

4 Database-Backed Authentication

This chapter covers the creation and use of a database-backed `javax.security.auth.spi.LoginModule` and `javax.security.auth.login.Configuration`. The default Configuration implementation provided by the J2SE SDK is flat-file based, requiring VM arguments to use. Our dynamic Configuration is backed by a database, allowing it be changed during runtime much more easily than through flat-files. The `LoginModule` we'll develop will verify a Subject's credentials against those stored in the database, and then add a `Principal` for each user group the Subject is a member of, as specified in the database. As in the previous chapter, each Subject will be given a credential that wraps their username.

To allow us to focus on JAAS itself for this chapter, the "application" that uses JAAS will be a simple class with a `main()` method, as in our quick example. The second section of this book will cover integrating JAAS in more real-world applications. For our database, we'll use the easy-to-use, embedded database Hypersonic, with simple JDBC for data access. These two choices don't provide the full-blown data access layer you'd see in a production application, but they allow us to focus on using JAAS itself, rather than the use of production-grade data access layers.

4.1 The Application

Our "application" class, which will drive the examples in this chapter is below:

```
package chp04;

import java.security.Principal;
import java.util.Collections;
import java.util.Iterator;
import java.util.Set;

import javax.security.auth.Subject;
import javax.security.auth.login.AppConfigurationEntry;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;

public class Main {

    public static void main(String[] args) throws Exception {
        // set Configuration
        DbConfiguration.init(); #1
        // create DbLoginModule entry
        String appName = "simpleDb";
```



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License: <http://creativecommons.org/licenses/by-nc/2.5/>

```
DbConfiguration dbConfig = DbConfiguration.getDbConfiguration();

AppConfigurationEntry appEntry = new AppConfigurationEntry(
    DbLoginModule.class.getName(),
    AppConfigurationEntry.LoginModuleControlFlag.REQUIRED,
    Collections.EMPTY_MAP);
dbConfig.deleteAppConfigurationEntry(appName, appEntry
    .getLoginModuleName());
dbConfig.addAppConfigurationEntry(appName, appEntry); #2
// create LoginContext, login
String username = "mcote";
String password = "secret";
LoginContext ctx = new LoginContext(appName,
    new DbCallbackHandler(username, password));
// authenticate user
boolean authenticated = true;
try {
    ctx.login(); #3
} catch (LoginException e) {
    e.printStackTrace();
    authenticated = false;
}
if (authenticated) {
    // print username
    Subject subject = ctx.getSubject();
    Set creds = subject
        .getPublicCredentials(DbUsernameCredential.class);
    System.out.println("Subject's username: "
        + creds.iterator().next()); #4
    // print principals
    for (Iterator itr = subject.getPrincipals().iterator(); itr
        .hasNext();) {
        Principal p = (Principal) itr.next();
        System.out.println("Principal: " + p.getName()); #5
    }
} else {
    System.out.println("Did not authenticate " + username);
}
}
```

(annotation) <#1 Dynamically sets the **Configuration** to use by calling the **DbConfiguration.init()**.>

(annotation) <#2 Creates an **AppConfigurationEntry** for **DbLoginModule**, adding it to the database, specifying a name and the **login()** method is required to succeed.>

(annotation) <#3 Uses a **LoginContext** instance to authenticate user mcote.>

(annotation) <#4 Prints out the username credential for the authenticated **Subject**.>

(annotation) <#5 Prints out the **Principals** the login process added to the **Subject**.>

To run the example, make sure you're in the base directory of this book's code, and type `ant run-chp04`. The output of running the above code will include:



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>

```
Subject's username: (DbUsernameCredential: name=mcote)
Principal: sysadmin
Principal: users
Principal: austin-lab
```

Though our example application is overly simple, it represents the process you'll most likely follow in your code. First, you tell JAAS which Configuration to use, adding entries for LoginModules as needed. Once the Configuration is set, you can create a LoginContext, passing in a CallbackHandler that can acquire credentials as needed. After the LoginContext has successfully authenticated the Subject, you can access the Subject's newly added Principals and credentials with Subject's get methods.

4.2 Configuration

As explained in the previous chapter, the authentication process in JAAS relies on named groups of LoginModules. In JAAS, a Configuration provides these LoginModule groups during the authentication process. A group of LoginModules is represented by an array of AppConfiguratioEntry instances, each of which specifies a LoginModule implementation, its control flag, and an optional Map of configuration properties.

The abstract class `javax.security.auth.login.Configuration` provides both an interface that all Configuration implementations must implement, and static methods for setting and getting the current Configuration. Our database-backed implementation, then must, at a minimum, implement the abstract methods `getApplicationConfigurationEntry()` and `refresh()`. In addition to these methods, `DbConfiguration` provides methods for creating and deleting `AppConfigurationEntry`s.

The full implementation of `chp04.DbConfiguration` implementation is below¹:

```
package chp04;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Locale;

import javax.security.auth.login.AppConfigurationEntry;
import javax.security.auth.login.Configuration;
import javax.security.auth.login.AppConfigurationEntry.
```

¹ Logging and exception handling code has been removed for brevity.



```
        LoginModuleControlFlag;

import util.db.DbService;

public class DbConfiguration
    extends Configuration {

    static private DbConfiguration dbConfig;

    static public void init() {
        dbConfig = new DbConfiguration();
        Configuration.setConfiguration(dbConfig);
    }

    static public DbConfiguration getDbConfiguration() {
        return dbConfig;
    }

    public void addAppConfigurationEntry(String appName,
        AppConfigurationEntry entry) throws SQLException {
        Connection conn = null;
        try {
            conn = DbService.getInstance().getConnection();
            String sql = "INSERT INTO app_configuration VALUES (?, ?, ?)";
            PreparedStatement pstmt = conn.prepareStatement(sql);
            pstmt.setString(1, appName);
            pstmt.setString(2, entry.getLoginModuleName());
            pstmt.setString(3, controlFlagString(entry.getControlFlag()));
            pstmt.executeUpdate();
        } finally {
            if (conn != null) {
                conn.close();
            }
        }
    }

    public boolean deleteAllAppEntries(String appName)
        throws SQLException {
        Connection conn = null;
        try {
            conn = DbService.getInstance().getConnection();
            String sql = "DELETE FROM app_configuration WHERE appName=?";
            PreparedStatement pstmt = conn.prepareStatement(sql);
            pstmt.setString(1, appName);
        }
    }
}
```



```
        return pstmt.executeUpdate() > 0;
    } finally {
        if (conn != null) {
            conn.close();
        }
    }
}

public boolean deleteAppConfigurationEntry(String appName,
    String loginModuleName) throws SQLException {
    Connection conn = null;
    try {
        conn = DbService.getInstance().getConnection();
        String sql = "DELETE FROM app_configuration "
            + "WHERE appName=? AND loginModuleClass=?";
        PreparedStatement pstmt = conn.prepareStatement(sql);
        pstmt.setString(1, appName);
        pstmt.setString(2, loginModuleName);
        return pstmt.executeUpdate() > 0;
    } finally {
        if (conn != null) {
            conn.close();
        }
    }
}

public AppConfigurationEntry[] getAppConfigurationEntry(
    String applicationName) {
    if (applicationName == null) {
        throw new NullPointerException(
            "applicationName passed in was null.");
    }

    Connection conn = null;
    try {
        conn = DbService.getInstance().getConnection();
        String sql = "SELECT loginModuleClass, controlFlag "
            + "FROM app_configuration WHERE appName=?";
        PreparedStatement pstmt = conn.prepareStatement(sql);
        pstmt.setString(1, applicationName);
        ResultSet rs = pstmt.executeQuery();
        List entries = new ArrayList();
        while (rs.next()) {
```



```
String loginModuleClass = rs.getString("loginModuleClass");
String controlFlagValue = rs.getString("controlFlag");
AppConfigurationEntry.LoginModuleControlFlag controlFlag =
    resolveControlFlag(controlFlagValue);
AppConfigurationEntry entry = new AppConfigurationEntry(
    loginModuleClass, controlFlag, new HashMap());
entries.add(entry);
}

return (AppConfigurationEntry[]) entries
    .toArray(new AppConfigurationEntry[entries.size()]);
} catch (SQLException e) {
    throw new RuntimeException(
        "SQLException retrieving for applicationName="
        + applicationName, e);
} finally {

    if (conn != null) {
        try {
            conn.close();
        } catch (SQLException e) {
            // Time to panic!
        }
    }
}

}

public void refresh() {
}

static String controlFlagString(LoginModuleControlFlag flag) {
    if (LoginModuleControlFlag.REQUIRED.equals(flag)) {
        return "REQUIRED";
    } else if (LoginModuleControlFlag.REQUISITE.equals(flag)) {
        return "REQUISITE";
    } else if (LoginModuleControlFlag.SUFFICIENT.equals(flag)) {
        return "SUFFICIENT";
    } else if (LoginModuleControlFlag.OPTIONAL.equals(flag)) {
        return "OPTIONAL";
    } else {
        // default if unknown
        return "OPTIONAL";
    }
}
```



```
}

static LoginModuleControlFlag resolveControlFlag(String name) {
    if (name == null) {
        throw new NullPointerException(
            "control flag name passed in is null.");
    }

    String uppedName = name.toUpperCase(Locale.US);
    if ("REQUIRED".equals(upperName)) {
        return LoginModuleControlFlag.REQUIRED;
    } else if ("REQUISITE".equals(upperName)) {
        return LoginModuleControlFlag.REQUISITE;
    } else if ("SUFFICIENT".equals(upperName)) {
        return LoginModuleControlFlag.SUFFICIENT;
    } else if ("OPTIONAL".equals(upperName)) {
        return LoginModuleControlFlag.OPTIONAL;
    } else {
        // default if unknown
        return AppConfiguratonEntry.LoginModuleControlFlag.OPTIONAL;
    }
}

}
```

4.2.3 Setting the Configuration to Use

The `init()` method creates a new `DbConfiguration` instance, and uses the static method `Configuration.setConfiguration()`. This sets the global `LoginModuleConfiguration`, making all JAAS code running in the VM use our `DbConfiguration`.

An alternative to setting a single Configuration is to create a composite Configuration. This type of Configuration would contain any number of Configuration instances, and delegate calls to `getAppConfigurationEntry()` to the list of aggregated Configurations.

4.2.4 `getAppConfigurationEntry()`

`DbConfiguration`'s implementation of `getAppConfigurationEntry()` looks up the `LoginModule` group from the database, creating each `AppConfigurationEntry`, and returns the array of `AppConfigurationEntry`s. No options are used or allowed in this example [should we add them, or is it easy to see how it'd be done?], so the empty `Map` provided by `java.util.Collections` is used.

4.2.5 `refresh()`

`DbConfiguration`'s `refresh()` method does nothing. Because the persistence store is a database, changes to the `LoginModule` groups are automatically refreshed. If we were to



implement a caching scheme, where we didn't have to lookup `AppConfigurationEntry`s each time, we might use the `refresh()` method to blow, and reload, the cache as appropriate.

4.2.6 Database Methods

The other methods in `DbConfiguration` are used to maintain the `LoginModule` group database entries.

4.2.7 Control Flags

Unfortunately, `LoginControlFlag` doesn't provide a good way to get a `String` version of instances suitable for storing in a database or other persistence store. The method `controlFlagString()` is used to create a suitable `String` value for `LoginControlFlag` instances. The corresponding method `resolveControlFlag()` is used to go reconstitute the `String` version of the `LoginModule`'s control flag.

4.3 DbLoginModule

The `DbLoginModule` must implement 5 methods: `initialize()`, `login()`, `commit()`, `abort()`, and `logout()`. Both the `login()` and `commit()` methods lookup the information they need from the database, while the `initialize()` and `abort()` methods store or clear out state. Unlike, for example, `Servlets`, `DbLoginModules` are used once and thrown away, so state may be saved on them. Indeed, this is the only way to effectively implement a `LoginModule`.

The code for `DbLoginModule` is below:

```
package chp04;

import java.io.IOException;
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.security.Principal;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Collections;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.UnsupportedCallbackException;
import javax.security.auth.login.LoginException;
```



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>


```
import javax.security.auth.spi.LoginModule;

import util.db.DbService;
import util.id.Id;

public class DbLoginModuleNoLogging implements LoginModule {

    private Subject subject;
    private CallbackHandler callbackHandler;
    private Map sharedState = Collections.EMPTY_MAP;
    private Map options = Collections.EMPTY_MAP;
    private Set principalsAdded;
    private boolean authenticated;
    private String username;
    private Id userId;
    private String password;

    public void initialize(Subject subject,
        CallbackHandler callbackHandler, Map sharedState, Map options) {
        this.subject = subject;
        this.callbackHandler = callbackHandler;
        this.sharedState = sharedState;
        this.options = options;
    }

    public boolean login() throws LoginException {
        NameCallback nameCB = new NameCallback("Username");
        PasswordCallback passwordCB = new PasswordCallback("Password",
            false);
        Callback[] callbacks = new Callback[] { nameCB, passwordCB };
        try {
            callbackHandler.handle(callbacks);
        } catch (IOException e) {
            LoginException ex = new LoginException(
                "IOException logging in.");
            ex.initCause(e);
            throw ex;
        } catch (UnsupportedCallbackException e) {
            String className = e.getCallback().getClass().getName();
            LoginException ex = new LoginException(className
                + " is not a supported Callback.");
            ex.initCause(e);
            throw ex;
        }

        // Authenticate username/password
        username = nameCB.getName();
        password = String.valueOf(passwordCB.getPassword());

        //
        // lookup credentials
        //
    }
}
```



```
Connection conn = null;
try {
    conn = DbService.getInstance().getConnection();
    String sql = "SELECT id, password FROM db_user "
        + "WHERE username = ?";
    PreparedStatement pstmt = conn.prepareStatement(sql);
    pstmt.setString(1, username);
    ResultSet rs = pstmt.executeQuery();
    if (rs.next()) {
        String idStr = rs.getString("id");
        userId = Id.create(idStr);
        String storedPassword = rs.getString("password");
        if (storedPassword == null && password == null) {
            authenticated = true;
        } else if (storedPassword != null
            && storedPassword.equals(password)) {
            authenticated = true;
        } else {
            authenticated = false;
        }
    } else {
        // user does not exist...
        authenticated = false;
    }
} catch (SQLException e) {
    LoginException ex = new LoginException(
        "SQLException logging in user " + username);
    ex.initCause(e);
    throw ex;
} finally {
    if (conn != null) {
        try {
            conn.close();
        } catch (SQLException e) {
            // Log exception.
        }
    }
}
return authenticated;
}

public boolean commit() throws LoginException {
    if (!authenticated) {
        return false;
    }
    // set credential
    DbUsernameCredential cred = new DbUsernameCredential(userId,
        username);
    subject.getPublicCredentials().add(cred);
    // lookup user groups, add to Subject
    Set principals = lookupGroups(username);
    subject.getPrincipals().addAll(principals);
}
```



```
principalsAdded = new HashSet();
principalsAdded.addAll(principals);
return true;
}

static private Set lookupGroups(String username)
    throws LoginException {
    Set principals = new HashSet();

    Connection conn = null;
    try {
        conn = DbService.getInstance().getConnection();
        String sql = "SELECT principal.id, principal.name, "
            + "principal.class "
            + "fbhqwFROM db_user, principal_db_user, principal "
            + "WHERE db_user.username = ? "
            + "AND principal_db_user.user_id=db_user.id "
            + "AND principal.id=principal_db_user.principal_id";
        PreparedStatement pstmt = conn.prepareStatement(sql);
        pstmt.setString(1, username);
        ResultSet rs = pstmt.executeQuery();
        while (rs.next()) {
            String idStr = rs.getString("id");
            Id groupId = Id.create(idStr);
            String groupName = rs.getString("name");
            String className = rs.getString("class");
            if (USR_GRP_CLASS.equals(className)) {
                UserGroupPrincipal grp = new UserGroupPrincipal(groupId,
                    groupName);
                principals.add(grp);
            } else {
                Principal p = resolvePrincipal(groupName, className);
                if (p != null) {
                    principals.add(p);
                }
            }
        }
    } catch (SQLException e) {
        LoginException ex = new LoginException(
            "SQLException logging in user " + username);
        ex.initCause(e);
        throw ex;
    } finally {
        try {
            if (conn != null) {
                conn.close();
            }
        } catch (SQLException e) {
            // Log exception.
        }
    }
}
```



```
    return principals;
}

private static Principal resolvePrincipal(String groupName,
    String className) {
    Exception e = null;
    try {
        Class clazz = Class.forName(className);
        if (Principal.class.isAssignableFrom(clazz)) {
            Constructor c = clazz.getConstructor(STR_ARG);

            return (Principal) c
                .newInstance(new Object[] { groupName });
        }
    } catch (ClassNotFoundException ex) {
        e = ex;
    } catch (SecurityException ex) {
        e = ex;
    } catch (NoSuchMethodException ex) {
        e = ex;
    } catch (IllegalArgumentException ex) {
        e = ex;
    } catch (InstantiationException ex) {
        e = ex;
    } catch (IllegalAccessException ex) {
        e = ex;
    } catch (InvocationTargetException ex) {
        e = ex;
    }
    // Log exception
    return null;
}

public boolean abort() {
    username = null;
    password = null;
    authenticated = false;
    return true;
}

public boolean logout() throws LoginException {
    //
    // Remove usergroup principals
    //

    if (principalsAdded != null && !principalsAdded.isEmpty()) {
        subject.getPrincipals().removeAll(principalsAdded);
    }
    return true;
}

protected boolean isAuthenticated() {
```



```
    return authenticated;
}

protected Subject getSubject() {
    return subject;
}

protected Set getPrincipalsAdded() {
    return principalsAdded;
}

protected String getUsername() {
    return username;
}

static private final String USR_GRP_CLASS = UserGroupPrincipal.class
    .getName();
static private final Class[] STR_ARG = new Class[] { String.class };
}
```

4.3.1 *initialize()*

This method saves the passed in parameters as state on the DbLoginModule instances. Saving these parameters, especially the Subject and CallbackHandler is critical for a LoginModule implementation.

4.3.2 *login()*

The most challenging aspect of LoginModule implementations is understanding how CallbackHandlers and Callbacks are used. Callbacks are objects that represent a type of credential that a LoginModule requests during authentication. A CallbackHandler is an object that knows how to gather certain types of Callbacks. DbLoginModule is only interested in two credentials: username and password, so it creates instances of the JAAS provided Callbacks NameCallback and PasswordCallback. To make sure the correct CallbackHandler is used, when we created the LoginContext in the Main class, we passed in an instance of DbCallbackHandler, which knows how to handle NameCallbacks and PasswordCallbacks. The code for DbCallbackHandler is listed in the next section, XXX.

In our example, we always know that the CallbackHandler we'll be given is an instance of DbCallbackHandler. However, the interface for LoginModule can't guarantee this, so we may get a CallbackHandler that can't gather the credentials DbLoginModule needed. In such cases, the unknown CallbackHandler may throw an `javax.security.login.callback.UnsupportedCallbackException`. Also, if there is an error gathering the credentials, a `java.io.IOException` may be thrown. Throwing an `IOException` seems too narrow in web-centric Java world, but it makes sense from the perspective of a single user application, where gathering credentials may actually involve reading from `System.in` or other input streams.

In either case, how you handle the exceptions is determined by the requirements of your LoginModule. Instead of wrapping and re-throwing the exceptions, as we do in the example, you may want to silently fail. If you'd like your LoginModule to silently fail when it can't



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>

authenticate a user, for whatever reason, then you would catch the exception, perhaps logging it, and simply return `false` from `login()`, telling JAAS to ignore this `LoginModule`.

Once `DbLoginModule` successfully authenticated the `Subject`, `member` field `authenticated` is set to `true`, allowing other methods to check whether authentication was successful or not. The `login()` method returns `true` once it has successfully authenticated a user. Returning `false` would indicate that this `LoginModule` should be ignored, telling JAAS disregard this `LoginModule` when determining if a `Subject` has been authenticated or not.

4.3.3 *commit()*

The `commit()` method is called if authentication for all needed `LoginModules`, as specified each `LoginModule`'s control flag, in a `LoginContext` succeeded. `DbLoginModule`'s implementation adds a `DbUsernameCredential` to the `Subject`'s public credentials, allowing us to retrieve a `Subject`'s username in the future. Next, `commit()` looks up the `UserGroupPrincipals` for the authenticated `Subject`. `UserGroupPrincipals` is a simple `Principal` implementation that wraps around a single name, the name of the user group. A `Subject` may belong to zero or more user groups. To add `Principals` to a `Subject`, you directly modify the `Set` of `Principals` returned by `Subject`'s `getPrincipals()` method. Our `commit()` method delegates to the private method `lookupGroups()` to retrieve user groups from the database.

4.3.4 *abort()*

`DbLoginModule`'s `abort()` implementation cleans up state that was stored on the `LoginModule` instance. More complicated `abort()` methods might remove any state the `LoginModule` saved in the shared session `Map`.

Because the `Subject` should only be modified by the `login()` method, called when overall authentication was successful, `abort()` implementations shouldn't need to revert any modifications done to the `Subject`'s `Principals` or credentials.

4.3.5 *logout()*

When a `Subject` logs out, or is logged out by the system, the `logout()` method is called. There is no guarantee that the `logout()` method will be called on the same instance of the `LoginModule` used to authenticate a user. Because of this `logout()` cannot depend on stored state in the `LoginModule`: `logout()` can certainly *attempt* to use stored state, but it should have a backup plan if the state is not available, and be cautious to avoid null pointers. While stored state may not be available, the `initialize()` method will have been called, assuring that the `Subject`, `CallbackHandler`, shared session `Map`, and `LoginModule` options `Map` are available.

`DbLoginModule`'s `logout()` method first sees if the `Set` of `Principals` added (updated in the `login()` method) is available. If the `Set` is not available, `logout()` looks up the list of `Principals` again, retrieving the username from the `Subject`'s public credentials. Once the `logout()` method obtains the `Set` of `Principals`, it then attempts to remove all of those `Principals` from the `Subject`.



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>

4.4 DbCallbackHandler

The class `chp04.DbCallbackHandler` is an example of a “credential cached” [better phrase] `CallbackHandler`: it doesn’t actually collect credentials, they’re provided through some other means, outside of the responsibility of the `CallbackHandler`, for example, in a JSP page.

The `handle()` method iterates through the `Callbacks` passed in, and fills in the two types of `Callbacks` it knows how to handle, `NameCallback` and `PasswordCallback`. Though `handle()` can throw an `UnsupportedCallback` if it’s given a `Callback` it can’t handle, `DbCallbackHandler`’s implementation instead ignores such `Callbacks`. Depending on your needs, you may implement that handling differently.

Below is the code for `DbCallbackHandler`:

```
package chp04;

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;

public class DbCallbackHandler implements CallbackHandler {

    private String username;
    private String password;

    public DbCallbackHandler(String username, String password) {
        this.username = username;
        this.password = password;
    }

    public void handle(Callback[] callbacks) {
        for (int i = 0; i < callbacks.length; i++) {
            Callback callback = callbacks[i];
            if (callback instanceof NameCallback) {
                NameCallback nameCB = (NameCallback) callback;
                nameCB.setName(username);
            } else if (callback instanceof PasswordCallback) {
                PasswordCallback passwordCB = (PasswordCallback) callback;
                passwordCB.setPassword(password.toCharArray());
            }
        }
    }
}
```



4.5 Principals and Credentials

DbLoginModule adds two types of objects to a Subject: one UserGroupPrincipal for each user group that the Subject is a member of, and one UsernameCredential which wraps the Subject's username.

4.5.1 UserGroupPrincipal

As we'll see in chapter XXX, UserGroupPrincipal is used to associate Permissions with a Subject. Furthermore, a Subject may have any number of UserGroupPrincipals. As with UserGroupPrincipal, many Principal implementations simply wrap the name of the Principal, for example, the user group name. Whenever you create a Principal implementation, you should override the equals() and hashCode() methods as well to ensure that the Principals can be stored in Collection classes. The UserGroupPrincipal, overrides both of these methods, simply keying equality and the hash code value off of the Principal name:

```
package chp04;

import java.security.Principal;

import util.id.Id;

public class UserGroupPrincipal implements Principal {

    private Id id;
    private String name;

    public UserGroupPrincipal(Id id, String name) {
        if (id == null) {
            throw new NullPointerException("Id may not be null.");
        }

        if (name == null || name.length() == 0) {
            throw new NullPointerException(
                "User group name may not be empty.");
        }
        this.id = id;
        this.name = name;
    }

    public Id getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public boolean equals(Object obj) {
```



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5

License: <http://creativecommons.org/licenses/by-nc/2.5/>


```
    if (!(obj instanceof UserGroupPrincipal)) {
        return false;
    }
    UserGroupPrincipal other = (UserGroupPrincipal) obj;
    return getName().equals(other.getName())
        && getId().equals(other.getId());
}

public int hashCode() {
    return getName().hashCode() * 27 + getId().hashCode() * 27;
}
}
```

4.5.2 UsernameCredential

Much like `UserGroupPrincipal`, `UserCredential` wraps a `String` value, the Subject's username. Creating a class for the credential is necessary because the one of the only way to retrieve the credential is by using the method `getPublicCredential(Class)` on `Subject`. This method takes the `Class`, or super-class, of the credential you want to retrieve, and returns a `Set` of credentials of that type. So, if you were to put in the `String` of the Subject's username, it would become difficult to distinguish the `String` username from other credentials that were simply added to the Subject's credential `Set` as a `String`.

Since credential implementations will be stored in collections, they too should override `equals()` and `hashCode()`. The code for `UsernameCredential` is below:

```
package chp04;

import util.id.Id;

public class DbUsernameCredential {

    private String name;
    private Id id;

    public DbUsernameCredential(Id id, String name) {
        if (name == null || id == null) {
            throw new NullPointerException(
                "name and/or id may not be null.");
        } else {
            this.name = name;
            this.id = id;
        }
    }

    public Id getId() {
        return id;
    }

    public String getName() {
        return name;
    }
}
```



```
}

public int hashCode() {
    return getName().hashCode() * 13 + getId().hashCode() * 13;
}

public boolean equals(Object obj) {
    if (obj == this) {
        return true;
    }

    if (!(obj instanceof DbUsernameCredential)) {
        return false;
    } else {
        DbUsernameCredential other = (DbUsernameCredential) obj;
        return getName().equals(other.getName())
            && getId().equals(other.getId());
    }
}

public String toString() {
    StringBuffer buf = new StringBuffer();
    buf.append("(");
    buf.append("DbUsernameCredential: name=");
    buf.append(getName());
    buf.append(")");
    return buf.toString();
}
}
```

4.6 Database Schema

Finally, we must define the database tables that back the above database calls. The following tables are used:

```
CREATE TABLE app_configuration
(
    appName varchar(32) NOT NULL,
    loginModuleClass varchar(255) NOT NULL,
    controlFlag varchar(10),
    PRIMARY KEY ( appName, loginModuleClass )
);
```

```
CREATE TABLE db_user
(
    username(32) NOT NULL,
    password(32),
    PRIMARY KEY ( username )
);
```



This work is licensed under a Creative Commons Attribution-NonCommercial 2.5
License: <http://creativecommons.org/licenses/by-nc/2.5/>

```
CREATE TABLE db_user_group
(
  name(32) NOT NULL,
  username(32) NOT NULL
  PRIMARY KEY ( name )
);
```

Summary

While the previous chapter introduced the domain classes that compose JAAS authentication services, this chapter demonstrated a way to customize those classes to create a database-backed authentication layer. First, we went over one way to store the components of the `Configuration` object in the database, allowing you to more dynamically specify the `LoginModules` required to authenticate users. Next, we covered one way to implement database-backed `Subjects` and their `Principals`. Finally, to perform the actual authentication, we created a custom `LoginModule` that used the database to perform credential verification and, if the `Subject` successfully logged in, add the appropriate database backed `Principals` to the `Subject`.

