Novell exteNd Director

5.0

www.novell.com

USER MANAGEMENT GUIDE

Novell®

Novell Trademarks

ConsoleOne is a registered trademark of Novell, Inc.
eDirectory is a trademark of Novell, Inc.
GroupWise is a registered trademark of Novell, Inc.
exteNd is a trademark of Novell, Inc.
exteNd Composer is a trademark of Novell, Inc.
exteNd Director is a trademark of Novell, Inc.
iChain is a registered trademark of Novell, Inc.
jBroker is a trademark of Novell, Inc.
NetWare is a registered trademark of Novell, Inc.
Novell is a registered trademark of Novell, Inc.
Novell eGuide is a trademark of Novell, Inc.

SilverStream Trademarks

SilverStream is a registered trademark of SilverStream Software, LLC.

Third-Party Trademarks

Acrobat, Adaptive Server, Adobe, AIX, Autonomy, BEA, Cloudscape, DRE, Dreamweaver, EJB, HP-UX, IBM, Informix, iPlanet, JASS, Java, JavaBeans, JavaMail, JavaServer Pages, JDBC, JNDI, JSP, J2EE, Linux, Macromedia, Microsoft, MySQL, Navigator, Netscape, Netscape Certificate Server, Netscape Directory Server, Oracle, PowerPoint, RSA, RSS, SPARC, SQL, SQL Server, Sun, Sybase, Symantec, UNIX, VeriSign, Windows, Windows NT

All third-party trademarks are the property of their respective owners.

Third-Party Software Legal Notices

The Apache Software License, Version 1.1

Copyright (c) 2000 The Apache Software Foundation. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The end-user documentation included with the redistribution, if any, must include the following acknowledgment: "This product includes software developed by the Apache Software Foundation (http://www.apache.org/)."

Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear.

4. The names "Apache" and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact apache@apache.org.

5. Products derived from this software may not be called "Apache", nor may "Apache" appear in their name, without prior written permission of the Apache Software Foundation.

THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Autonomy

Copyright ©1996-2000 Autonomy, Inc.

JDOM.JAR

Phaos

Sun

W3C

3. Notice of any changes or modifications to the files, including the date changes were made. (We recommend you provide URIs to the location from which the code is derived.)

Contents

About This Guide

Purpose

This book describes how to use the three Novell® exteNd Director™ subsystems related to user management:

Subsystem	Used for
Directory	Realm configuration for user authentication
Security	ACL-based user authorization for portal and subsystem elements
User	User profiling and provisioning

Audience

This book is primarily for Java developers.

The chapters about using the Director Administration Console (DAC) are for system administrators:

- Chapter 3, "Using the Directory Section of the DAC"
- Chapter 6, "Using the Security Section of the DAC"
- Chapter 9, "Using the Profiles Section of the DAC"

Directory Management

Provides background information, programming concepts, and code examples for the Directory subsystem

1 About Pluggable Realms

This chapter summarizes the pluggable realm implementations of exteNd Director and provides information about configuring realms. It has these sections:

- About realms
- J2EE application server realms
- LDAP server realms
- PersistManager realm
- Writing a custom realm
- Configuring realms

NOTE: For information about configuring realms, see the section on directory configuration in *Developing exteNd Director Applications*.

About realms

A *realm* is an exteNd Director application's interface to a persistent repository of users, groups, and passwords. In an exteNd Director application, a realm is a class that implements the interface **EbiRealm** or **EbiWritableRealm**.

Types of realms

These are the types of pluggable realms:

Realm type	Description
LDAP realm	Provides an interface to the Novell eDirectory LDAP server. Typically this is a writable realm.
	Support for eDirectory actually consists of several realms: a base LDAP realm plus LDAP realms specific to the exteNd Application Server, BEA WebLogic, and IBM WebSphere.
	NOTE: You cannot use exteNd Director to add or remove containers or custom user attributes in an LDAP realm. Those operations require you to use the LDAP server's own administration interface.
J2EE application server realm	Provides a uniform, platform-independent interface to vendor-specific authentication and user/group management APIs. Can be a readable or writable realm.
	The actual authentication mechanism can be internal or external to the application server. From a programming standpoint, this is entirely transparent to your application.
	For example, if you are using the Novell exteNd Application Server, you can change the authentication provider from Windows NT to NIS+ by making only configuration changes; no code changes are needed.
PersistManager realm	Uses the exteNd Director database as a user/group repository and does its own authentication using a user/password pairing accessed from the database. Typically it is a a writable realm.
Compatibility realm	Provides an interface to an internal authentication API used in previous versions of exteNd Director. Users and groups are stored in an exteNd Director application database. Must be a writable realm. Not recommended for new applications.

Realm access

Realms can be either *readable* (read-only) or *writable* (read-write) as described below. For details about each realm, see the section on directory configuration in *Developing exteNd Director Applications*.

Realm access usage	Description
One readable (read-only) realm	In a readable realm, the Directory subsystem cannot add new users or groups or modify existing ones.
	A readable realm is useful when you want full control over the users and groups that can access an application. For example, intranets often use a central administration application to manage all users and groups.
One writable (read-write) realm	In a writable realm, administrators can use the Directory subsystem to add, delete, and modify users and groups.
	A writable realm is useful when you want to allow anonymous users to add themselves to the realm. For example, Internet sites often allow users to register or create accounts for themselves.
One readable realm and one writable realm	An application with two realms can use each realm for a different purpose.
	For example, an corporate portal application might use a readable realm for employees and a writable realm for customers.

J2EE application server realms

This section describes the pluggable realm implementations for supported J2EE application servers (non-LDAP).

exteNd Application Server realm

This realm uses the exteNd Application Server Directory APIs and can be configured to use LDAP, Windows NT, SilverUsers, and NIS+ security providers. The LDAP and Windows NT security providers are read-only.

NOTE: If you are using eDirectory on the exteNd Application Server, it is recommended that you use the exteNd LDAP realm.

The **EbiSilverServerRealm** interface provides access methods to the exteNd Application Server realm:

EbiSilverServerRealm   SilverServer API

Directory subsystem →

Readable
-NT
-LDAP
-NIS+
-SilverUsers

Writable
-SilverUsers

login()
getUsers()
getGroups()

addUser()
addGroup()

→ External realm

→ SilverUsers
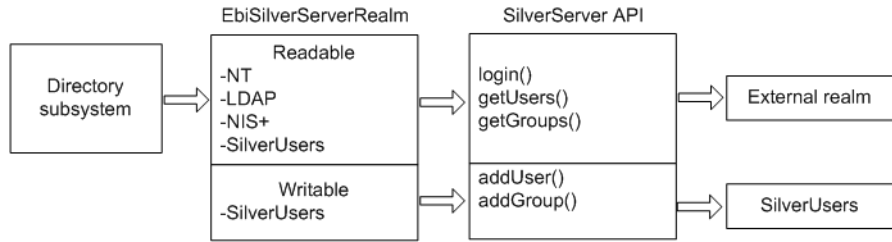
Interface:	EbiSilverServerRealm
Implementation:	EboSilverServerRealm
Authentication provider:	SilverSecurity (default)
	NTSecurity
	LDAPSecurity
	NISPLUSSecurity

exteNd application server compatibility realm

This realm exists for compatibility with ePortal 2.x directory services or any application that requires nested groups. It uses a set of database tables for user and group bindings. It also uses the exteNd Application Server realm's SilverUsers directory as its user repository:

Interface:	EbiUserManagerRealm
Implementation:	EboUserManagerRealm
Authentication provider:	SilverUsers

BEA WebLogic realm

This realm is for BEA WebLogic 6.x. It calls the underlying server API to provide readable access to external realms (LDAP and Windows NT) and readable/writable access to the default realm stored in the file system on the application server.

The exteNd Director **EbiWebLogicRealm** interface provides access methods to the WebLogic server realm API:
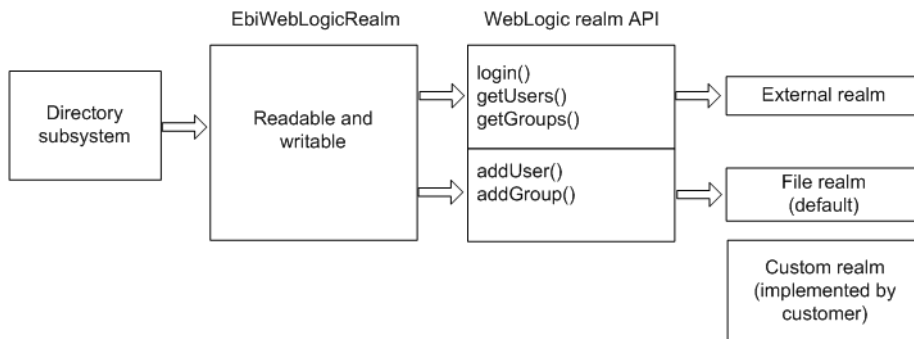
Interface:	EbiWeblogicRealm
Implementation:	EboWeblogicRealm
Authentication provider:	WebLogic internal store

IBM WebSphere realm

This realm is for IBM WebSphere 4.x. exteNd Director accesses data through a custom registry based on a relational database to provide directory services. (IBM WebSphere does not provide a realm-access API other than authentication.) All method calls go through the exteNd Director database.

For authentication, the exteNd Director **EbiWebSphereRealm** interface calls the WebSphere authentication method, which in turn calls through to the database:
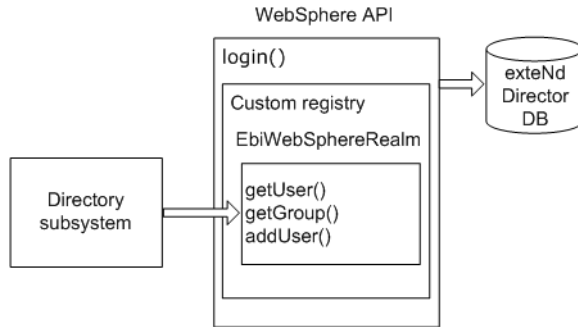
Interface:	EbiWebsphereRealm
Implementation:	EboWebsphereRealm
Authentication provider:	AUTHUSERS (exteNd Director internal store)

LDAP server realms

For writable LDAP realms, exteNd Director provides:

- A **generic base class** that implements the Java Naming Service Interface (JNDI) (JNDI is the standard way in Java to access an LDAP realm hierarchy)
- A **derived class** for each supported J2EE application server to authenticate users using a Novell eDirectory LDAP realm.Base LDAP realm

Base LDAP realm

The JNDI realm base class provides an administrator connection to the LDAP server for retrieving groups and users. This connection is internal, and thus unauthorized external access to data is prevented. A user's JNDI connection is stored as part of the user session when the user is authenticated through the realm:
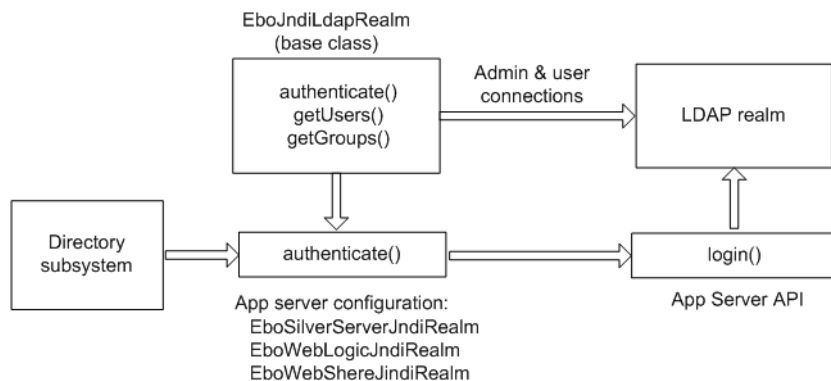
LDAP application server realms

The base class supports generic LDAP authentication only and does not provide authentication through an application server. This latter is provided by a separate class for each application server. The application server realm overrides the authenticate method in the JNDI realm super class and uses the Application Server API for authentication

PersistManager realm

This is a generic realm that can be used to access users and groups directly from the exteNd Director database using the Directory subsystem API. It does not rely on any native application server APIs:
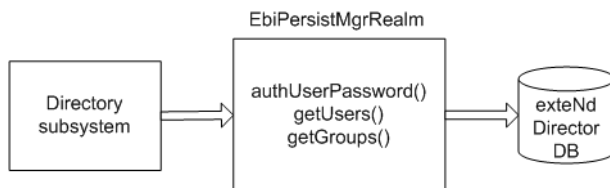
Interface:	EbiPersistMgrRealm
Implementation:	EboPersistMgrRealm
Authentication provider:	AUTHUSERS (exteNd Director internal store)

Writing a custom realm

You can implement your own realm to directly access a directory server—or you can rely on an existing database structure. You can create a custom security realm by implementing these interfaces in the com.sssw.fw.directory.api package:

Directory class	Description
EbiRealm	Interface that custom realms need to implement if they want to provide read-only directory services.
	The directory manager loads instances of realms that implement this interface as well as the subinterface EbiWriteableRealm.
EbiWriteableRealm	Interface that custom realms need to implement if they want to provide write access in the directory service.
	The directory manager loads instances of realms that implement this interface as well as the superinterface EbiRealm.
EbiRealmUser	The wrapper principal used by custom realms. The original principal implementation can be used internally in order to leverage existing principal functionality and APIs.
EbiRealmGroup	The wrapper group principal used by custom realms. The original principal implementation can be used internally in order to leverage existing principal functionality and APIs.

Configuring realms

You can configure the realm used in an exteNd Director application automatically or manually.

Configuring realms automatically

You can configure the realm used in an exteNd Director application in the exteNd Director development environment using either of the following tools, which perform exactly the same function:

- ◆ exteNd Director Project Wizard—for **new** projects

  📖 For a full description of the wizard, see the section on creating a project in *Developing exteNd Director Applications*.

- ◆ exteNd Director Configuration Tool—for **existing** projects

  📖 For a full description of the tool, see the section on reconfiguring an exteNd Director application in *Developing exteNd Director Applications*.

Configuring realms manually

Two descriptor files contain editable key/value pairs representing your application's realm and Directory subsystem configuration properties. The files are located in your project tree in the **DirectoryService-conf** folder.

Descriptor	Contents	For information, see
config.xml	Realm configuration properties	The section on changing configurations in *Developing exteNd Director Applications*
services.xml	Directory subsystem service configuration	

Configuring a different authentication provider

This section applies to exteNd Application Server realms only.

The default realm for the exteNd Application Server is SilverUsers. You can reconfigure your realm to be any of the authentication providers supported by the exteNd Application Server, including Windows NT and NIS+.

➢ **To configure a different authentication provider:**

1 In exteNd Director, open config.xml for the Directory subsystem:
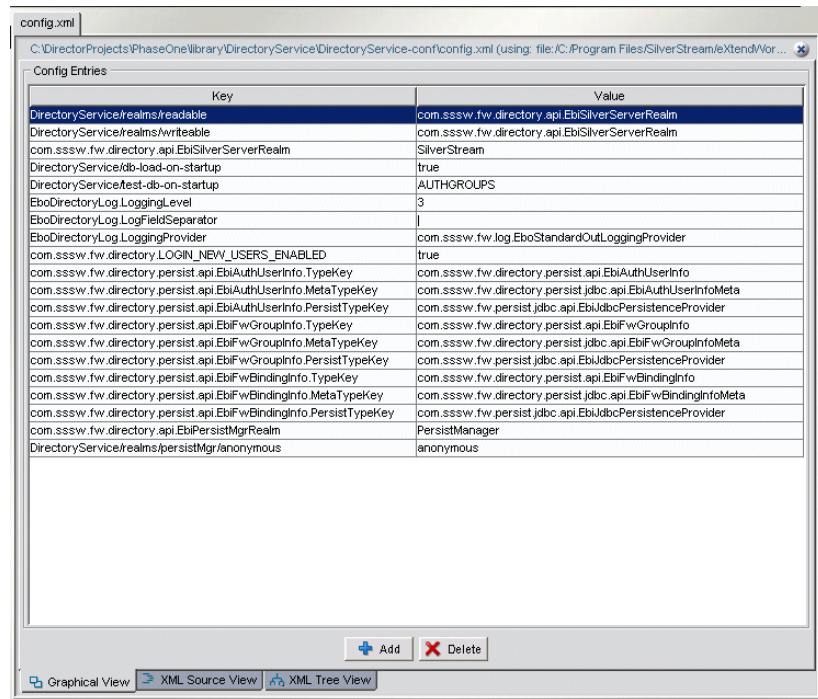
2 Click **Add**.

3 For each key/value pair, double-click the **Key** field and the **Value** field and enter these values:

For Windows NT:

- ◆ Key: **DirectoryService/realms/readable/params/PROVIDER**
- ◆ Value: **NTSecurity**
- ◆ Key: **DirectoryService/realms/readable/params/AUTHORITY**
- ◆ Value: *Your NT realm domain*

For NIS+:

- ◆ Key: **DirectoryService/realms/readable/params/PROVIDER**
- ◆ Value: **NISPLUSSecurity**
- ◆ Key: **DirectoryService/realms/readable/params/AUTHORITY**
- ◆ Value: *Your NIS+ server*

NOTE: If you want to reconfigure your primary realm, see "Configuring the primary realm" on page 25.

4 Redeploy your project.

    📖   For deployment information, see the chapter on deploying an exteNd Director project in *Developing exteNd Director Applications*.

Configuring the primary realm

By default, the readable realm is the primary realm. For API method calls, the Directory subsystem checks the primary realm first.

➢ **To specify your writable realm as the primary realm:**

1 In exteNd Director, open config.xml for the Directory subsystem.

2 Click **Add**.

3 Enter this key/value pair:

- ◆ Key: **DirectoryService/realms/primary**
- ◆ Value: **DirectoryService/realms/writable**

4 Open your project's Directory service descriptor:

```
...\library\DirectoryService\DirectoryService-conf\services.xml
```
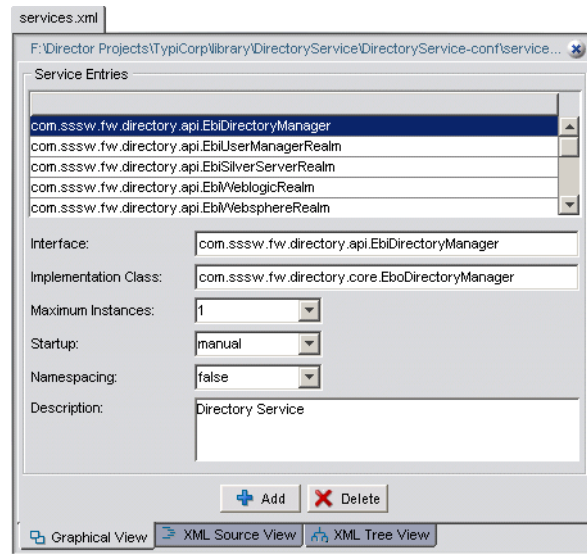
5 Click **Add**.

6 Enter the values as shown:

Form information	Value
Interface	**com.sssw.fw.directory.api.EbiSilverServerRealm**
Implementation Class	**com.sssw.fw.server.silverserver.realm. EboSilverServerRealm**
Maximum Instances	**0** **IMPORTANT:** You must set Maximum Instances to **0** so that the readable realm and writable realm are separate instances of the EboSilverServerRealm implementation.
Startup	**manual**
Description	Any string

7 Redeploy your project.

    📖 For more information, see the section on deploying an exteNd Director project in *Developing exteNd Director Applications*.

Configuring a custom realm

To write a custom pluggable realm, you need to implement the interface com.sssw.fw.directory.EbiRealm (for a readable realm) or EbiWriteableRealm (for a writable realm).

    📖 For more information, see "Writing a custom realm" on page 22.

➢ **To configure a custom realm:**

1 In exteNd Director, open services.xml for the Directory subsystem.

2 Click **Add**.

3 Enter the appropriate values:

Form information	Description
Interface	**A key for the interface or the fully qualified name**. For example: *com.acme.MyCustomRealmInterface*.
Implementation Class	**The fully qualified implementation class**. For example: *com.acme.MyCustomRealmImpl*.

Form information	Description
Maximum Instances	Set this value to **1** if you are planning to use the class as both readable and writable realm or if you are using only one instance of the realm.
	Otherwise, set it to **0** (for multiple instances).
Startup	If you want the class instantiated on server startup, select **automatic**. Otherwise, select **manual**.
Description	Any string.

4   In exteNd Director, open services.xml for the Directory subsystem.

5   If your realm is readable-only, enter a key/value pair that matches the value you entered in services.xml:

   ◆   Key: **DirectoryService/realms/readable**

       Value: Your readable realm interface. For example:
       com.acme.MyCustomRealmInterface

6   If the custom realm is readable/writable, add the same value with this key:

   ◆   Key: **DirectoryService/realms/writable**

       Value: Your readable/writable realm interface. For example:
       com.acme.MyCustomRealmInterface

7   Redeploy your project.

   For more information, see the section on deploying an exteNd Director project in *Developing exteNd Director Applications*.

2 Managing Users and Groups

This chapter describes how to manage realm users, groups, and LDAP containers. It has these sections:

- About the Directory subsystem
- Authenticating users
- Adding users and groups
- Accessing users, groups, and containers

About the Directory subsystem

The Directory subsystem is used to manage readable and writable realms that you have configured using the exteNd Director Project Wizard. To facilitate authorization and other user-based operations within a realm, the Directory subsystem supports the concepts of users and groups. In the case of an LDAP realm, the Directory subsystem also supports the concept of containers and subcontainers.

Directory API

The Directory API provides complete programmatic access to users, groups, and containers. Users, groups, and containers are associated with java.security.Principal. A *principal* is used to authorize access to application resources.

📖 For more information, see Chapter 4, "Using ACL-Based Authorization".

These are the key Directory subsystem classes:

Directory class	Package	Contents
EbiDirectoryDelegate	com.sssw.fw.directory.api	Methods for adding users and groups and accessing user and group principals
EbiRealmContainerDelegate	com.sssw.fw.directory.api	Methods for accessing realm container principals
EbiRealmGroup	com.sssw.fw.directory.api	Methods for adding and accessing group members
EbiDirectoryUsersQuery	com.sssw.fw.directory.api	Methods for getting filtered lists of users
EbiDirectoryGroupsQuery	com.sssw.fw.directory.api	Methods for getting filtered lists of groups
EboDirectoryHelper	com.sssw.fw.directory.client	Directory helper methods for accessing users and security principals
EboFactory	com.sssw.fw.directory.client	Methods for getting delegates and realm provider objects

Authenticating users

Authentication is performed by obtaining a user name and password and checking them against a list of registered users in a directory realm. Knowledge of a registered user ID and the corresponding password is assumed to guarantee that a user is authentic.

Each user registers initially (or is registered by an administrator), using an assigned or self-declared user ID and password. On each subsequent use the user must use the previously declared user ID and password.

When a user of an exteNd Director application successfully authenticates, exteNd Director obtains the list of group and LDAP container memberships for that user and keeps the list available for the duration of the user's session. The list is used for the purpose of authorization when the user attempts to access a protected resource.

About the Login portlet

The Portal subsystem provides a core portlet (Portal Login) for the purpose of user authentication. Custom Web applications generated by the Project Wizard use this portlet by default

Allows registered users to authenticate. Requires users to provide a user ID and password in order to access protected resources.

In your Portal application, you can:

◆ Customize PortalLogin in your project to meet your own requirements

◆ Use the PortalLogin portlet as a template for your own authentication portlet

◆ Write your own authentication portlet using the same basic algorithms

The sources for this portlet are located in your exteNd installation directory at: Director/templates/TemplateResources/portal-core-resource.

Authenticating a user

This code shows how to authenticate a user:

```
// Get a directory delegate.
EbiDirectoryDelegate dirService =
com.sssw.fw.directory.client.EboFactory.getDirectoryDelegate();
// Initialize principal object.
java.security.Principal prin = null;

try {
    // Attempt to authenticate the user ID and password.
```

```
    prin = dirService.authUserPassword(context, uid, pwd);
} catch (Exception ex) {}
```

Adding users and groups

This section describes how to use the Directory API to add users and groups to a configured realm. Adding users and groups automatically adds the associated principal object for assigning security ACLs.

Adding containers   You cannot add LDAP containers from the Directory API. Use your native LDAP realm tools for this purpose. However, you can access existing containers; see "Accessing users, groups, and containers" on page 34.

Adding users and groups using the DAC   You can also use the Director Administration Console (DAC) to add new users and groups. For more information, see Chapter 3, "Using the Directory Section of the DAC".

Adding users using the New User portlet   The Portal subsystem provides a core portlet called New User that allows anonymous users to register themselves. Custom Web applications generated by the Project Wizard use this portlet by default, and the DAC and CMS Administration Console both use customized versions of the New User portlet.

For more information, see "About the New User portlet" on page 73.

Adding a user

To add a user, use the addUser() method on the EbiDirectoryDelegate object. You can also use methods on EboDirectoryHelper to get information about the user, as shown in the example that follows.

Example: checking the self-registration key   This code determines whether or not users are allowed to self-register:

```
m_isLoginNewUsersEnabled = EboDirectoryHelper.isLoginNewUsersEnabled();
```

The property is set in the Directory subsystem config.xml file:

```
<property>
    <key>com.sssw.fw.directory.LOGIN_NEW_USERS_ENABLED</key>
    <value>true</value>
</property>
```

Example: adding the user to the realm   This code shows how to register a new
user:

```
// Get the context. Use this method or one of the others
// available on the factory object.
EbiContext.context = com.sssw.fw.factory.getDirectoryDelegate()
// Get directory delegate.
EbiDirectoryDelegate delegate =
com.sssw.fw.directory.client.EboFactory.getDirectoryDelegate();
// Add the user.
    delegate.addUser(context, user, pwd);
}
```

Adding LDAP users   For a writable LDAP realm you can specify the fully
qualified name (Distinguished Name) or the common name. The API relies on the
LDAP config parameters specified in the Project Wizard.

    For more information, see LDAP realm configuration in *Developing exteNd
Director Applications*.

NOTE: It is best to use the fully qualified name whenever possible. This avoids
potential conflicts with users having identical IDs in separate containers.

Adding a group

To add a group to the writable realm, use the addGroup() method on the directory
delegate:

```
EbiDirectoryDelegate delegate =
com.sssw.fw.directory.client.EboFactory.getDirectoryDelegate();
// Add the user.
    delegate.addGroup(context, groupName);
}
```

Adding a user to a group

To add a user to a group, use the addMember() method on a Group object:

```
try {
    EbiDirectoryDelegate delegate =
    com.sssw.fw.directory.client.EboFactory.getDirectoryDelegate();

    Group group = delegate.getGroup();

    Principal user = delegate.getUser(context, username);
    group.addMember(user);
                            }
     return true;
}
catch (Exception e) { }
        return false;
```

Accessing users, groups, and containers

The directory delegate provides several methods for retrieving users and groups. See the **getUsers()** and **get Groups()** methods in EbiDirectoryDelegate. The realm container delegate has methods for accessing containers in tree realms like LDAP. See EbiRealmContainerDelegate.

Most of the methods described in this section return principals associated with the user, group, or container. This allows you to set security ACLs using the Security API.

📖   For more information, see Chapter 4, "Using ACL-Based Authorization".

User and group queries

The Directory API supports user and group queries for supported realm configurations. This feature allows you to include query strings to get filtered lists of users and groups. Using queries provides performance benefits, especially with large directories. The Directory API includes two classes to support queries:

◆   EbiDirectoryUsersQuery
◆   EbiDirectoryGroupsQuery

Some usage examples follow:

Example: user query   :

```
// Get a directory delegate.
EbiDirectoryDelegate delegate =
com.sssw.fw.directory.client.EboFactory.getDirectoryDelegate();

// Get the query object.
EbiDirectoryUsersQuery query =EboFactory.getDirectoryUsersQuery();

// Specify a query string.
query.whereUserIDStartsWith(search_str);
if (!EboStringMisc.isEmpty(myRealm))query.whereRealmName(myRealm);
if (!EboStringMisc.isEmpty(myGroup))query.whereGroupID(myGroup);

// Get collection of EbiRealmUsers.
Collection users =delegate.getUsers(context, query);
// EbiRealmUser objects are returned.
```

Example: group query

```
// Get a directory delegate.
EbiDirectoryDelegate delegate =
com.sssw.fw.directory.client.EboFactory.getDirectoryDelegate();

// Specify query strings.
EbiDirectoryGroupsQuery query = getDirectoryGroupsQuery();
query.whereGroupIDStartsWith(search_str);
```

```
// To retrieve children of "root" groups only.
query.whereParentGroupID(EbiDirectoryConstants.DIRECTORY_ROOT_GROUP);
if (!EboStringMisc.isEmpty(myRealm)) query.whereRealmName(myRealm);

// Get a collection of EbiRealmGroups.
Collection groups = delegate.getGroups(context, query);
// EbiRealmGroup objects are returned.
```

Dynamic groups support

The Directory subsystem supports accessing existing dynamic groups in eDirectory realms. Dynamic groups are an LDAP realm feature that allows groups to be defined dynamically by a query.

NOTE: exteNd Director does not support the creation or modification of dynamic groups. Use your LDAP client tool for this purpose.

You need to configure your exteNd Director project to access dynamic groups. For more information, see the section on LDAP realm configuration in *Developing exteNd Director Applications*.

To access a dynamic group, use one of the getGroup() methods on EbiDirectoryDelegate. The delegate also has this method for determining dynamic group realm status:

```
public boolean isDynamicGroupSupported(String realm)
```

For more information about dynamic groups in eDirectory, go to:

http://developer.novell.com/research/appnotes/2002/april/05/a0204054.htm

Getting container principals in a tree realm

You can use container delegate methods to access container principals in an LDAP or similar tree-type realm.

NOTE: Setting ACLs on a container principal allows you to apply security inheritance to users in the container hierarchy. For more information, see "Accessing ACLs for containers" on page 55.

This example shows how to get a root container and its descendants:

```
// Intialize Java container object.
Collection availContainers = null;
// Get container delegate.
EbiRealmContainerDelegate conDelegate =
com.sssw.fw.directory.client.EboFactory.getRealmContainerDelegate();
// Get the root (top) container.
EbiRealmContainer root = conDelegate.getRootContainer(context);
// Get Collection of EbiRealmPrincipals.
```

```
EbiRealmContainer's and/or EbiRealmUser's)
if (root != null) {


   availContainers = conDelegate.getDescendants(context, root, true, false);
      if (availContainers != null) {
         availContainers.add(root) ;
      }
      else {
         availContainers = new ArrayList();
         availContainers.add(root) ;
      }


   }

   return availContainers;
}
```

3 Using the Directory Section of the DAC

This chapter describes how to manage the Directory subsystem using the Director Administration Console (DAC). It contains the following sections:

- About the Directory section of the DAC
- Users
- Groups

For information about how to access the DAC, see the section on accessing the DAC in *Developing exteNd Director Applications*.

About the Directory section of the DAC

The **Directory** section of the DAC allows you to view information about the security realm of a deployed exteNd Director application. In the case of a writable realm, you can also change the information.

The Directory section has the following pages:

- Users
- Groups

Search facility

The Directory section provides a search facility for querying users and groups. This is helpful when dealing with large directory structures. A Search dialog appears at appropriate places in the User and Group pages.

➤ **To search for a user or group:**

◆ Enter one or more characters that start the user or group name, then click **Go**. For example:

Users

The **Users** page allows authorized users to add and remove users from the authentication realm.

The left side of the page shows a list of users. The user list from an LDAP realm looks like this:

The **Realm Name** dropdown list is useful only if you have configured separate readable and writable realms.

The **Flush Cache** button updates the user list to match the realm. This is useful if user data can be concurrently modified by another user. This function also applies to servers running in a cluster.

➢ **To change a password:**

   1   Select one of the users listed in the left panel.

   2   In the right panel, click **Modify Password**:

   3   Type the new password twice.

4 Click **Save**.

➢ **To add a new user:**

  1 Click **Add**.

    Add New User

    User ID:*   [   ]

    Password: [   ]

    * Required fields

    💾 Save   ✖ Cancel

  2 Enter the **user ID**.

  3 Enter a **Password**.

  4 Click **Save**.

➢ **To remove a nonadministrative user:**

  1 Select the user.

  2 Click **Remove**.

➢ **To remove an administrative user:**

  1 Make sure that at least one user will remain in each administrative group. Otherwise, administrative security for that group will become open to everyone.

  2 Go to the Groups page in the Directory section of the DAC.

  3 Remove the user from all administrative groups.

  4 If necessary, remove the user from all administrative ACLs:

    4a Go to the Security section of the DAC.

    4b Remove the user from all admin types and permissions.

  5 Go back to the Users page in the Directory section of the DAC.

  6 Select the user.

  7 Click **Remove**.

Groups

The **Directory** section's **Groups** page in the DAC allows authorized users to add and remove groups from the authentication directory and add and remove users from these groups.

The left side of the page shows a list of groups. The user list from an LDAP realm looks like this:

The **Realm Name** dropdown list is useful only if you have configured separate readable and writable realms.

The **Flush Cache** button updates the group list to match the realm. This is useful if group data can be concurrently modified by another user. This function also applies to servers running in a cluster.

➢ **To modify a group:**

1 Select the group.

Group Name	cn=g1,o=administration

Search for User starting with `a` ➜Go

Users selected

LDAPRealm\cn=as,o=Administration
LDAPRealm\cn=linda,o=Administration

Users available (Search Results)

LDAPRealm\cn=a,ou=Portal,o=administration
LDAPRealm\cn=Admin,o=administration

`<` `>`

💾 Save

2 Select the users in the right panel:

♦ To select multiple users: click the first user, then Ctrl-click each additional user.

♦ To select a range of users: click and drag from the first user to the last user.

Use the `<` button to add members to the group and the `>` button to remove members from the group.

3 Click **Save**.

➢ **To add a group:**

1 Click +**Add**.

Add New Group

Group Name*	

* Required fields

💾 Save ✖ Cancel

2 Enter a name for the group.

3 Click **Save**.

➢ **To remove a group:**

1 Select a group.

2 Click **Remove**.

TIP: The users in the group are not removed; only the group itself it removed.
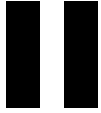
Ⅱ Security Management

Provides background information, programming concepts, and code examples for the Security subsystem

4 Using ACL-Based Authorization

This chapter describes how to use ACLs (access control lists) in exteNd Director. It has these sections:

- About the Security subsystem
- ACLs in exteNd Director
- ACL subsystem administrators
- Accessing ACLs for users and groups
- Accessing ACLs for containers
- Customizing ACL-based authorization

About the Security subsystem

The purpose of the Security subsystem is *authorization*, the process of restricting access to application resources. The Security subsystem is built on top of the standard J2EE security API.

The Security subsystem depends on the Directory subsystem for authentication—in other words, the Security subsystem assumes that requests for protected application resources can only be made by authenticated users.

Authorization is performed by verifying that a user or group (represented by a *principal*) has sufficient permission to perform the operation requested. Principals can be defined using Access Control Lists (ACLs) or mapped to resources using security roles.

   For information about role-based security, see Chapter 5, "Using Security Roles".

ACLs in exteNd Director

ACL-based authorization protects subsystem administrative functions and application resource objects that persist across multiple deployments, such as portlets, documents, folders, group pages, user pages, and profiles.

The following terms define exteNd Director's support for ACL-based authorization:

Term	Definition
ACL	Access control list. A list of entries that restricts access to a specific *element* or *element type*. Each ACL entry associates a *principal* with a set of *permissions*.
	If no ACL is associated with an *element* or with the *element type* to which it belongs, access is unrestricted.
Element	A uniquely identified, persistent resource artifact that is managed by an exteNd Director subsystem. For example, documents and folders in the Content Management subsystem are elements. exteNd Director applications can set and get permissions for securable elements.
	Elements persist across the lifetime of the application server; they are not affected by redeploying the exteNd Director project.
Element type	A string used to define a group of objects with similar functionality or behavior (framework elements such as EbiFolder, EbiDocument, and so on). You can apply ACLs to element types as well as to individual elements.
	The Security subsystem provides a set of built-in element types for different subsystems. Each element type has a list of permissions it supports.
Principal	An authenticated user, group, or container. In exteNd Director, a principal is implemented as a class that extends the Java 2 standard interface java.security.Principal.

Term	Definition
Permission	A type of access to an *element*. exteNd Director includes a set of built-in permissions: CREATE, DELETE, EXECUTE, LIST, PROTECT, PUBLISH, READ, SELECT, UPDATE, and WRITE.
	Built-in permissions are hardcoded and cannot be modified using the Security API.
	NOTE: Negative permissions are not supported in exteNd Director.
	Each ACL subsystem uses a subset of these built-in permissions. Each permission can have a different meaning in each subsystem.
	For information about permissions in the Portal subsystem, see in the section on assigning pages to users and groups in the *Portal Guide*.
	For information about permissions in the Content Management subsystem, see in the chapter on securing content in the *Content Management Guide*.

Accessing principals

In exteNd Director, a principal is implemented as a class that extends the Java 2 standard interface **java.security.Principal**. A *principal* can be used to represent any entity, such as an individual, a corporation, and a login ID.

There are three types of principals defined in exteNd Director:

- **user**—Individual users
- **group**—Groups of users
- **container**—Organizational units within an LDAP-based directory service such as eDirectory. This principal allows you to set ACLs on user containers for security inheritance. See "Accessing ACLs for containers" on page 55.

Principals are represented in the Directory API as interfaces in the **com.sssw.fw.directory.api** package:

```
public interface EbiRealmUser extends Principal
public interface EbiRealmGroup extends Group
public interface EbiRealmContainer extends Principal
```

You can instantiate the principal interface directly in your code, or use one of the following API methods:

Principal type	How to access
user	To access a single principal, use this method in the com.sssw.fw.directory.client package:
	`EboDirectoryHelper.getEbiRealmUser()`
	To get a Collection of principals for a group:
	`EbiRealmGroup.getUserMembers()`
group	Use methods on EbiRealmGroup. For example:
	To get the Java principal for this EbiRealmGroup:
	`EbiRealmGroup.getGroup()`
	To get a Collection of group principals, use one of the methods on EbiRealmGroup:
	`EbiRealmGroup.getChildGroups()`
container	Use methods on EbiRealmContainer. For example:
	To get the parent container for this EbiContainer:
	`EbiContainer.getParentContainer()`
	To get a Collection of descendants for this container:
	`EbiContainer.getDescendants()`

How ACL processing works

Whenever a user attempts to access an element, the Security subsystem checks whether the user has the permission to perform the specified action. If an element has an ACL, exteNd Director checks whether the user has been granted the specified permission.

Situations   These situations can occur:

Situation	Access outcome
The object doesn't have an ACL.	The action proceeds
The object has an ACL but the user is not included by name or by group.	Access is denied; EboSecurityException

Situation	Access outcome
The user is in the ACL for the Locksmith user or for a subsystem admin group with permission for the particular type of access (permissions for admin groups override permissions on individual elements). 📖 For more information, see "ACL subsystem administrators" on page 49.	The action proceeds
The object has an ACL that includes the user (by ID or group), and the user has permission for the particular type of access.	The action proceeds
The object has an ACL that includes the user (by ID or group), and the user has been denied the particular type of access.	Access is denied; EboSecurityException

Process   The Security subsystem determines in three steps whether the user has permission to access an element:

1   Does the element have an ACL?

   ◆   No—Allow access

   ◆   Yes—Go to Step 2

2   Does the user have permission in the element's ACL?

   ◆   Yes—Allow access

   ◆   No—Go to Step 3

3   Is the user a Locksmith user or a subsystem administrator with the appropriate permission for this subsystem element?

   ◆   Yes—Allow access

   ◆   No—Deny access; throw EboSecurityException

ACL subsystem administrators

exteNd Director includes a set of built-in groups that define administrative access to each subsystem using ACLs. You can add and remove users for each permission using the Director Administration Console (see Chapter 6, "Using the Security Section of the DAC").

Here is a general description of access rights for each subsystem administrator group:

Admin element type	Permission	Authorizes users to
ContentAdmin	READ	Get subsystem elements (folders, categories, and documents) in the Content Management subsystem.
	WRITE	Add subsystem elements to the Content Management subsystem.
	PROTECT	Set ACLs for the ContentAdmin type.
GeneralAdmin	PROTECT	A generic Admin type that can be applied to any exteNd Director subsystem. (Reserved for custom subsystem implementation)
LocksmithElement Type	PROTECT	Access all exteNd Director application and ACL subsystem objects, regardless of granted authority. After the Locksmith user is authenticated, the exteNd Director security subsystem adds the user to the admin ACL for each ACL subsystem. The Locksmith can then add individual users to each subsystem ACL. When you first configure your project, the Locksmith user is set to Anonymous by default. This allows any user to access a secure server to redeploy the project, which is convenient in a development environment. **IMPORTANT:** For production deployment, you should change the Locksmith user to a user known to exist in your authentication realm.
PortalAdmin	PROTECT	Access the DAC and portal-related functions. **NOTE:** This permission by itself does not include access to the User, Directory, and Security functions in the DAC.
SearchAdmin	READ	Get existing searchable repositories.
	WRITE	Add, remove, clear, reinitialize, and reset searchable repositories.
	PROTECT	Set ACLs for the SearchAdmin type.

Admin element type	Permission	Authorizes users to
SecurityAdmin	PROTECT	Set ACLs for any Admin type except Locksmith.
UserAdmin	CREATE	Add users, groups, and group profiles.
	READ	View information about profile users.
	DELETE	Remove profile users.
	UPDATE	Update profile user records.
	PROTECT	Set ACLs for the UserAdmin type.

Restricting access to administrators using the API

You can restrict access to portal and content management elements using the EbiSecurityManager.setRestrictedAccess() method. For example, if you restrict access to a document folder for the WRITE permission, only members of the ContentAdmin group have WRITE access to the element.

NOTE: The restricted access right takes precedence over any other ACL associated with the restricted element.

Here are the related methods on the EbiSecurityManager interface:

Method	Returns	Description
setRestrictedAccess()	boolean for success	Restricts specified access for an element to system administrators
check RestrictedAccess()	boolean	Checks whether an element has restricted access

Accessing ACLs for users and groups

This section shows some techniques for using exteNd Director's Directory and Security APIs. The main points of access for ACL security objects are the following delegate interfaces:

Security delegate	Provides access to
EbiSecurityAclDelegate	Security ACLs
EbiSecurityMetaDelegate	ACL-based security metadata
EbiSecurityDelegate	Runtime ACL-based security or role-based security
EbiRealmContainerDelegate	Container principals
EbiDirectoryDelegate	User and group principals

📖   For background information on delegates, see the section on accessing subsystem services in *Developing exteNd Director Applications*.

Accessing ACLs using the DAC   You can also use the Director Administration Console (DAC) to access ACLs.

📖   For more information, see Chapter 6, "Using the Security Section of the DAC".

Getting Security API delegates

This example shows how to get the delegate objects used in the other Security API examples that follow:

```
import com.sssw.fw.security.api.*;

// Getting delegate objects from a factory --
// must be done within a try block.
try {
// Get a metadata delegate.
EbiSecurityMetaDelegate smd =
com.sssw.fw.security.client.EboFactory.getSecurityMetaDelegate();
// Get an ACL delegate.
EbiSecurityAclDelegate ad =
com.sssw.fw.security.client.EboFactory.getSecurityAclDelegate();
// Use the delegate objects.
}
catch (EboFactoryException e) {
      sb.append( e.getMessage() );
}
```

Getting an element type and identifier

This example shows how to get an element type and UUID. It is used in the other examples:

```
// Get delegates. See "Getting Security API delegates" on page 52.
// Get the element type metadata from the EbiSecurityMetaDelegate.
EbiElementTypeMeta typeMeta = smd.getElementTypeMeta(context);
// This example uses the PortalAdmin element.
String portalSubSystem =
EbiSecurityConstants.SUBSYSTEM_PORTAL_SERVICE;
String adminType = typeMeta.getAdminType(portalSubSystem);
String adminID = typeMeta.getAdminID(portalSubSystem);
```

NOTE: Element type names are defined as constants in subinterfaces of **EbiFrameworkElement**. For example, a document in the Content Management subsystem is defined in **com.sssw.cm.api.EbiDocument.EL_DOCUMENT**.

Listing the permissions associated with an element

This example shows how to get a list of the permissions that can be granted to an element:

```
// Get delegates. See "Getting Security API delegates" on page 52.
// Get the element type. See "Getting an element type and identifier" on page 53.
// Get the EbiAccessRightMeta object for the element type.
EbiAccessRightMeta meta = smd.getAccessRightMeta(context,adminType);
// Retrieve the list of permissions.
String[] rights = meta.getPermissionNames();
for (int i = 0; i < rights.length; i++) {
    sb.append( rights[i] );
```

Listing the principals with permission for an element

This example shows how to get a list of principals that have a specific permission for an element. It gets a list of principals assigned to the **PROTECT** permission for the **PortalAdmin** element:

```
import java.security.*;

// Get delegates. See "Getting Security API delegates" on page 52.
// Get the element type. See "Getting an element type and identifier" on page 53.
// Get the principals for a specific permission type.
Principal [] prins = null;
prins = ad.getPrincipalsFromAcl(context, adminID, adminType, EbiPermission.PROTECT);
for (int i = 0; i < prins.length; i++) {
    sb.append( prins[i].toString() );
```

Listing the elements with permissions for a principal

Use this method (available from the EbiSecurityManager) to enumerate all the accessible resources (elements) of a certain type that are accessible to the principal in the session context:

```
/**
    * Returns a Collection of elements accessible to the userwhose context is passed in.
    * @param context context
    * @param elType framework element type, tells the method which
    * element type to determine accessibility for
    * @param rights a list of access right permissions to be
    * checked, e.g. EbiPermission.READ, EbiPermission.WRITE,
    * EbiPermission.EXECUTE, etc.; note that if multiple
    * rights are specified, the method will treat the list
    * as a Boolean OR and will attempt to find elements that
    * have either READ or WRITE or EXECUTE etc. for the user
    * @return a Collection of accessible framework elements of the
    * specified type; the Collection is empty if no accessible
    * elements of the type are found
    */

public Collection getAccessibleElements(EbiContext context, String elType, String[]
rights)
throws EboUnrecoverableSystemException
```

Getting the content of an ACL

This code shows how to get the string representation of an ACL:

```
import java.security.*;

// Get delegates. See "Getting Security API delegates" on page 52.
// Get the element type;
// see "Getting an element type and identifier" on page 53.
// Get the contents of the ACL in the form of a string.
Acl adacl = ad.getAcl(context, adminID, adminType);
String adaclcontent = adacl.toString();
sb.append( adaclcontent );
```

Assigning a principal to an ACL

This code shows how to add a principal to an ACL for an Admin element:

```
import com.sssw.fw.directory.api.*;
import java.security.*;

// Get delegates. See "Getting Security API delegates" on page 52.
// Get the element type. See "Getting an element type and identifier" on page 53.
// Get a Directory delegate.
EbiDirectoryDelegate dd = com.sssw.fw.directory.client.EboFactory.getDirectoryDelegate();
```

```
// Get a principal. Must be a valid realm user.
Principal user = dd.getUser(context,"SomeUser");
// Add the principal to the ACL.
Principal [] prins = new Principal[1];
prins[0] = user;
ad.addPrincipalsToAcl(context, adminID, adminType,
      EbiPermission.PROTECT,
   prins);
sb.append( "Added " + user.toString() + " PROTECT");
```

NOTE: This example requires your code to handle the following exceptions in addition to **EboFactoryException**:

```
catch (EboSecurityException e) {
      sb.append( e.getMessage() );
}
catch (EboException e) {
      sb.append( e.getMessage() );
}
```

Accessing ACLs for containers

The principal type **container** represents an organizational unit within an LDAP tree. The container principal allows you to set ACLs on an LDAP directory container (or similar tree directory structure) and have the ACLs apply to all users in the specified tree hierarchy. This capability is known as *security inheritance*. In exteNd Director, security inheritance is available in applications that are configured for a hierarchical LDAP realm—such as Novell eDirectory.

The Directory API allows:

◆   Traversal of the LDAP hierarchy from the root all the way down to leaf nodes
◆   Direct navigation to a specific node within the hierarchy

The following interfaces are provided in the `com.sssw.fw.directory.api` package:

```
public interface EbiTreeRealm
public interface EbiRealmContainerDelegate
```

NOTE: You cannot add containers to an LDAP realm from exteNd Director. Use your native LDAP tools for this purpose.

Assigning a container principal to an ACL

This code is based on the preceding example (). It shows how to add a container principal to an ACL for an Admin element:

```
import com.sssw.fw.directory.api.*;
import java.security.*;
```

```
// Get delegates. See "Getting Security API delegates" on page 52.
// Get the element type;
// see "Getting an element type and identifier" on page 53.
// Get a new Container delegate.
EbiRealmContainerDelegate rcd = new EbiRealmContainerDelegate();

// Get Container principal. Must be a valid realm container.
Principal container =
   rcd.getEbiRealmContainer(context,"cn=sample,o=users");
// Add the principal to the ACL.
Principal [] prins = new Principal[1];
prins[0] = container;
ad.addPrincipalsToAcl(context, adminID, adminType,
      EbiPermission.PROTECT,
   prins);
```

NOTE: The container principal object is accessible in the API and in the section of the Director Administration Console (DAC) that controls shared and group pages in the Portal. However, you cannot use the DAC to assign administrative access nor Content Management element access to a container principal.

Customizing ACL-based authorization

Customizing the Security service

There are three ways to customize ACL security:

- Write a class that extends the **EboSecurityManager** class to override the runtime ACL validation logic. For example, you could modify the Locksmith ACL metadata to allow additional permissions such as PROTECT, READ, and WRITE.

- Completely reimplement **EbiSecurityManager**. Then change **services.xml** in:

   XWB/DirectorTemplate/Director/library/SecurityService/
      SecurityService-conf

The service definition looks like this:

```
<service>
  <interface>com.sssw.fw.security.api.EbiSecurityManager
  </interface>
  <impl-class>com.sssw.fw.security.core.EboSecurityManager
  </impl-class>
  <description>Security manager that provides authentication
               and permission validation
  </description>
  <max-instances>1</max-instances>
  <startup>M</startup>
</service>
```

Replace **EboSecurityManager** with the name of your own class.

◆ Add a subsystem to provide a different security API, as described next.

Adding ACL-based security to a new subsystem

Adding a new subsystem may be necessary when you are trying to integrate exteNd Director with a third-party security service.

NOTE: This topic goes beyond the scope of this guide. The procedure is merely outlined here. For detailed information, contact Novell Technical Support.

◆ **Add** metadata information for the subsystem into the existing subsystem element type metadata:

```
com.sssw.fw.security.api.EbiElementTypeMeta, singleton
```

◆ **Call** security meta delegate to modify the metadata persistently:

```
com.sssw.fw.security.api.EbiSecurityMetaDelegate
```

◆ **Create** a new access right meta for the administrator type and for any element type that is defined in the subsystem element type metadata.

An *access right meta* object is an API object used to define metadata for associating permissions with a specific element type (or admin type):

```
com.sssw.fw.security.api.EbiAccessRightMeta
```

◆ **Call** security meta delegate to store the metadata object(s) persistently.

◆ Write a custom UI to allow setting ACLs based on the newly created subsystem's admin type and element types by calling the security ACL delegate:

```
com.sssw.fw.security.api.EbiSecurityAclDelegate
```

◆ **Add** runtime ACL validation logic in your new subsystem by calling the security delegate:

```
com.sssw.fw.security.api.EbiSecurityDelegate
```

◆ **Check** administrator access:

```
userHasAccessRight(context, right, adminID, adminType)
```

(Note that Locksmith is checked internally.)

◆ **Check** element level access (if any):

```
userHasAccessRight(context, right, elementUUID, elementType)
```

Custom permissions

exteNd Director allows you to define your own custom permissions using the Security API. See EbiPermissionMeta in Javadoc.

Custom permissions provide a way to use ACL-based authorization on any level of granularity in your application. For example, you can create a set of custom permissions, each of which permits access to a specific method in your application code.

Custom permissions are stored as XML files in the application database. Do not edit the XML files directly—use the Security API.

5 Using Security Roles

This chapter describes how to use exteNd Director security roles in your applications. It has these sections:

- About J2EE role-based authorization
- About exteNd Director security roles
- Creating a security role
- Mapping a security role to a workflow process
- Mapping a security role to a portal page layout
- Accessing security roles programmatically

About J2EE role-based authorization

The exteNd Director Security subsystem supports declarative security in the form of *role-based authorization*. Role-base authorization applies to portlets as defined in the Servlet and Portlet specifications.

The steps for implementing J2EE-compliant roles for portlets are:

1 Define the roles in your portlet deployment descriptor (portlet.xml).

For details, see the portlet.xml schema descriptor, included with your installation at Director_install_dir/Common/SchemaCatalog/portal-app_1_o.xsd

2 Define the same roles in your EAR or WAR application descriptor.

For information about defining roles for a project, see the chapter on the Deployment Descriptor Editor in *Utility Tools*.

3 Map the roles to users in your directory realm. The role-mapping process is distinct for each application server type.

 📖 For information about mapping roles for deployment to the exteNd Application Server, see the chapter on the Deployment Plan Editor in *Utility Tools*.

About exteNd Director security roles

exteNd Director also provides its own role-based authorization in the form of security roles. A *security role* is an XML descriptor that defines user and/or group principals that can be mapped to access rights for certain exteNd Director application objects. exteNd Director security roles can be used outside of the J2EE context, for example, with a custom realm.

The Security subsystem defines declarative role mapping for workflow processes and for portal layouts. For all other application objects, the Security subsystem uses ACL-based authorization, as described in Chapter 4, "Using ACL-Based Authorization".

Creating a security role

➢ **To create a security role using the graphical view:**

1   In the exteNd Director development environment, go to
     File>New>Portal>Security Role.

     The graphical view of a new role descriptor displays.

     ◆   If you prefer to edit the source directly choose the **XML Source View** tab at
          the bottom of the form.
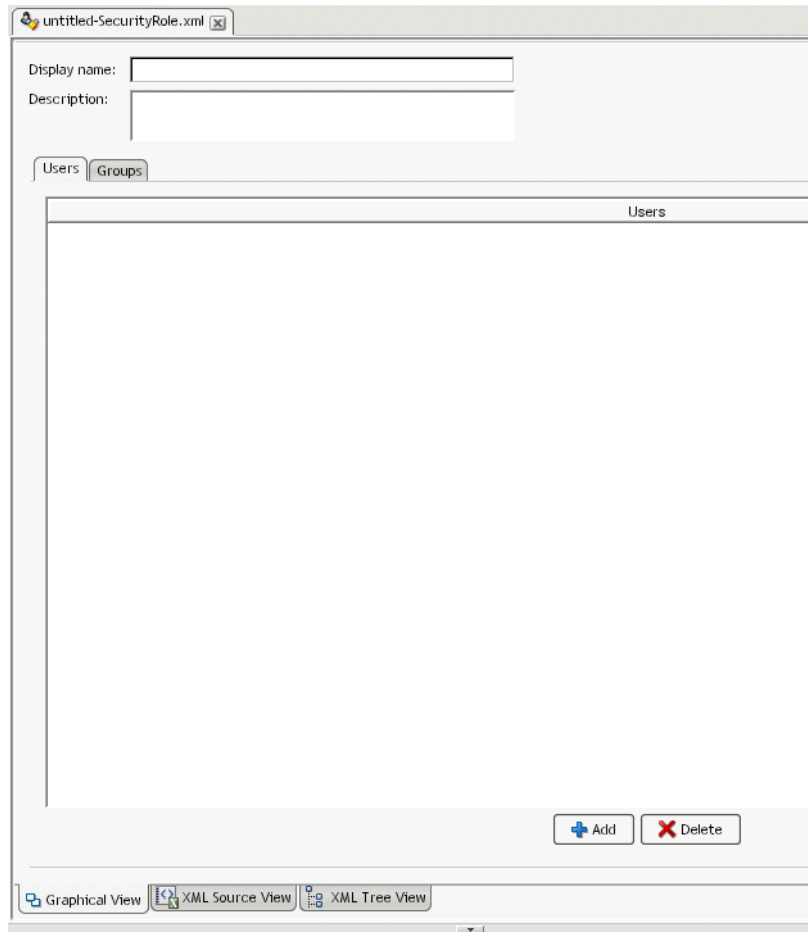
2 Use the editor to enter the XML data:

XML attribute	What you enter
display-name	(Optional) The display name of the access group. This can be any string.
description	(Optional) A description that matches the display name. This can be any string.
User	To add each user: ◆ Choose the **User** tab and click **Add**. ◆ Double-click the new user field and enter name of a valid user defined in your directory realm: If you are accessing an LDAP realm you need to specify the distinguished name, for example: cn=sample,o=acme. NOTE: Roles do not support LDAP **container** elements.
Group	To add each group: ◆ Choose the **Group** tab and click **Add**. ◆ Double-click the new group field and enter name of a valid group defined in your directory realm: If you are accessing an LDAP realm you need to specify the distinguished name, for example: cn=Administrators,o=acme. NOTE: Roles do not support LDAP **container** elements

Here is an example of a security role in the source view:

```
<security-role>
   <display-name>System Administrator</display-name>
   <description>Administers Portal Applications</description>
   <user-map>
      <principal>jdoe</principal>
      <principal>jsmith</principal>
   </user-map>
   <group-map>
      <principal>administrators</principal>
   </group-map>
</security-role>
```

3 Choose **File>Save**.

The role is saved in your exteNd Director project's resource set. For information about the file location click here: security role descriptor location.

Mapping a security role to a workflow process

The workflow process descriptor defines role mappings for user access to workflow processes. The **access-role-map** element determines the user and group principals who are authorized to start new process instances. Here is how the role map element is defined in the workflow process dtd.

```
<!-- Access Role-Map Definition -->
<!ELEMENT access-role-map (role-name*)>
<!ELEMENT role-name (#PCDATA)>
<!-- Description Definition -->
<!ELEMENT description (#PCDATA)>
```

You can use the Workflow Modeler to map roles to the a workflow process. For details, see in the section on process properties in the *Workflow Guide*.

Mapping a security role to a portal page layout

The portal layout descriptor defines role mappings for **list** and **run** access to portal layouts. Here is an example showing list and run access mapped to a role named myRole.xml:

```
<run-role-map>
   <role-name> myRole </role-name>
</run-role-map>
<list-role-map>
   <role-name> myRole </role-name>
</list-role-map>
```

 For information about mapping roles to a portal layout, see the section on creating a layout descriptor in the *Portal Guide*.

Accessing security roles programmatically

After you set up roles and role mappings, you can access role information using the EbiSecurityDelegate interface. This example shows how to check whether a user is included in a specified role.

```
// Variable to hold the name of the security role
// descriptor, without the ".xml" ext
  String mapfile = "mysecurityXML";
  // Get a security manager
  try {
   com.sssw.fw.security.api.EbiSecurityDelegate sd =
     com.sssw.fw.security.client.EboFactory.getSecurityDelegate();
     // Check if user is in this role
     if (sd.isUserInRole(context, mapfile));
```

```
   // get a document, else display "no access" message
}
catch (com.sssw.fw.exception.EboFactoryException e)
{
   // display message
}
```

6 Using the Security Section of the DAC

This chapter describes how to use the Director Administration Console (DAC) to control user access to ACL subsystem administrative functions.

This chapter includes one section:

◆ Modifying administrative access

📖 For information about how to access the DAC, see the section on accessing the DAC in *Developing exteNd Director Applications*.

Modifying administrative access

The Security section allows you to view the members of each pairing of a subsystem administrator type and a permission, as shown below:

➢ **To assign users and groups:**

   1   Select an **Admin type** from the dropdown list:

   📖   For a description of the Admin types, see "ACL subsystem administrators" on page 49.

2   Select a **Permission** from the dropdown list.

The list of permissions depends on the Admin type you selected.

```
PROTECT ▼
DELETE
UPDATE
PROTECT
CREATE
READ
```
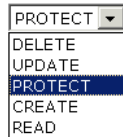
    📖   For a description of the permissions for each type, see "ACL subsystem administrators" on page 49.

After you select a permission, lists of users and groups for the selected ACL appear in the **Users selected** and **Groups selected** lists.

3   Select **users** or **groups** from the users available and groups available lists:

    ◆   To select multiple users or groups: click the first, then Ctrl-click each additional.

    ◆   To select a range of users or groups: click and drag from the first to the last.

Use the ⟨ button to add items, and the ⟩ button to remove items.

4   Click **Save**.

NOTE: If you need to remove an administrative user from the realm:

5   Make sure there will be at least one user remaining in each administrative group. Otherwise, anyone will be able to administer the subsystem.

6   Remove the user from all administrative groups and, if necessary, from all administrative ACLs.

▮▮▮ User Profiling

Provides background information, programming concepts, and code examples for the User subsystem

7 Managing User Profiles

This chapter describes how to manage user profiles. It contains the following sections:

- About user profiles
- Profiles and realm configurations
- About the New User portlet
- Accessing profiles using the API
- Rules and user profiling

About user profiles

The purpose of the User subsystem is the profiling of Web application users. User *profiles* are persistent collections of data associated with individual users of a Web application. Individual data items within a user profile are called *attributes*. Your exteNd Director application can obtain attributes and store the information in a profile. Later it can retrieve, act upon, and analyze the information.

For more information, see Chapter 8, "Accessing User Attributes".

How profiles are used

Typically, profiles are used for two general purposes:

- Allowing users to personalize Web applications
- Tracking user behavior

A portal application might rely on the information in the user profile to determine what content can or should be delivered to a given user or what operations can be performed.

In a retail business, for example, a profile for a customer could contain:

- Account information such as shipping and billing address, e-mail address, age, gender, occupation, credit card information, and so on
- User-specified areas of interest, such as product categories
- Feature and page layout preferences other than those provided by the Portal subsystem
- Login history, pages viewed, links and buttons clicked, and transactions made (items bought and processes started)

Profiles and realm configurations

Profiles are stored differently depending on whether you are using a non-LDAP or an LDAP realm. Here are the main differences:

Non-LDAP realms	LDAP realms
Profile information is stored in the application database, which is also used to store ACL-based security information.	Profile information is stored within the LDAP directory; the application database is used for ACL-based security information.
The profile database exists independently from the authentication realm.	Profile information and authentication information are stored in the same user record.
exteNd Director applications use the JDBC persistence provider to access the database.	exteNd Director uses the JNDI persistence provider to access the LDAP directory.

Checking the realm configuration

exteNd Director allows you to reconfigure an application to use a different authentication realm. If you need this kind of flexibility, you can write code that is general enough to work with both LDAP and non-LDAP realms.

To find out whether or not your application is configured for an LDAP realm, use:

```
EboUserHelper.getUserDataStore()
```

This method returns a string value indicating whether the User subsystem uses **JNDI** (LDAP directory) or **JDBC** (application database).

Checking for a writable realm

To find out whether or not your application is configured for a writable realm, use:
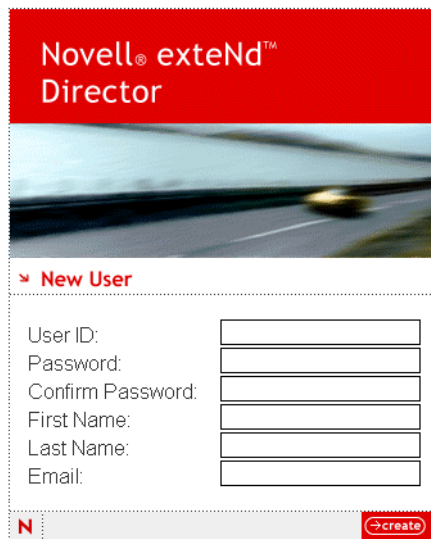
```
EboUserHelper.isReadOnlyUserSchema()
```

This method returns a boolean value indicating whether or not the User service schema is modifiable. A JDBC schema can be readable or writable, but a JNDI schema is not modifiable within exteNd Director.

About the New User portlet

The New User portlet is one of the core portlets used in the custom Web application. It allows anonymous users to add themselves to the writable realm and automatically creates a profile for each new user.

The New User portlet is provided as a template for your application. You can copy and customize it or design a new one:

The New User portlet does the following:

- Adds a user name and password to the writable realm (if enabled)
- Creates a user profile identified by the combination of the user's realm name and an automatically generated 32-bit UUID
- Adds some default attributes to the profile

The sources for this portlet are located in your exteNd installation directory at: Director/templates/TemplateResources/portal-core-resource.

Accessing profiles using the API

The User API provides methods to create profiles, find profiles, and store and retrieve user-specific information. These are the key classes for user profiling:

User subsystem class	Provides access to
EbiUserDelegate	Methods for creating and accessing user profiles
EbiUserInfo	User attributes
EbiUserQuery	Methods for querying users based on attributes and other criteria
EboUserHelper	Convenience methods for accessing user information
EboFactory	Methods for instantiating the user delegate and related objects

Accessing profiles using the DAC   You can also use the Director Administration Console (DAC) to access user attributes. For more information, see Chapter 9, "Using the Profiles Section of the DAC".

Creating a new profile

This code shows how the New User portlet adds a user profile:

```
//
// Get a profile delegate from the user service.
//
EbiUserDelegate userDelegate =
    com.sssw.fw.usermgr.client.EboFactory.getUserDelegate();
//
// Instantiate an empty profile object for this user.
//
EbiUserInfo userInfo = (EbiUserInfo)userDelegate.createUserInfo();
//
// Add profile info (default attributes) for the user.
```

```
//
userInfo.setUserID(m_uid);
userInfo.setUserFirstName(m_firstName);
userInfo.setUserLastName(m_lastName);
userInfo.setUserEmailAddress(m_email);
userInfo.setUserAuthenticatedRealmName
    (dirService.getPrimaryRealmName());
//
// Add the new profile.
//
boolean status = userDelegate.createUser(context, userInfo);
```

Looking up user profiles

The User API includes user query and user metadata query classes that you can implement to retrieve a list of user profiles that meet certain criteria.

For more information, see EbiUserQuery and EbiJndiQuery in the *API Reference*.

Getting a user profile

This code shows how to obtain and display a user profile:

```
import com.sssw.fw.usermgr.api.*;
import com.sssw.fw.usermgr.client.*;

try {
    //
    // Get the user identifier.
    //
    String userUUID = EboUserHelper.getUserUUID(context);
    //
    // Get a user delegate object from the factory.
    //
    EbiUserDelegate userDelegate = EboFactory.getUserDelegate();
    //
    // Get a user info object.
    //
    EbiUserInfo userInfo =
        (EbiUserInfo)userDelegate.
            getUserInfoByUserUUID(context,userUUID);
    //
    // Get the registration info and add to output buffer.
    //
    sb.append("UserID: " + userInfo.getUserID() + "<br>");
    sb.append("UserUUID: " + userInfo.getUserUUID() + "<br>");
    sb.append("UserFirstName: " +
        userInfo.getUserFirstName() + "<br>");
    sb.append("UserLastName: " +
        userInfo.getUserLastName() + "<br>");
```

```
            sb.append("UserEmailAddress: " +
                userInfo.getUserEmailAddress() + "<br>");
    }
    catch (EboFactoryException e) { sb.append( e.getMessage() ); }
    catch (EboSecurityException e) { sb.append( e.getMessage() ); }
```

Rules and user profiling

If you are developing an application that implements profiling extensively, you should consider using exteNd Director's Rule subsystem. The following scenario suggests how rules can be applied to user profiling.

Suppose you have a retail Web site where you want to track the total amounts of customers' Web purchases and specify a threshold amount that triggers a special discount. Here is how you could use rules to develop this application:

1 Add a user attribute to track the amount—AllPurchasesAmt, for example.

2 In the Rule Editor, create a rule using the built-in CheckWhiteboard condition that allows you to check user profiles (through the ^attributename syntax). Enter a threshold amount and give it a key value—such as ^threshold. For the action section you can return a boolean or appropriate content.

3 In your code:

   ◆ Get the purchase amount for a completed transaction.

   ◆ Get the attribute value, add the purchase amount, and update the total.

   ◆ Set the value on the session whiteboard using the EbiContext.setValue() method. Give it the keyname you defined in the rule.

   ◆ Fire the rule and handle the result.

About conditions and actions

Conditions and actions are available in the exteNd Director rules engine to interact with the User and Content Management subsystems (through the Content Query Action). This means you can easily implement personalization rules like this:

```
If "UserAge" > 35
AND "PortfolioTotal" > 30,000
Then Select Investing Documents Level 3
AND set "FinanceLevel" to "Gold"
```

Through exteNd Director's easy-to-use API, custom tag library, and rules engine conditions and actions, you can quickly deliver personalization services to users.

📖 For more information about rules, see the chapter on how to use rules in the *Rules Guide*.

8 Accessing User Attributes

This chapter discusses profiling with user attributes and has these sections:

- About attributes
- Attribute properties
- Accessing attributes using the API

For background information, see Chapter 7, "Managing User Profiles".

About attributes

Attributes are individual data items within a user profile. Each attribute corresponds to a column or field (in database terminology).

Attributes can be any data that you want to associate with a user. The User subsystem has a set of built-in attributes and you can create and use your own user attributes for user profiling.

Built-in attributes

The following built-in string attributes are present in each user profile regardless of realm configuration:

- User ID
- User UUID
- First name
- Last name
- E-mail address

The values of the User ID and User UUID attributes are used by many of the exteNd Director API packages to identify users.

New User portlet   The New User portlet is a core portlet used in the custom Web application in exteNd Director projects. It allows anonymous users to add themselves to the writable realm and automatically creates a profile for each new user. This profile includes the built-in attributes.

     For more information, see "About the New User portlet" on page 73.

Attributes and non-LDAP realms

Applications that use a writable non-LDAP realm use the application database to store profile information. By default, the profiles in the database contain only built-in attributes. These applications can create custom attributes as needed.

     For an example, see "Creating an attribute (non-LDAP)" on page 82.

Attributes and LDAP realms

Applications that use an LDAP realm use the LDAP directory to store profile information. LDAP provides a rich set of attributes that are intended to be sufficient to meet the requirements of most Web applications. Use the LDAP administration console to add new (custom) attributes to the directory.

NOTE: You cannot add custom attributes to an LDAP directory from an exteNd Director application.

The User API provides a way for an application to retrieve a list of all available LDAP attributes. The **User LDAP Options** panel in the EAR Wizard allows you to make specific attributes available to the User API and to exclude others. It also allows you to exclude certain syntax definitions.

     For more information, see the section on LDAP user options in *Developing exteNd Director Applications*.

Attribute properties

Each attribute has a name, a description, a display property, and a data type.

Display properties

Each attribute has a display property that can have either of two values:

Attribute usage	Description
Displayable	Information you collect directly from a user, such as personal data or preferences. The user typically specifies this information on a registration form.
Hidden	Information you store that is not explicitly provided by the user—for example, buying patterns or click stream counts.

Data types

There are two types of attribute data values:

- String values up to 255 characters long
- BLOB (Binary Large OBject) as defined in JDBC 2.0

BLOB attributes are used to store binary data (such as large documents and images) in the form of a byte array. Separate API methods are provided for using BLOB attributes.

NOTE: The User API supports multivalued attributes for applications that use an LDAP realm. The Director Administration Console (DAC) also allows the display and modification of existing values for these attributes.

Accessing attributes using the API

You can define whatever attributes you need and use them in your code to personalize content. This section describes how to access attributes and how to create and set custom attributes.

📖   For an overview of the User API, see "Accessing profiles using the API" on page 74.

Accessing attributes using the DAC   You can also use the Director Administration Console (DAC) to access user attributes.

📖   For more information, see Chapter 9, "Using the Profiles Section of the DAC".

Getting a list of attributes (non-LDAP)

This code shows how to obtain a user profile and display all the custom attributes. Note that the **EbiUserInfo** object has a specific method for each built-in attribute:

```
import com.sssw.fw.usermgr.api.*;
import com.sssw.fw.usermgr.client.*;

try {
    //
    // Get the user identifier.
    //
    String userUUID = EboUserHelper.getUserUUID(context);
    //
    // Get a user delegate object from the factory.
    //
    EbiUserDelegate userDelegate = EboFactory.getUserDelegate();
    //
    // Get a user info object.
    //
    EbiUserInfo userInfo =
        (EbiUserInfo)userDelegate.
            getUserInfoByUserUUID(context,userUUID);
    //
    // Get the registration info and add to output buffer.
    //
    sb.append("UserID: " + userInfo.getUserID() + "<br>");
    sb.append("UserUUID: " + userInfo.getUserUUID() + "<br>");
    sb.append("UserFirstName: " +
        userInfo.getUserFirstName() + "<br>");
    sb.append("UserLastName: " +
        userInfo.getUserLastName() + "<br>");
    sb.append("UserEmailAddress: " +
        userInfo.getUserEmailAddress() + "<br>");
    //
    // Get a user metadelegate object from the factory.
    //
    EbiUserMetaDelegate userMetaDelegate =
        EboFactory.getUserMetaDelegate();
    //
    // Get the current user's metadata.
    //
    EbiUserMeta userMeta = userMetaDelegate.getUserMeta(context);
    if (userMeta == null) {
        userMeta = userMetaDelegate.createUserMeta();
        userMetaDelegate.addUserMeta(context, userMeta);
    }
    else {
        //
        // Get all custom attribute names and values.
        // Add to output buffer.
        //
        String[] attributes = userMeta.getUserAttributes();
        if (attributes == null) {
```

```
                    sb.append("No attributes - string array is null");
            }
            else {
                for (int i=0;i<attributes.length;i++) {
                    String attributeValue =
                     userInfo.getAttributeValue(context,attributes[i]);
                    sb.append(attributes[i] + ": " +
                        attributeValue + "<br>");
                }
            }
        }
    }
    catch (EboFactoryException e) { sb.append( e.getMessage() ); }
    catch (EboSecurityException e) { sb.append( e.getMessage() ); }
```

NOTE: When you run this on a profile that has no custom attributes, the EbiUserMeta object is null.

Getting a list of attributes (LDAP)

To retrieve specific attribute values directly from an LDAP directory, use:

```
EboUserHelper.getDirectoryUserAttributes(context, userdn, String[]
names)
```

This method inputs the context, the user DN string, and an array of strings naming the attributes for which to return values. The return value is a Map (object array) containing values for each requested user attribute.

NOTE: This method returns only the first value of a multivalued attribute.

Identifying multivalued attributes

When using an LDAP directory, attributes can have multiple values. To check for this type of attribute, use:

```
EbiUserMeta.isUserAttributeSingleValued(attrname)
```

This method returns a boolean value indicating whether or not the specified attribute is limited to a single value.

Creating an attribute (non-LDAP)

This code adds a custom attribute to a user profile:

```
import com.sssw.fw.usermgr.api.*;

try {
    //
    // Get a user metadelegate object from the factory.
    //
    EbiUserMetaDelegate userMetaDelegate =
        com.sssw.fw.usermgr.client.
            EboFactory.getUserMetaDelegate();
    //
    // Get a writable copy of the user metadata.
    //
    EbiUserMeta userMeta =
        userMetaDelegate.getClonedUserMeta(context);
    //
    // Add a custom attribute.
    //
    String attrib_name = "Employer";
    userMeta.addUserAttribute(attrib_name,"Name of employer",true);
    //
    // Save the modified metadata including the new attribute.
    //
    userMetaDelegate.modifyUserMeta(context,userMeta);
    //
    // Return the attribute name and value.
    //
    sb.append("Added attribute: " + attrib_name);
}
catch (EboFactoryException e) { sb.append( e.getMessage() ); }
catch (EboSecurityException e) { sb.append( e.getMessage() ); }
```

Setting an attribute value

This code sets the value of a custom attribute:

```
import com.sssw.fw.usermgr.api.*;

try {
    //
    // Get a user delegate object from the factory.
    //
    EbiUserDelegate userDelegate =
        com.sssw.fw.usermgr.client.EboFactory.getUserDelegate();
    //
    // Get a user info object from the factory.
    //
    EbiUserInfo userInfo =
        com.sssw.fw.usermgr.client.EboUserHelper.
            getUserInfo(context);
```

```
        //
        // Set the value of the attribute to Novell.
        //
        String attrib_name = "Employer";
        userInfo.setAttributeValue(context, attrib_name, "Novell");
        //
        // Write the new value into the user info object.
        //
        userDelegate.modifyUser(context,userInfo);
        //
        // Get the new attribute value and append to output buffer.
        //
        sb.append(attrib_name + ": " +
            userInfo.getAttributeValue(context, attrib_name));
}
catch (EboFactoryException e) { sb.append( e.getMessage() ); }
catch (EboSecurityException e) { sb.append( e.getMessage() ); }
```

9 Using the Profiles Section of the DAC

This chapter describes how to use the Director Administration Console (DAC) to manage users and user attributes in the user profile directory. It has these sections:

◆ About the Profiles section of the DAC
◆ User profiles
◆ Attributes

    For information about how to access the DAC, see the section on accessing the DAC in *Developing exteNd Director Applications*.

About the Profiles section of the DAC

The **Profiles** section of the DAC allows you to view user profiles in a deployed exteNd Director application. In the case of a writable non-LDAP realm, you can also change the information.

The Profiles section has two pages:

◆ User profiles
◆ Attributes

User profiles

The User Profiles page allows authorized users to add and remove user profiles by selecting from a list. In a non-LDAP realm, profiles are stored in the application database and thus do not necessarily have a one-to-one correspondence with users. In an LDAP realm, however, each user record is the user profile.

The left side of the page shows a list of profiles:

User Profiles	Attributes

Realm Name:
exteNd Server ▼

Search for User Profile starting with:

a	➡ Go

☑ Show All Users

----
admin

➕ Add ⬅ Remove

➤ **To modify a profile:**

1   Select the writable realm from the Realm Name list.

2   Select a user from the list to view profile data.

3 Click **General**:

4 Change the **User ID**, **First Name**, **Last Name**, and **Email** data as needed; these fields are the same in both non-LDAP and LDAP realms.

> **TIP:** The other attributes vary according to realm type and are discussed in "Attributes" next.

5 Click **Defaults** to view user profile defaults:

6 Change the defaults as needed.

7 Click **Save**.

➢ **To add a profile:**

1 Click +**Add**:

Create User Profiles

Select User: (Directory Users)	admin ▾
User ID:*	admin
First Name:	
Last Name:	
Email:	

* Required fields

💾 Save ✕ Cancel

2 Select a user from the **Select User** dropdown list, or enter the user name in the **User ID** field.

3 Fill in the information.

4 Click **Save**.

Attributes

The DAC allows authorized users to view and modify attribute data.

NOTE: In an LDAP realm, users cannot modify their own attributes because of security restrictions in eDirectory. By default, users have rights only to read their own attributes; they do not have modify rights. The eDirectory administrator can grant rights to modify attributes to any user.

The **User Profiles** page allows you to manage the attribute values associated with a user profile:

The **Attributes** page also allows you to view the attributes that exist within a realm. The non-built-in attributes vary according to realm type, as follows.

- In a non-LDAP realm, authorized users can add, remove, and edit attributes:

- In an LDAP realm, you cannot manage attributes using the DAC. You must use the LDAP directory server's administration console, such as ConsoleOne in Novell eDirectory, to add or remove attributes.

➢ **To modify an attribute:**

1 Click **edit**:

> User Profiles | Attributes
>
> Edit attribute (Employer)
>
> **Name***   Employer
>
> **Show**   ☑
>
> **Description**   Name of employer
>
> * Required fields
>
> 💾 Save ✖ Cancel

2 Specify the **Name** (required) you will use to identify the attribute in your application code.

   TIP: Changing the name of an attribute invalidates any code you have written that uses the previous name.

3 Check the **Show** box to make the attribute visible to users; uncheck the **Show** box for tracking, usage statistics, and so on.

4 Specify the **Description**. This field is searchable using methods on the `EbiUserInfo` class (see `EbiUserQuery` in the *API Reference*).

5 Click **Save**.

➢ **To add an attribute:**

1 Click **add**:

> Add User Attribute
>
> **Name***   [   ]
>
> **Show**   ☑
>
> ⦿ Non-blob Attribute   ○ Blob Attribute
>
> **Description** [   ]
>
> * Required fields
>
> 💾 Save ✖ Cancel

2 Specify the **Name** (required) you will use to identify the attribute in your application code.

3 Check the **Show** box to make the attribute visible to users; uncheck the Show box for tracking, usage statistics, and so on.

4 Specify the attribute value type as **Non-blob** or **Blob** (see "Attribute properties" on page 79).

5 Specifiy the **Description**. This field is searchable using methods on the `EbiUserInfo` class (see EbiUserQuery in the *API Reference*).

6 Click **Save** to add the attribute to the profile directory.

➢ **To remove an attribute:**

◆ Click **remove** next to the attribute.

IV **Reference**

Describes the JSP tag library for user management functions

- Chapter 10, "Framework Tag Library"

10 **Framework Tag Library**

This chapter provides reference information about the user management framework tags (**FrameworkTag.jar**).

📖 For background information, see the chapter on using the exteNd Director tag libraries in *Developing exteNd Director Applications*.

- addUserToGroup
- createGroup
- createUser
- getGroupList
- getResource
- getUserID
- getUserInfo
- getUserList
- getUserPreference
- login
- logoff
- removeGroup
- removeUserFromGroup
- setUserPassword
- userInGroup
- userLoggedIn

addUserToGroup

Description   Adds a specified user to a group.

Wrapping   This tag wraps the addMember() method on the
EbiRealmGroupDelegate interface.

Syntax   `<prefix:addUserToGroup id="ID" userid="userid" groupid="groupid" />`

Attribute	Required?	Request-time expression values supported?	Description
id	No	No	Specifies the name of the variable that will be used to store the result of the operation. If the operation is successful, this variable holds a value of true. If the operation fails, this variable holds a value of false.
			If no value is specified, a default ID of **addedusertogroup** is used.
userid	Yes	Yes	Specifies the ID for the user to add.
groupid	Yes	Yes	Specifies the ID for the group to which the user will be added.

Example
```
<% taglib uri="/fw" prefix="fw" %>
...
<fw:addUserToGroup id="result" userid="User1" groupid="Group1" />
<%=pageContext.getAttribute("result")%>
```

createGroup

Description   Creates a new group.

Wrapping   This tag wraps the addGroup() method on the EboDirectoryManager
interface.

Syntax

```
<prefix:createGroup id="ID" groupid="groupid" />
```

Attribute	Required?	Request-time expression values supported?	Description
id	No	No	Specifies the name of the variable that will be used to store the result of the operation. If the operation is successful, this variable holds a value of true. If the operation fails, this variable holds the string **Group already exists**.
			If no value is specified, a default ID of **addedgroup** is used.
groupid	Yes	Yes	Specifies the ID for the new group.

Example

```
<% taglib uri="/fw" prefix="fw" %>
...
<fw:createGroup id="result" groupid="Group1" />
<%=pageContext.getAttribute("result")%>
```

createUser

Description   Creates a new user and a new default profile for that user. This tag creates the new user in the realm that was specified in the configuration for the Directory subsystem.

Wrapping   This tag wraps the createUser() method on the EbiUserDelegate interface.

Syntax

```
<prefix:createUser id="ID" userid="userid" password="password"
firstname="firstname" lastname="lastname" emailaddress="emailaddress"/>
```

Attribute	Required?	Request-time expression values supported?	Description
id	No	No	Specifies the name of the variable that will be used to store the result of the operation. If the operation is successful, this variable holds a value of true. If the operation fails, an exception is thrown.
			If no value is specified, a default ID of **addeduser** is used.
userid	Yes	Yes	Specifies the ID for the new user.
password	Yes	Yes	Specifies the password for the new user.
firstname	No	No	Specifies the first name for the new user.
lastname	No	No	Specifies the last name for the new user.
emailaddress	No	No	Specifies the e-mail address for the new user.

Example

```
<% taglib uri="/fw" prefix="fw" %>
...
<fw:createUser id="result" userid="User1" password="MyPassword"
firstname="John" lastname="Smith"/>
<%=pageContext.getAttribute("result")%>
```

getGroupList

Description   Returns a list of groups for the framework. The objects returned are of type
java.security.acl.Group. They can be cast to Group objects, or a more specific subclass
if necessary.

Wrapping   This tag wraps the getGroups() method on the EbiDirectoryManager
interface.

```
<prefix:getGroupList id="ID" iterate="iterate" />
```

Attribute	Required?	Request-time expression values supported?	Description
id	No	No	Specifies the name of the variable that will be used to store the list of groups.
			If no value is specified, a default ID of **grouplist** is used.
iterate	Yes	No	Specifies a boolean value (true or false) that indicates whether this tag will operate as a body tag so that each row can be processed separately.
			If the iterate attribute is set to true, the following value can be accessed from within the getGroupList tag:
			◆ groupid
			This is the group name as referenced in EbiRealmGroup.
			The variable groupid has a scope of NESTED.
			If the iterate attribute is set to false, this tag will operate as a nonbody tag that returns an object of type List that contains a list of objects of type EbiGroupInfo.

Examples

This example shows how to use the getGroupList tag with the iterate attribute set to **true**:

```
<%@ taglib uri="/fw" prefix="fw" %>
<html>
<head>
</head>
<body>
<fw:login userid="admin" password="admin"/>
<fw:getGroupList iterate="true">
Group Name = <%=groupid%><br/>
<p/>
</fw:getGroupList>
<fw:logoff />
</body>
</html>
```

This example shows how to use the getGroupList tag with the iterate attribute set to **false**:

```
<%@ taglib uri="/fw" prefix="fw" %>
<html>
<head>
</head>
<body>
<fw:login userid="admin" password="admin"/>
<fw:getGroupList iterate="false"/>

<%= ((java.util.List)pageContext.getAttribute("grouplist")).size()
%> = the size of the list...
<fw:logoff />
</body>
</html>
```

getResource

Description   Retrieves resource set objects by string path. If the id attribute **is not set**, the resource will be assumed to be a string and returned inline. If the id attribute **is set**, the returned object will be returned via the variable named in the id attribute.

Syntax   `<prefix:getResource resourcePath="resourcePath" returnType="returnType" id="ID" />`

Attribute	Required?	Request-time expression values supported?	Description
resourcePath	Yes	Yes	The path to a resource.
returnType	No	No	Specifies the data type for the requested resource. If the id attribute is set, the return type must be set to one of the following values: ◆ document (when the return type is an org.w3c.dom.Document) ◆ string (when the return type is String) ◆ bytes (when the return type is byte[ ]) If the id attribute is not set, the resource will be assumed to be a string and returned inline.

Attribute	Required?	Request-time expression values supported?	Description
id	No	No	Specifies the name of the variable that will be used to store the object.
			If no value is specified, a default id of **resource** is used.

getUserID

Description   Retrieves the user ID for the current user. If the current user is not logged in, the tag returns **anonymous**.

Wrapping   This tag wraps the getUserID() method on the EboDirectoryHelper class.

Syntax   `<prefix:getUserID id="ID" />`

Attribute	Required?	Request-time expression values supported?	Description
id	No	No	Specifies the name of the variable that will be used to store the user ID.
			If no value is specified, a default id of **userid** is used.

Example
```
<% taglib uri="/fw" prefix="fw" %>
...
<fw:getUserID id="user" />
<%=pageContext.getAttribute("user")%>
```

getUserInfo

Description   Retrieves information about a particular user. The object returned is of type EbiUserInfo. If you don't need the object, you can access some commonly used attributes directly from the page context.

Wrapping   This tag wraps the getUserInfoByUserID() method on the EbiUserDelegate interface.

Syntax

```
<prefix:getUserID id="ID" userid="userid" />
```

Attribute	Required?	Request-time expression values supported?	Description
id	No	No	Specifies the name of the variable that will be used to store the user information object.
			If no value is specified, a default ID of **userinfo** is used.
userid	Yes	No	Specifies the ID of a user.
userid	No	Yes	Gets the user ID from the context.
firstname	No	Yes	Gets the user firstname from the context (if not null).
lastname	No	Yes	Gets the user lastname from the context (if not null).
emailaddress	No	Yes	Gets the user emailaddress from the context (if not null).

Examples   This example shows how to get an attribute from the EbiUserInfo object:

```
<% taglib uri="/fw" prefix="fw" %>
...
<fw:getUserInfo userid="User1" />
<%=pageContext.getAttribute("userinfo")%>
Hello, <%=
((com.sssw.fw.usermgr.api.EbiUserInfo)pageContext.getAttribute("use
rinfo")).getUserFirstName() %>
```

This example shows how to access attributes from the context:

```
<% taglib uri="/fw" prefix="fw" %>
...
<fw:getUserInfo userid="User1" />
<%=pageContext.getAttribute("userid") %>
<%=pageContext.getAttribute("firstname") %>
<%=pageContext.getAttribute("lastname") %>
<%=pageContext.getAttribute("emailaddress") %>
```

getUserList

Description   Returns a list of users for the framework. The objects returned are of type EbiUserInfo.

Wrapping   This tag wraps the getUsers() method on the EbiDirectoryDelegate interface.

Syntax   `<prefix:getUserList id="ID" iterate="iterate" />`

Attribute	Required?	Request-time expression values supported?	Description
id	No	No	Specifies the name of the variable that will be used to store the list of users.
			If no value is specified, a default id of **userlist** is used.
iterate	Yes	No	Specifies a boolean value (true or false) that indicates whether this tag will operate as a body tag so that each row can be processed separately.
			If the iterate attribute is set to true, the following values can be accessed from within the getUserList tag:
			◆ userid
			◆ uuid
			Each of these variables has a scope of NESTED.
			If the iterate attribute is set to false, this tag will operate as a nonbody tag that returns an object of type List that contains a list of objects of type EbiUserInfo.
realmName	No	No	Specifies a directory realm.
			If no value is specified, the default primary realm is used.

Examples   This example shows how to use the getUserList tag with the iterate attribute set to **true**:

```
<%@ taglib uri="/fw" prefix="fw" %>
<html>
```

```
<head>
</head>
<body>
<fw:login userid="admin" password="admin"/>
<fw:getUserList iterate="true">
UserID = <%=userid%><br/>
UUID = <%=uuid%><br/>
<p/>
</fw:getUserList>
<fw:logoff />
</body>
</html>
```

This example shows how to use the getUserList tag with the iterate attribute set to
false:

```
<%@ taglib uri="/fw" prefix="fw" %>
<html>
<head>
</head>
<body>
<fw:login userid="admin" password="admin"/>
<fw:getUserList iterate="false"/>

<%= ((java.util.List)pageContext.getAttribute("userlist")).size()
%> = the size of the list...
<fw:logoff />
</body>
</html>
```

getUserPreference

Description   Retrieves the user preference object for the ID passed in or the current user if no ID is
provided. This tag is for getting and setting custom preferences. Preferences for portal
objects should be done through the portal manager and the tags that support those
functions such as getUserComponentInfoTag.

Wrapping   This tag wraps the getUserPreference() method on the
EbiUserPreferenceDelegate interface.

Syntax   `<prefix:getUserProfile profilename="profilename" userid="userid"`
`id="ID" />`

Attribute	Required?	Request-time expression values supported?	Description
elementType	Yes	Yes	Specifies an element type.

Attribute	Required?	Request-time expression values supported?	Description
elementID	Yes	Yes	Specifies an element ID.
id	No	No	Specifies the name of the variable that will be used to store the user preference object. This object is of type EbiUserPreferenceInfo. If no value is specified, a default id of **userPreference** is used.
userIID	No	Yes	Specifies the UUID for a particular user.

login

Description

Logs a user in to exteNd Director.

Wrapping This tag wraps the authUserPassword() method on the EbiDirectoryDelegate interface.

Syntax

```
<prefix:login userid="userid" password="password" id="ID" />
```

Attribute	Required?	Request-time expression values supported?	Description
userid	Yes	Yes	Specifies the ID for the user logging in.
password	Yes	Yes	Specifies the password for the user logging in.

Attribute	Required?	Request-time expression values supported?	Description
id	No	No	Specifies the name of the variable that will be used to store the result of the operation. If the login attempt is successful, this variable holds a value of true. If the login attempt fails, this variable holds a value of false. A login may fail for the following reasons: ◆ A user has already been logged in to the current session ◆ The user ID is not recognized ◆ The user ID/password combination is not valid If no value is specified, a default id of **success** is used.

Example

```
<% taglib uri="/fw" prefix="fw" %>
...
<fw:login userid="admin" password="admin" id="result" />
<%=pageContext.getAttribute("result")%> = the result of the login...
```

logoff

Description   Logs off the current exteNd Director user.

Wrapping   This tag wraps the logoff() method on the EbiSession interface.

Syntax

```
<prefix:logoff id="ID" />
```

Attribute	Required?	Request-time expression values supported?	Description
id	No	No	Specifies the name of the variable that will be used to store the result of the operation. If the logoff attempt is successful, this variable holds a value of true. If the logoff attempt fails, this variable holds a value of false.
			If no value is specified, a default id of **logoff** is used.

Example

```
<% taglib uri="/fw" prefix="fw" %>
...
<fw:logoff id="result" />
<%=pageContext.getAttribute("result")%> = the result of the logoff...
```

removeGroup

Description   Deletes a group.

Wrapping   This tag wraps the removeGroup() method on the EbiDirectoryManager interface.

Syntax

```
<prefix:removeGroup groupid="groupid" id="ID" />
```

Attribute	Required?	Request-time expression values supported?	Description
groupid	Yes	Yes	Specifies the ID of the group you want to delete.

Attribute	Required?	Request-time expression values supported?	Description
id	No	No	Specifies the name of the variable that will be used to store the result of the operation. If the operation is successful, this variable holds a value of true. Otherwise, an exception is thrown.
			If no value is specified, a default id of **deletedgroup** is used.

Example   This example shows how to use the removeGroup tag:

```
<%@ taglib uri="/fw" prefix="fw" %>
<html>
<head>
</head>
<body>
<fw:login userid="admin" password="admin"/>
<fw:removeGroup groupid="Group1"/>
<fw:logoff />
</body>
</html>
```

removeUserFromGroup

Description   Removes a specified user from a group.

Wrapping   This tag wraps the removeMember() method on the EbiRealmGroupDelegate interface.

| | *Syntax* | `<`*`prefix`*`:removeUserFromGroup id="`*`ID`*`" userid="`*`userid`*`" groupid=`
`"`*`groupid`*`" />` |

Attribute	Required?	Request-time expression values supported?	Description
id	No	No	Specifies the name of the variable that will be used to store the result of the operation. If the operation is successful, this variable holds a value of true. If the operation fails, this variable holds a value of false.
			If no value is specified, a default id of **removeduserfromgroup** is used.
userid	Yes	Yes	Specifies the ID for the user to remove.
groupid	Yes	Yes	Specifies the ID for the group from which the user will be removed.

Example

```
<% taglib uri="/fw" prefix="fw" %>
...
<fw:removeUserFromGroup id="result" userid="User1" groupid="Group1" />
<%=pageContext.getAttribute("result")%>
```

setUserPassword

Description   Changes the password for a specified user.

Wrapping   This tag wraps the modifyUser() method on the EbiUserDelegate
interface.

Syntax   `<prefix:setUserPassword id="ID" userid="userid" passowrd="password" />`

Attribute	Required?	Request-time expression values supported?	Description
id	No	No	Specifies the name of the variable that will be used to store the result of the operation. If the operation is successful, this variable holds a value of true. Otherwise, an exception is thrown.
			If no value is specified, a default id of **setpassword** is used.
userid	Yes	Yes	Specifies the ID for the user whose password will be modified.
password	Yes	Yes	Specifies the new password.

Example
```
<% taglib uri="/fw" prefix="fw" %>
...
<fw:setUserPassword id="result" userid="User1" password="MyPassword" />
<%=pageContext.getAttribute("result")%>
```

userInGroup

Description   Determines whether a specified user is in a particular group.

Wrapping   This tag wraps the isMember() method on the EbiRealmGroupDelegate interface.

Syntax   `<prefix:userInGroup groupid="groupid" userid="userid" id="ID" />`

Attribute	Required?	Request-time expression values supported?	Description
groupid	Yes	No	Specifies the ID of a group.
userid	Yes	No	Specifies the ID of a user.

Attribute	Required?	Request-time expression values supported?	Description
id	No	No	Specifies the name of the variable that will be used to store the result of the operation. If the user is in the specified group, this variable holds a value of true. If the user is not in the group, this variable holds a value of false. If no value is specified, a default id of **useringroup** is used.

Example

```
<% taglib uri="/fw" prefix="fw" %>
...
Is in group...
<fw:userInGroup groupid="ContentAdmins" userid="JSmith" id="result" />
<%=pageContext.getAttribute("result")%>
```

userLoggedIn

Description   Determines whether the current user is logged in to the session.

Wrapping This tag wraps the isAnonymous() method on the EboDirectoryHelper class.

Syntax   `<prefix:userLoggedIn id="ID" />`

Attribute	Required?	Request-time expression values supported?	Description
id	No	No	Specifies the name of the variable that will be used to store the result of the operation. If the current user is already logged in, this variable holds a value of true. If the user has not yet logged in, this variable holds a value of false. If no value is specified, a default id of **loggedin** is used.

Example

```
<% taglib uri="/fw" prefix="fw" %>
...
Is logged in...
<fw:userLoggedIn />
<%=pageContext.getAttribute("loggedin")%>
```

Index