out.println(client.authorizeErrorMessage());
        }
        else {
           System.out.println(client.authorizeErrorMessage());
           System.out.println(client.toString());
           System.out.println(user);
        }
    }
    catch (IOException e) {
        System.err.println(e);
    }
    String cpath = System.getProperties().getProperty("java.class.path");
    System.out.println("CLASSPATH (aka java.class.path) = " + cpath);
}
```

Listing 4: `HorizonAccessClient.initialize()`

```
HorizonServer horizon;

public void initialize(Object serverObj, Object userObject)
{
    String method = className + ".initialize()";
    log.println(Log.TRACE_ENTERS_AND_EXITS, method, "Enter method");
    try {
        // Should check argument types, log errors;
        server = (HorizonAccessServer) serverObj;
        user = (UserStateObject) userObject;

        try {
           HorizonServer.loadDriver(server.serverSQLDriver());
           horizon = new HorizonServer(server.serverURL(),
                                     server.serverLogin(),
                                     server.serverPassword());
           horizon.open();
        } catch (HorizonException e) {
           log.println(Log.TRACE_ALL, method,
                      "Failed in HorizonServer.loadDriver():\n" + e +);
           e.printStackTrace(log.getLogfile());
        } catch (Exception e) {
           System.err.println("Failed opening HorizonServer: " + e);
           log.println(Log.TRACE_ALL, method,
                      "Failed opening HorizonServer: " + e);
           e.printStackTrace(log.getLogfile());
        }
    } catch (NoClassDefFoundError e) {
        log.println(method, e);
        e.printStackTrace(log.getLogfile());
    }
    log.println(Log.TRACE_ENTERS_AND_EXITS, method, "Exit method");
}
```

Listing 5: **`HorizonAccessClient.authorizeByName()`**

```
String barcode, pin;

public boolean authorizeByName(String autho, String password)
{
    String method = className + ".authorizeByName()";
    log.println(Log.TRACE_ENTERS_AND_EXITS, method, "Enter method");
    this.barcode = autho;
    this.pin = password;
    HorizonAuth auth = new HorizonAuth(horizon);

    try {
        authorizedByName = auth.isFacStuStaff(this.barcode, this.pin);
        if (auth.getAuthCode() != HorizonAuth.AUTH_SUCCESS)
           authCode = INVALID_NAME;
        else
           authCode = AUTHORIZED_BY_NAME;
        authMsg = auth.getAuthMsg();
    }
    catch (HorizonAuthError e) {
        authorizedByName = false;
        authCode = UNAUTHORIZED;
        authMsg = e.toString();
        log.println(method, "Error during authorization");
       e.printFullStackTrace(log.getFile());
    }
    log.println(Log.TRACE_MSGS, method,
               "authCode = " + authCode + ", authMsg = " + authMsg);

    if (authorizedByName)
        getBorrowerInformation(this.barcode);

    log.println(Log.TRACE_ENTERS_AND_EXITS, method, "Exit method");
    return authorizedByName;
}

private void getBorrowerInformation(String barcode)
{
    String method = "HorizonAccessClient.getBorrowerInformation()";
    log.println(Log.TRACE_ENTERS_AND_EXITS, method, "Enter method");

    try {
        BorrowerBarcode bb = new BorrowerBarcode(horizon);
        bb.getByBbarcode(barcode);
        if (!bb.next()) {
           return;            // TODO: log error
        }
        int borrowerNum = bb.getBorrowerNum();

        Borrower b = new Borrower(horizon);
        b.getByBorrowerNum(borrowerNum);
        b.next();
        String borrowerName = Reconstruct.Reconst(b.getName(),
                                               b.getReconst());
        put("clientName", borrowerName, "authos");
        put("clientIdentifier", barcode, "authos");

        BorrowerAddress ba = new BorrowerAddress(horizon);
        ba.getByBorrowerNumAndOrd(borrowerNum, (byte) 0);
        if (ba.next())
           put("email", ba.getEmailAddress(), "authos");

        BorrowerPhone bp = new BorrowerPhone(horizon);
        bp.getByBorrowerNumAndOrd(borrowerNum, (byte) 0);
        if (bp.next())
           put("phone", bp.getPhoneNo(), "authos");
    }
    catch (HorizonException e) {
       e.printStackTrace(log.getLogfile());
    }
    catch (NullPointerException e) {
       e.printStackTrace(log.getLogfile());
    }
    log.println(Log.TRACE_ENTERS_AND_EXITS, method, "Exit method");
}
```

Listing 6: `HorizonAccessClient.{get|set}DbList()`

```
public Object getDbList()
{
    String method = className + ".getDbList()";
    log.println(Log.TRACE_ENTERS_AND_EXITS, method, "Enter method");
    log.println(Log.TRACE_ENTERS_AND_EXITS, method, "Exit method");
    return dbList;
}

public void setDbList(Object dbs)
{
    String method = className + ".setDbList()";
    log.println(Log.TRACE_ENTERS_AND_EXITS, method, "Enter method");
    try {
        if (dbs instanceof Hashtable) {
            Hashtable dbHash = (Hashtable) dbs;
            dbList = new Vector(dbHash.size());
            Enumeration elts = dbHash.elements();
            while (elts.hasMoreElements()) {
                ZDb zdb = (ZDb) elts.nextElement();
                dbList.addElement(zdb.name());
            }
        }
        else {
            log.println(method, "Expected Hashtable argument, got " +
                       dbs.getClass().getName() + ":\n" + dbs.toString());
        }
    }
    catch (Exception e) {
        log.println(method, e);
        e.printStackTrace(log.getLogfile());
    }
    log.println(Log.TRACE_ENTERS_AND_EXITS, method, "Exit method");
    return;
}
```