BEA WebLogic Workshop™ Help

Version 8.1 SP4
December 2004

Copyright

Copyright © 2003 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

This software and documentation is subject to and made available only pursuant to the terms of the BEA Systems License Agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the agreement. This document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent, in writing, from BEA Systems, Inc.

Use, duplication or disclosure by the U.S. Government is subject to restrictions set forth in the BEA Systems License Agreement and in subparagraph (c)(1) of the Commercial Computer Software−Restricted Rights Clause at FAR 52.227−19; subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227−7013, subparagraph (d) of the Commercial Computer Software−−Licensing clause at NASA FAR supplement 16−52.227−86; or their equivalent.

Information in this document is subject to change without notice and does not represent a commitment on the part of BEA Systems. THE SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, BEA Systems DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIAL IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

Trademarks or Service Marks

BEA, Jolt, Tuxedo, and WebLogic are registered trademarks of BEA Systems, Inc. BEA Builder, BEA Campaign Manager for WebLogic, BEA eLink, BEA Liquid Data for WebLogic, BEA Manager, BEA WebLogic Commerce Server, BEA WebLogic Enterprise, BEA WebLogic Enterprise Platform, BEA WebLogic Enterprise Security, BEA WebLogic Express, BEA WebLogic Integration, BEA WebLogic Personalization Server, BEA WebLogic Platform, BEA WebLogic Portal, BEA WebLogic Server, BEA WebLogic Workshop and How Business Becomes E−Business are trademarks of BEA Systems, Inc.

All other trademarks are the property of their respective companies.

Table of Contents

Table of Contents

WebLogic Workshop Security Overview

The following overview sets out the aims of security and the security technologies available in WebLogic Workshop.

Security Goals

All security technologies are designed to achieve three basic goals.

1. Authentication of participants

   When a participant is authenticated it means that there is some assurance that the participant really is who they say they are. Different scenarios call for different levels of authentication. In some cases, only the web resource needs to be authenticated, while the client can remain largely anonymous. For example, if your web resource takes customer credit card numbers, you want your customers to have piece of mind that they are providing their card numbers to you, and not some malicious third party. But your customers may remain anonymous. In other cases, the web resource will want proof of identity from its clients. For example, if a bank provided online access to its customer's checking accounts, the bank should require some form of client authentication from those parties who want to access the online accounts.
2. Confidential communication

   Data transmission is confidential when only the intended recipient can read the data.
3. Integrity of transmitted data

   Data integrity means that the data has not been altered in the process of transmission. (When using transport security, there is generally no need to take special measures to ensure data integrity. This is because the encryption processes used by SSL ensures data integrity.)

The topics below provide detailed information to help you implement a security strategy for your WebLogic Workshop application.

WebLogic Workshop Security Technologies

WebLogic Workshop offers three main areas of security technology:

- transport security
- web service security
- role−based security

Transport security refers to the mechanisms used to enable the http protocol to operate over a secure transport connection. Transport security lets you secure your web resources through SSL, username/password authentication, and client digital certificates.

An advantage of transport security is that is well known and relatively easy to implement. A disadvantage is that data is secured only while it is in transport over the wire. The transport security mechanisms no longer apply once the data has reached the recipient, so if the data is logged on the recipient's machine, its confidentiality may be at risk. This is not the case with Web service security, where the security mechanisms are applied to the data itself.

For detailed information on implementing SSL and client certificates see Transport Security. For detailed information on implementing username/password authentication see Username/Password Authentication.

Web service security provides message−level security for web services through an implementation of the Oasis Web Service Security standard. Web service security, often referred to as "WS−Security" or simply "WSSE", lets you secure the SOAP messages that pass between web services with security tokens (username and password), digital signatures, and encryption.

An advantage of WS−Security is that the security mechanisms are applied to the SOAP messages that pass between web services. So WS−Security security mechanisms apply both while the SOAP message is in transit and once the message has arrived at the recipient's machine.
The disadvantages of WS−Security are that it is not a widely used form of security and it is relatively more difficult to implement than the analogous transport security technologies. For example, users must be familiar with some of the inner workings of the Public Key Infrastructure (PKI) to effectively use WS−Security's encryption and digital signature technologies.

For detailed information on implementing see Web Service Security.

Role−based security lets you secure a web resource by restricting access to only those users who have been granted a particular security role. For detailed information on see Role−Based Security.

Topics Included in This Section

Transport Security

These topics explain how to protect a web resource using SSL.

Web Service Security

These topics explain how to implement Web Service security to secure the SOAP messages that pass between web services.

Role−Based Security

These topics explain how to restrict access to a web resource to users within specific roles.

Authentication

These topics explain how to login users with a username and password.

Samples

The following samples illustrate WebLogic Workshop's security technologies.

Transport Security Samples

BasicAuthentication.jws Sample

HelloWorldSecureClient.jws Sample

ClientCert Sample

Web Service Security Samples

WS−Security Callback Sample

WS−Security ReqResp Sample

WS−Security UserToken Sample

Role−Based Security Samples

VeriCheck.jws Sample

EJB Security Sample

Login Samples

Login Samples

Transport Security

Transport security refers to a group of security technologies that ensure the authenticity of both clients and servers and the integrity and confidentiality of data passed between web servers and their clients.

In most cases, transport security alone is sufficient to secure a web resource such as a web application or web service; but there is another security option available specifically for web services. For detailed information see Web Service Security.

Transport Security Strategies

Transport security offers three basic strategies for achieving the three main security goals: authentication of participants, confidential communication and data integrity. See WebLogic Workshop Security Overview for a description of these security goals.

One−way SSL

One−way SSL offers two primary benefits. First it authenticates the identity of the web server. Second, it ensures confidential communication by encrypting the messages between the client and the server. The "one−way" in one−way SSL refers to the fact that only the identity of the server is authenticated, not the client. You should use one−way SSL when you want to ensure private communication, but where the identity of the client is not a critical factor.

One−way SSL with Basic Authentication

Basic authentication ensures the identity of clients by requiring a username and password. Basic authentication should always be used together with one−way SSL, otherwise the username and password could be intercepted by a malicious third party. You should use one−way SSL when you want to ensure the identities of both the client and server. For details on implementing a basic authentication process, see Basic Authentication.

Two−way SSL

Two−way SSL combines server authentication, encryption of data, and client authentication through client digital certificates.

Topics Included in This Section

One−way SSL

Two−way SSL

Using Controls to Access Transport Secured Resources

Related Topics

Security

One−Way SSL

The following topic explains how to configure a WebLogic Workshop web application for one−way SSL.

One−way SSL sets up a secure connection between a web server and client by requiring the server to present a digital certificate to its clients and by encrypting the data passed between the client and server. The main goals of one−way SSL are the integrity and confidentiality provided by the encryption and the authentication of the server provided by the digital certificate. Note that authentication of the client is not a goal of one−way SSL in itself. To achieve authentication of the client, you should supplement one−way SSL with either basic authentication or by configuring your web resource for two−way SSL.

Securing a Web Resource with One−Way SSL

To secure a web resource with one−way SSL, you must (1) obtain a digital certificate from a trusted authority or create your own digital certificate, (2) ensure that the WebLogic Server is SSL enabled, (3) define a web resource as a protected web resource, (4) expose the web resource over an HTTPS enabled server port, and (5) configure WebLogic Server to encrypt the data traffic with the web resource.

1. Digital Certificates

To enable SSL on a production server, you first need to obtain a private−public key pair and a certificate from a trusted third−party certificate authority. For detailed information, see Obtaining Private Keys, Digital Certificates and Trusted Certificate Authorities in the WebLogic Server 8.1 documentation. (As a variation you may also generate your own digital certificates to present to clients, but most web applications use a digital certificate obtained from a third party.) You will also need to store your digital certificate in the appropriate repositories in WebLogic Server. For detailed information see Storing Private Keys, Digital Certificates, and Trusted Certificate Authorities in the WebLogic Server 8.1 documentation.

2. Configuring WebLogic Server

WebLogic Server is by default configured to support one−way SSL. For detailed information see Configuring SSL in the WebLogic 8.1 documentation.

3. Defining a Protected Web Resource

You define a web resource, such as a web application or web service, as a *protected* resource by placing a security constraint on that resource. Security constraints are specified by <security−constraint> XML elements in the web.xml configuration file in the WEB−INF directory.

Web resources are defined in terms of the URL where they reside. In the following example, the web service HelloWorldSecure.jws is defined as protected because the URL where it resides ("/security/transport/helloWorldSecure/HelloWorldSecure.jws/") is defined as protected.

```
<security−constraint>
<display−name>
    Security Constraint for HelloWorldSecure.jws
</display−name>
        <web−resource−collection>
    <web−resource−name>HelloWorldSecure.jws</web−resource−name>
    <description>A web service secured by SLL.</description>
```

```
            <!--
            Defines the scope of the web resource to be secured with SSL.
            Secure all methods calls to the HelloWorldSecure web service.
            -->
            <url-pattern>/security/transport/helloWorldSecure/HelloWorldSecure.jws/*</url-patte
            <http-method>GET</http-method>
            <http-method>POST</http-method>
                </web-resource-collection>
    </security-constraint>
```

If you want to protect the whole folder where the HelloWorldSecure web service resides, you would specify the following URL.

```
<url-pattern>/security/transport/helloWorldSecure/*</url-pattern>
```

Using the <url−pattern> element you can restrict access to an entire web application, a folder or a particular file within the web application.

For example, suppose that you have written a web project called myWebProject containing a web service called myWebService containing a method called myWebServiceMethod.

The following <url−pattern> element declares the entire project as protected.

```
<url-pattern>/*</url-pattern>
```

The following <url−pattern> element declares the webservices folder as protected.

```
<url-pattern>webservices/*</url-pattern>
```

The following <url−pattern> element declares that the web service myWebService should be protected.

```
<url-pattern>webservices/myWebService.jws</url-pattern>
```

4. Exposing a Web Resource Over an HTTPS−enabled Port

You configure WebLogic Server to expose your web resource over an HTTPS−enabled port by editing the wlw−config.xml file, found in the WEB−INF directory of the WebLogic Workshop project where your web resource resides; or by editing the wlw−runtime−config.xml file, found in your application's META−INF directory. Note that we generally recommend that you use the wlw−runtime−config.xml file rather than the wlw−config.xml file, since values specified in the wlw−config.xml file because they are hard−coded into the EAR file that is deployed to production servers and cannot be overridden at runtime.

The example, wlw−config.xml file below shows how to expose the HelloWorldSecure web service on the HTTPS port, in this case, specified as port 7002 by the <https−port> element. Note that this does not force your resource to be exposed exclusively on the HTTPS port. Users will be able to access the resource both over the default port, specified by the <protocol> element , and the HTTPS port. You *enforce* the HTTPS port by placing a <transport−guarantee> element in the resource's security constraint (see step 5 below).

```
 <http-port>7001</http-port>
 <https-port>7002</https-port>
 <service>
     <class-name>security.transport.helloWorldSecure.HelloWorldSecure</class-name>
```

```
    <protocol>https</protocol>
</service>
```

In order to expose different services on differently−enabled ports, add a <service> element with child <class−name> and <protocol> elements. The example wlw−config.xml file below specifies that the HelloWorld web service should use the HTTP−enabled port and that the web service HelloWorldSecure should use the HTTPS−enabled port.

```
<http−port>7001</http−port>
<https−port>7002</https−port>
<service>
    <class−name>HelloWorld</class−name>
    <protocol>http</protocol>
</service>
<service>
    <class−name>security.transport.helloWorldSecure.HelloWorldSecure</class−name>
    <protocol>https</protocol>
</service>
```

5. Ensuring Encryption of Transmitted Data

To encrypt the traffic with the protected web service by include a <transport−guarantee> element with the value CONFIDENTIAL in the security constraint. This enforces the use of the HTTPS port specified in step 4 above.

```
<security−constraint>
<display−name>
    Security Constraint for HelloWorldSecure.jws
</display−name>
        <web−resource−collection>
    <web−resource−name>BasicAuthentication.jws</web−resource−name>
    <description>A web service secured by SLL and basic authentication</description>
    <!−−
    Defines the scope of the web resource to be secured with SSL.
    Secure all methods calls to the HelloWorldSecure web service.
    −−>
    <url−pattern>/security/transport/helloWorldSecure/HelloWorldSecure.jws/*</url−patte
    <http−method>GET</http−method>
    <http−method>POST</http−method>
        </web−resource−collection>
<!−−
Encrypt the traffic between the client and this web resource.
−−>
<user−data−constraint>
    <transport−guarantee>CONFIDENTIAL</transport−guarantee>
</user−data−constraint>
</security−constraint>
```

For detailed information about the syntax used in a the web.xml configuration file, see web.xml Deployment Descriptor Elements in the WebLogic 8.1 documentation. See especially the documentation for the <security−constraint>.

Related Topics

Transport Security

Security

wlw−config.xml

Two−Way SSL

As in one−way SSL, in two−way SSL the server presents a digital certificate to the client; unlike one−way SSL, in two−way SSL the client must present a digital certificate to the server before the SSL session is established.

To secure a web resource using two−way SSL, follow the procedures set out for securing a resource with one−way SSL. See One−way SSL for procedure details. To complete the process you must take the additional step of configuring WebLogic Server for two−way SSL, either through the WebLogic Server console or by directly editing the server's config.xml file.

For information on configuring two−way SSL through the WebLogic Server console see Configuring Two−Way SSL in the WebLogic Server 8.1 documentation.

You can also configure WebLogic Server for two−way SSL by directly editing the server's config.xml file. (config.xml is in your server's root directory. For instance, the config.xml file for SamplesApp, which is part of the Workshop domain, is located at BEA_HOME\weblogic81\samples\workshop\config.xml.)

You *enable* two−way SSL by setting the TwoWaySSLEnabled attribute to true in the <SSL> element of the config.xml.

```
<SSL Enabled="true" TwoWaySSLEnabled="true" IdentityAndTrustLocations="KeyStores"
    ListenPort="7002" Name="cgServer"/>
```

Note that enabling two−way SSL does not enforce two−way SSL. If you merely enable two−way SSL, the server will request a digital certificate from the client, but if one is not provided, the SSL session will continue.

You *enforce* two−way SSL by setting both the TwoWaySSLEnabled and ClientCertificateEnforced attributes to true.

```
<SSL TwoWaySSLEnabled="true" ClientCertificateEnforced="true" Enabled="true" IdentityAn
            ListenPort="7002" Name="cgServer"/>
```

For syntax details on the <SSL> element of config.xml see SSL in the WebLogic Server 8.1 documentation.

When a web resource resides in a server that enforces client certificates, the client can send the certificate by modifying properties on the resource's control file. See Using Controls to Access Transport Secured Resources for details.

Testing Two−Way SSL

When testing Client certification, you may want to use a browser other than the Workshop Test Browser, since this will allow you to control which client certificates are used. To change the browser used in testing, select ***Tools***−−>***IDE Properties***. In the ***IDE Properties*** dialog, click ***Browser***. In the ***Browser path*** field, enter the path to the browser you wish to use, for example, C:\Program Files\Internet Explorer\iexplore.exe. Also, ensure that the check box Use integrated browser for debugging is unchecked.

Related Topics

WebLogic Workshop Documentation

Security

Transport Security

One−way SSL

WebLogic Server 8.1 Documentation

Configuring Two−Way SSL

Using Controls to Access Transport Secured Resources

This topic explains to use WebLogic Workshop controls to access web resources secured with Transport security.

Specifying the HTTPS Protocol

Web resources secured with one−way and two−way SSL communicate with clients over HTTPS enabled ports. Clients using WebLogic Workshop controls to communicate with these resources must use the same port to exchange data with the resource.

You specify a web service control to use the appropriate HTTPS port in the control's @jc:location annotation. For example the following control communicates with the Credit Card Report web service on the HTTPS enabled port 7002.

```
/**
 * @jc:location http−url="https://localhost:7002/CreditCardReport/webservice/CreditCardReport.j
 */
public interface CreditCardReportControl extends com.bea.control.ControlExtension, com.bea.cont
```

Specifying Username and Password

When using a control to access a web resource requiring a username and password, set the username and password properties on the web services's control file with the setUsername() and setPassword() methods.

In the following example the VeriCheck web service calls the Bank web service via the Bank's control file.

```
public class VeriCheck implements com.bea.jws.WebService
{
    /**
     * @common:control
     */
    private security.roleBased.BankControl bankControl;


    public void checkForSufficientBalance(String checkingAccountID, int amount)
    {
        // Use the following username and password when calling the Bank web service.
        bankControl.setUsername("VeriCheck");
        bankControl.setPassword("aeraeraer");

        // Check the account for sufficient balance.
        bankControl.doesAccountHaveSufficientBalance(checkingAccountID, amount);
    }
}
```

Providing a Client Digital Certificate

When using a control to access a web resource that requires a digital certificates from clients, you can set properties on the control to provide the certificate.

In the following example, assume that WebServiceB requires a digital certificate from clients. WebServiceA can provide the necessary digital certificate by setting properties in WebServiceB's control file in the following way.

```
public class WebServiceA implements com.bea.jws.WebService
{
   /** @common:control */
    security.transport.clientCert.WebServiceBControl ctrl;

    /**
     * @common:operation
     */
    public void invokeWebServiceB()
    {
        /**
         * Enable client certificates for this web service.
         */

        ctrl.useClientKeySSL( true );

        /**
         * Specify the location and password for the keystore where the client certificates res
         *
         * The following method call to setKeystore is, strictly speaking,
         * unnecessary, since SamplesApp is already configured to use the
         * default keystore DemoIdentity.jks.
         * It is included to show how you would override the
         * location for another, non-default keystore.
         */
        String sep = File.separator;
        ctrl.setKeystore(Home.getPath() + File.separator + "lib" +
            File.separator + "DemoIdentity.jks", "DemoIdentityPassPhrase" );

        /**
         * Specify the alias in the keystore for both the client SSL certificate
         * and the client private key.  (The certificate and the private key must
         * be stored under the same alias in the same key store.)
         * The second parameter specifies the password required to access
         * the keystore.
         */
        ctrl.setClientCert("DemoIdentity", "DemoIdentityPassPhrase");

        /**
         * Invoke the requestCallback method on WebServiceB.
         * The client certificate specified above will be sent.
         */
        ctrl.requestCallback("WebServiceA");
    }

}
```

Setting and Overriding the Default Keystore

Note that you can set a default keystore location using the WebLogic Server console. See Configuring Keystores and SSL in the WebLogic Server 8.1 documentation.

You can override the default keystore location using the setKeystore(*path*, *password*) method (see the example above).

To override to a keystore type other than "JKS" (Java KeyStore), use setKeystore(*path*, *password*, *type*).

Related Topics

WebLogic Workshop Documentation

Transport Security

WebLogic Server 8.1 Documentation

Configuring Keystores and SSL

Web Service Security (WS−Security)

WebLogic Workshop provides message−level security for web services through an implementation of the WS−Security Oasis web service security standard. WS−Security lets you secure the SOAP messages passed between web services using (1) security tokens, (2) digital signatures, and (3) encryption.

Although WebLogic Workshop supports both transport and message−level security, it is generally not necessary to use both sorts of security to secure a web service. In most cases, developers should choose one or the other type of security to secure their web services.

Security Tokens

Security tokens are credentials used for authentication, authorization, or both. The WebLogic Workshop implementation supports two types of tokens. (1) Username and password tokens, and (2) X509 Binary Security Tokens.

When a X509 Binary Security Token accompanies an inbound SOAP message, the token is passed to the WebLogic Server security framework for authentication.

To include a Binary Security Token in an outbound SOAP message, you specify that you want to *sign* the outbound message. Signing the SOAP message will automatically include a X.509 BinarySecurityToken in the message. Note that sending a X509/Binary Security Token without signing the outbound SOAP message is not supported.

Digital Signatures

Digital signatures are used for two purposes: (1) to authenticate the identity of the sender and (2) to ensure the integrity of SOAP messages. If any part of an incoming SOAP message has been changed in transport, the signature validation performed by the recipient will fail. In WebLogic Workshop, if you require XML signatures for incoming SOAP messages, the SOAP body must be digitally signed to be processed by the web service.

By default, digital signatures are applied only to the body of outgoing SOAP messages. You must specifically provide for the signing of elements in the header. For details see <addtionalSignedElements> in the WS−Security reference documentation.

Encryption

Encryption is used to encrypt either the body of the SOAP message, the header, or both. If your web service requires encryption for incoming messages, then, at a minimum, the body of incoming SOAP messages must be encrypted.

For outgoing SOAP messages, encryption is applied only to the SOAP body by default. You must specifically provide for the encryption of elements in the header. For details see <addtionalEncryptedElements> in the WS−Security reference documentation.

Note that keys used in WebLogic Workshop's implementation of WS−Security must be RSA keys.

WSSE Policy Files

Web service security is controlled through WSSE policy files. WSSE policy files are XML files with a .WSSE file extension.

To *secure* a web service with web service security, you create a WSSE policy file and associate that file with your web service. All outbound and inbound SOAP messages are processed according to the policy called for in the WSSE file. Inbound messages are first checked for the necessary security measures called for in the policy file. If the inbound message is found to be appropriately secured, then the SOAP message, cleaned of its security enhancements, is passed to the web service for normal processing. Outbound messages go through the reverse process: they are enhanced with the security measures called for in the policy file before they sent out over the wire.

To *access* a web service secured with WS−Security, you create a policy file and associate that file with the web service control. The policy file you associate with a web service's control should match the policy file of the target web service. If the target web service requires encrypted incoming messages, then a control file targeting that web service should encrypt messages before they are sent to the web service.

For detailed information see Using WSSE Policy Files.

Related Topics

Applying WS−Security

WSSE Policy File Reference

Applying WS−Security

This topic explains how WS−Security policies are applied to the SOAP messages that pass between web services and web service controls.

A WS−Security is controlled in WS−Security policy files. One part of a WS−Security policy determines the security *requirements* for SOAP messages coming into a web service or web service control. This part of the policy determines what sorts of security mechanisms must be present in an inbound SOAP message in order to pass the security gate. The other part of a WS−Security policy determines the security *enhancements* to be added to outgoing SOAP messages before they are sent out over the wire. This part of the policy file determines the kinds of security mechanisms that a web service or web service control adds to SOAP messages (with an eye toward meeting the security requirements of the recipient).

WS−Security policies are configured in WSSE files, an XML file with the .WSSE extension. The <wsSecurityIn> element describes the security requirements for incoming SOAP messages; the <wsSecurityOut> element describes the security enhancements added to outgoing SOAP messages.

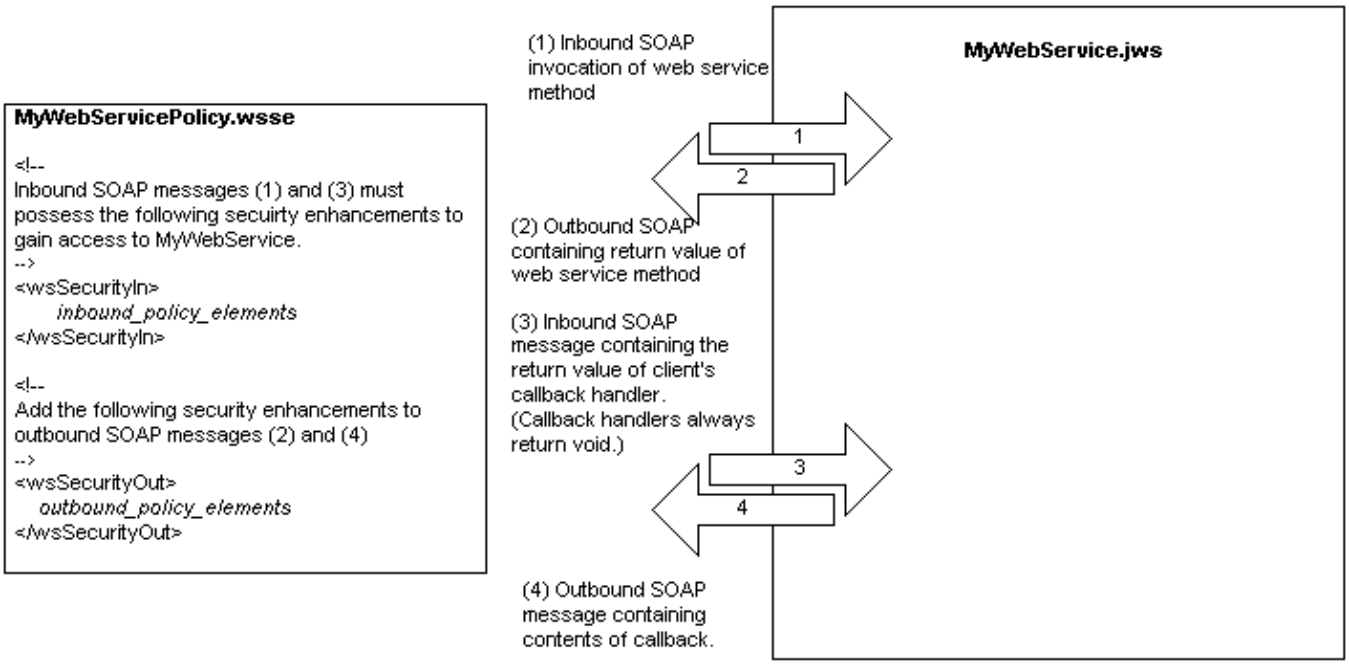
Applying WS−Security Policies to Web Services

To apply a WS−Security policy to a web service, add the annotations @jws:ws−security−service and @jws:ws−security−callback to the web service file.

```
/**
 * @jws:ws-security-service file="MyWebServicePolicy.wsse"
 * @jws:ws-security-callback file="MyWebServicePolicy.wsse"
 */
public class MyWebService implements com.bea.jws.WebService
```

If the web service communicates synchronously with its clients, you only need to use the @jws:ws−security−service annotation. If the web service sends callbacks to its clients, you must use both annotations.

The following illustration shows how policy files are applied to the SOAP messages sent and received by web services.



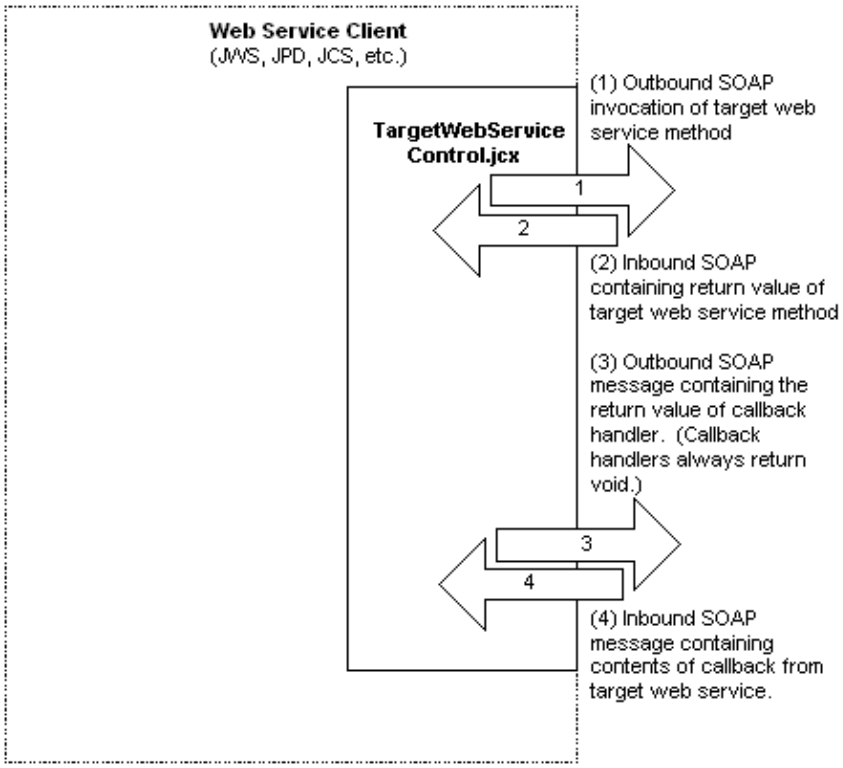
Applying WS−Security Policies to Web Service Controls

To apply a policy file to a web service control, use the control annotations @jc:ws−security−service and @jc:ws−security−callback.

```
/**
 * @jc:ws-security-service file="TargetControlPolicy.wsse"
 * @jc:ws-security-callback file="TargetControlPolicy.wsse"
 */
public interface TargetControl extends com.bea.control.ControlExtension, com.bea.control.Servic
```

If your web service control communicates synchronously with its target web service, you only need to use the @jc:ws−security−service annotation. If the control receives callbacks from its target service, you must use both annotations.

The following illustration shows how policy files are applied to the SOAP messages sent and received by web service controls.



TargetWebServiceControlPolicy.wsse

```
<!--
Inbound SOAP messages (2) and (4)  must
posses the following secuirty enhancements to
gain access to the web service control.
-->
<wsSecurityIn>
      inbound_policy_elements
</wsSecurityIn>

<!--
Add the following security enhancements to
outbound SOAP messages (1) and (3).
-->
<wsSecurityOut>
   outbound_policy_elements
</wsSecurityOut>
```

Related Topics

Security

Web Service Security

WS−Security Policy File Elements

WS−Security Policy File Reference (WSSE File Reference)

@jws:ws−security−service Annotation

@jws:ws−security−callback Annotation

@jc:ws−security−service Annotation

@jc:ws−security−callback Annotation

WS−Security Policy File Elements

This topic explains how SOAP messages are processed by the XML elements in a WS−Security policy file (WSSE file).

Username/Password Token

Username/password tokens serve to authenticate the sender of the SOAP message to the recipient. Note that username/password tokens should always be used in conjunction with either digital signatures or encryption, otherwise a malicious party could intercept the SOAP message and read the username and password.

Requiring Username/Password Tokens on Inbound SOAP Messages

You can specify that incoming SOAP messages must be accompanied by a username/password token. To pass the security gate set up by the WSSE file, the token must match a principle known to the WebLogic Server security provider.

To specify that a username/password token is required for inbound SOAP messages, use the following syntax.

```
<wsSecurityIn>
        <token tokenType="username"/>
</wsSecurityIn>
```

Enhancing Outbound SOAP Messages with Username/Password Tokens

To specify that outgoing SOAP messages include a username/password token use the following syntax. Note that the <password> element must contain the type attribute and that attribute must have the value TEXT.

```
<wsSecurityOut>
    <userNameToken>
        <userName>user1</userName>
        <password type="TEXT">password1</password>
    </userNameToken>
</wsSecurityOut>
```

Digital Signature

Digital signatures are used to authenticate the sender of the SOAP message and to ensure the integrity of the SOAP message (i.e., to ensure that the SOAP message is not altered while in transit). When a digital signature is applied to a SOAP message, a unique hash is produced from the message, and this hash is then encrypted with the sender's private key. When the message is received, the recient decrypts the hash using the sender's public key. (Note that the public key is contained in a digital certificate, provided either by a third party or by the sender himself. When the certificate is generated by the sender, the message is called "self−signed".) This serves to authenticate the sender, since only the sender could have encrypted the hash with his private key. It also serves to ensure that the SOAP message has not been tampered with while in transit, since the recipient can compare the hash sent with the message with a hash produced on the recipient's end.

Requiring Digital Signatures on Inbound SOAP Messages

To specify that a digital signature is required for inbound SOAP messages, use the following syntax.

```
    <wsSecurityIn>
            <signatureRequired>true</signatureRequired>
    </wsSecurityIn>
```

Enhancing Outbound SOAP Messages with Digital Signatures

To specify that outgoing SOAP messages be signed, use the following syntax.

```
<wsSecurityOut>
    <signatureKey>
        <alias>myAlias</alias>
        <password>myKeyStorePassword</password>
    </signatureKey>
</wsSecurityOut>

<keyStore>
    <keyStoreLocation>path_to_keystore</keyStoreLocation>
    <keyStorePassword>myKeyStorePassword</keyStorePassword>
</keyStore>
```

When signing an outbound message, the <keystore> element must be present so that the digital certificate (containing your public key) and your associated private key can be retrieved from the keystore.

Encryption

Encryption ensures confidential communication between the sender and the recipient. The sender uses the recipient's public key to encrypt the message. Only the recipient's private key can successfully decrypt the message ensuring that it cannot be read by third parties while in transit.

Requiring Digital Signatures on Inbound SOAP Messages

The following <encryptionRequired> element has two purposes. (1) It requires that incoming SOAP message must be encrypted with your public key and (2) it gives decryption instructions for the incoming message.

```
    <wsSecurityIn>
            <encryptionRequired>
                    <decryptionKey>
                            <alias>myAlias</alias>
                            <password>password_key_store</password>
                    </decryptionKey>
            </encryptionRequired>
</wsSecurityIn>
```

The <alias> and <password> elements are used to access the keystore where your decrypting private key is stored.

Enhancing Outbound SOAP Messages with Encryption

To encrypt outgoing messages with the intended recipient's public key, use the following syntax.

```
    <wsSecurityOut>
            <encryption>
                    <encryptionKey>
                            <alias>recipient_alias</alias>
                    </encryptionKey>
```

```
        </encryption>
</wsSecurityOut>

<keyStore>
        <keyStoreLocation>path_to_keystore</keyStoreLocation>
        <keyStorePassword>myKeystorePassword</keyStorePassword>
</keyStore>
```

Related Topics

Web Service Security

Securing WS−Security Passwords

When WS−Security policy files (WSSE files) are compiled, clear text passwords are embedded in the resulting CLASS files. Individuals with access to the production server, could learn these passwords. To mitigate this risk, two utilities are provided to encrypt these clear text passwords in the WSSE files before compilation. The resulting CLASS files will contain encrypted versions of the passwords.

Both utilities, generateSecretKeyFile and encryptWssePolicy, are located in the BEA_HOME/weblogic81/common/bin directory.

generateSecretKeyFile Utility

Use the utility generateSecretKeyFile to generate a secret key used to encrypt the passwords. When you run generateSecretKeyFile, it generates a key named pwdEncryptionInfo.key and places it in the BEA_HOME/weblogic81/common/bin directory.
Save the generated key pwdEncryptionInfo.key in your application's APP−INF/classes directory.

encryptWssePolicy Utility

Once you have generated a secret key, use the utility encryptWssePolicy to encrypt the passwords in the WSSE files.

The flag −s will encrypt all passwords in the WSSE files in place. For example, assuming that BEA_HOME is C:/bea, the following command will encrypt the passwords in the sample SamplesApp/WebServices/security/security/wsse/callback.

```
C:\bea\weblogic81\common\bin\encryptWssePolicy.cmd −s C:\bea\weblogic81\samples\workshop\Sample
 C:\bea\weblogic81\samples\workshop\SamplesApp\WebServices\security\wsse\callback\client\Target
 C:\bea\weblogic81\samples\workshop\SamplesApp\WebServices\security\wsse\callback\target\Target
```

When the WSSE files are compiled, the CLASS files will no longer show the passwords in clear text.

At runtime, WebLogic Server will decrypt the passwords using pwdEncryptionInfo.key when necessary.

Related Topics

Web Service Security

Calling a WSSE Enabled Web Service Through a Java Proxy Class

This topic explains how to call a WSSE enabled web service through its Java proxy class. The Java proxy provides an access point to a web service usable by Java code. (You can generate a Java proxy for a web service by selecting the ***Java Proxy*** link on that web service's Test View Overview Page.)

The following sections show typical clients for invoking proxies where the target web services are protected with WSSE user tokens, encryption, and signatures. You can use these clients as templates for building your own proxy clients.

Samples are located in the SamplesApp at BEA_HOME/weblogic81/samples/workshop/SamplesApp/ProxyClient/WSSE.

User Tokens

The following client class uses the weblogic.xml.security.UserInfo class to set the username and password token in the outgoing SOAP message sent through the proxy to the target web service.

MyWebServiceClient.java:

```
import java.util.List;
import java.util.ArrayList;
import javax.xml.namespace.QName;
import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.HandlerRegistry;
import weblogic.webservice.context.WebServiceSession;
import weblogic.webservice.context.WebServiceContext;
import weblogic.webservice.core.handler.WSSEClientHandler;
import weblogic.webservice.WLMessageContext;
import weblogic.xml.security.wsse.Security;
import weblogic.xml.security.wsse.Token;
import weblogic.xml.security.UserInfo;
import weblogic.xml.security.wsse.UsernameToken;
import weblogic.xml.security.wsse.SecurityElementFactory;
import weblogic.jws.proxies.*;

/*
 * This class shows how to call a web service protected with WSSE user token restrictions
 * through the web services's Java proxy.
 *
 * This class inserts username token by using UserInfo object
 */

public class MyWebServiceClient {

    public static void main(String[] args){

        try{
            /*
             * Instantiate the main proxy class. The proxy class has the same name as the
             * web service, with "_Impl" appended.
             */
            MyWebService myservice = new MyWebService_Impl("http://localhost:7001/WebServices/M
```

```
            WebServiceContext context = myservice.context();
            WebServiceSession session = context.getSession();

            /*
             * Registers a handler for the SOAP message traffic.
             */
            HandlerRegistry registry = myservice.getHandlerRegistry();
            List list = new ArrayList();
            list.add(new HandlerInfo(WSSEClientHandler.class, null, null));
            registry.setHandlerChain(new QName("hello"), list);

            /*
             * Set the username and password token for SOAP message sent from the client, throu
             * the proxy, to the web service.
             */
            UserInfo ui = new UserInfo("username", "password");
            session.setAttribute(WSSEClientHandler.REQUEST_USERINFO, ui);

            /*
             * Adds the username / password token to the SOAP header.
             */
            SecurityElementFactory factory = SecurityElementFactory.getDefaultFactory();
            Security security = factory.createSecurity(null);
            security.addToken(ui);
            session.setAttribute(WSSEClientHandler.REQUEST_SECURITY, security);

            /*
             * Get the protocol-specific proxy class.
             */
            MyWebServiceSoap msg=myservice.getMyWebServiceSoap();

            /*
             * Invoke the web service method hello(String str)
             */
            String result=msg.hello("Say Hello");

            System.out.println(result);
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

Encryption

The following client class MyWebServiceClient uses the weblogic.xml.security.wsse.Security class to encrypt and decrypt SOAP traffic. The class KeyUtil.java is also provided below.

MyWebServiceClient.java

```
import java.util.List;
import java.util.ArrayList;
import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.cert.X509Certificate;
import java.util.List;
import javax.xml.rpc.ServiceException;
```

```
import javax.xml.namespace.QName;
import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.HandlerRegistry;
import weblogic.webservice.context.WebServiceSession;
import weblogic.webservice.context.WebServiceContext;
import weblogic.webservice.core.handler.WSSEClientHandler;
import weblogic.webservice.WLMessageContext;
import weblogic.xml.security.wsse.Security;
import weblogic.xml.security.wsse.Token;
import weblogic.xml.security.wsse.BinarySecurityToken;
import weblogic.xml.security.wsse.SecurityElementFactory;
import weblogic.xml.security.specs.EncryptionSpec;
import weblogic.jws.proxies.*;

/*
 * This class encrypts/decrypts SOAP messages by using the Security object
 */
public class MyWebServiceClient {

    private static final String CLIENT_KEYSTORE = "C:/mykeystores/wlwwsse.jks";
    private static final String KEYSTORE_PASS = "password";
    private static final String KEY_ALIAS = "companya";
    private static final String SERVER_KEY_ALIAS = "companyb";
    private static final String KEY_PASSWORD = "password";

    public static void main(String[] args) {

        try{
            /*
             * The KeyUtil class is assumed to exist in the same directory as MyWebServiceClien
             * (The KeyUtil class is available at the bottom of this help topic.)
             */
            final KeyStore keystore = KeyUtil.loadKeystore(CLIENT_KEYSTORE, KEYSTORE_PASS

            /*
             * Instantiate the main proxy class. The proxy class has the same name as the
             * web service, with "_Impl" appended.
             */
            MyWebService myservice = new MyWebService_Impl("http://localhost:7001/WebServices/M

            WebServiceContext context = myservice.context();
            WebServiceSession session = context.getSession();

            /*
             * Registers a handler for the SOAP message traffic.
             */
            HandlerRegistry registry = myservice.getHandlerRegistry();
            List list = new ArrayList();
            list.add(new HandlerInfo(WSSEClientHandler.class, null, null));
            registry.setHandlerChain(new QName("hello"), list);

            PrivateKey clientprivate = KeyUtil.getPrivateKey(KEY_ALIAS, KEY_PASSWORD, keystore)
            X509Certificate clientcert = KeyUtil.getCertificate(KEY_ALIAS, keystore);
            X509Certificate servercert = KeyUtil.getCertificate(SERVER_KEY_ALIAS, keystore);

            SecurityElementFactory factory = SecurityElementFactory.getDefaultFactory();

            Token client_x509token = factory.createToken(clientcert, clientprivate);

            EncryptionSpec encSpec = EncryptionSpec.getDefaultSpec();
```

```
            Token serverToken = factory.createToken(servercert, null);

            Security security = factory.createSecurity(null);

            security.addEncryption(serverToken, encSpec);

            /*
             * Get the protocol-specific proxy class.
             */
            MyWebServiceSoap msg = myservice.getMyWebServiceSoap();

            /*
             * Add the security element to the request.
             */
            context.getSession().setAttribute(WSSEClientHandler.REQUEST_SECURITY, security);

            /*
             * Add a private key to decrypt the response
             */
            session.setAttribute(WSSEClientHandler.KEY_ATTRIBUTE, clientprivate);

            /*
             * Invoke the web service method hello(String str)
             */
            String result=msg.hello("Say Hello");

            System.out.println(result);

        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

Signatures

The following client class signs outgoing SOAP messages and verfies incoming messages using the weblogic.xml.security.wsse.Security object.

```
import java.util.List;
import java.util.ArrayList;
import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.cert.X509Certificate;
import java.security.cert.CertificateException;
import javax.xml.rpc.ServiceException;
import javax.xml.namespace.QName;
import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.HandlerRegistry;
import weblogic.webservice.context.WebServiceSession;
import weblogic.webservice.context.WebServiceContext;
import weblogic.webservice.core.handler.WSSEClientHandler;
import weblogic.webservice.WLMessageContext;
import weblogic.xml.security.wsse.Security;
import weblogic.xml.security.wsse.Token;
import weblogic.xml.security.wsse.BinarySecurityToken;
import weblogic.xml.security.wsse.SecurityElementFactory;
import weblogic.xml.security.specs.SignatureSpec;
import weblogic.jws.proxies.*;
```

```
/*
 * This class signs and verifies SOAP traffic by using the weblogic.xml.security.wsse.Security
 */
public class MyWebServiceClient {

    public static final String USERNAME="username";
    public static final String USER_PASSWORD="user_password";

    private static final String CLIENT_KEYSTORE = "C:/mykeystores/wlwwsse.jks";
    private static final String KEYSTORE_PASS = "password";
    private static final String KEY_ALIAS = "CompanyA";
    private static final String KEY_PASSWORD = "password";

    public static void main(String[] args) {

        try{

            /*
             * Instantiate the main proxy class. The proxy class has the same name as the
             * web service, with "_Impl" appended.
             */
            MyWebService myservice = new MyWebService_Impl("http://localhost:7001/WebServices/M

            WebServiceContext context = myservice.context();
            WebServiceSession session = context.getSession();

            /*
             * Registers a handler for the SOAP message traffic.
             */
            HandlerRegistry registry = myservice.getHandlerRegistry();
            List list = new ArrayList();
            list.add(new HandlerInfo(WSSEClientHandler.class, null, null));
            registry.setHandlerChain(new QName("hello"), list);

            /**
             * The KeyUtil class is assumed to exist in the same directory as MyWebServiceClien
             * (The KeyUtil class is available at the bottom of this help topic.)
             */
            final KeyStore keystore = KeyUtil.loadKeystore(CLIENT_KEYSTORE, KEYSTORE_PASS);

            X509Certificate clientcert = KeyUtil.getCertificate(KEY_ALIAS, keystore);

            PrivateKey clientprivate = KeyUtil.getPrivateKey(KEY_ALIAS, KEY_PASSWORD, keystore)

            SecurityElementFactory factory = SecurityElementFactory.getDefaultFactory();

            Token x509token = factory.createToken(clientcert, clientprivate);

            SignatureSpec sigSpec = SignatureSpec.getDefaultSpec();

            Security security = factory.createSecurity(null);

            security.addSignature(x509token, sigSpec);

            security.addToken(x509token);

            /*
             * Get the protocol-specific proxy class.
             */
            MyWebServiceSoap msg = myservice.getMyWebServiceSoap();
```

```
            /*
             * Add the security element to the request.
             */
            context.getSession().setAttribute(WSSEClientHandler.REQUEST_SECURITY, security);

            /*
             * Invoke the web service method hello(String str)
             */
            String result = msg.hello("Say Hello");

            System.out.println(result);
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

The KeyUtil class performs common operations with the Java keystore.

KeyUtil.java:

```
import java.io.IOException;
import java.io.FileInputStream;
import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.cert.X509Certificate;
import java.security.cert.CertificateException;

public class KeyUtil {

    public static KeyStore loadKeystore(String filename, String password)
        throws KeyStoreException, IOException, NoSuchAlgorithmException, CertificateException {
    final KeyStore ks = KeyStore.getInstance("JKS");
    ks.load(new FileInputStream(filename), password.toCharArray());
    return ks;
    }

    public static PrivateKey getPrivateKey(String alias, String password, KeyStore keystore)
        throws Exception {
    PrivateKey result =
        (PrivateKey) keystore.getKey(alias, password.toCharArray());

    return result;
    }

    public static X509Certificate getCertificate(String alias, KeyStore keystore)
        throws Exception {
    X509Certificate result = (X509Certificate) keystore.getCertificate(alias);
    return result;
    }

}
```

Related Topics

Web Service Security

Role−Based Security

The topics in this section describe how role−based security is used to authorize users, or to prevent users, from invoking methods on web services, page flows, Java controls, and EJBs.

Topics Included in This Section

An Overview of Role−Based Security

Provides a conceptual overview of the relationship between authorization and authentication, security roles, scoping, and role−principal mappings.

Implementing Role−Based Security

Demonstrates how to implement role−based authorization for web services, Java controls, page flows and EJBs.

Creating Principals and Role−Principal Mappings

Describes the relationship between role−principals mapping definitions and the creation of users or groups.

Related Topics

Authentication

An Overview of Role−Based Security

The topics in this section explain how role−based security can be used to restrict access to resources (web services, page flows, Java controls, EJBs) to only those users who have been granted a particular security role. It also explains the relationship between EJB−scoped, application−scoped, web−application scoped, and global security roles.

To restrict access you set up two kinds of tests that candidate users must pass to access some resources: an *authentication* process, which determines the user's identity and group membership, and an *authorization* process, which decides whether a user has the role membership necessary to access a particular resource. Once a user has access to a method and the method executes, it can run under the security role of the user or under a different security role.

The Authentication Process

A candidate user is first tested against the authentication process. The authentication process is generally a login process, where the candidate user is asked to provide a username and password. If the candidate succeeds in passing this challenge, the user is granted a set of identities: one identity is his username identity, the other identities are the set of groups that user has membership in. The user's username identity and group identities are called the user's *principals*: think of these principals as a set of credentials that the user presents when he/she wants to access some resource protected by an authorization process. For more information, see Authentication.

The Authorization Process

In the authorization process, users are tested to see if they have been granted the required role to access the protected resource. If they have been granted the required role, they can access the resource; if they haven't, they are denied access. A user has been granted a particular role if one of his/her principals has been granted a particular role. Principals are granted roles by a set of role−principal mappings.

Note. A user can be a person or another software component. For instance, a web service can invoke an EJB's method with security restrictions; if the web service does not pass the authorization process, it is prevented from invoking the EJB method.

Global Roles

Global roles are available to all resources within a server's security realm, that is, a server's domain. These roles can be used by any application and any resource using this domain. WebLogic Server predefines a set of global roles but you can define additional global roles as needed. For more information, see the WebLogic Server help topic Securing WebLogic Resources.

Scoped Roles

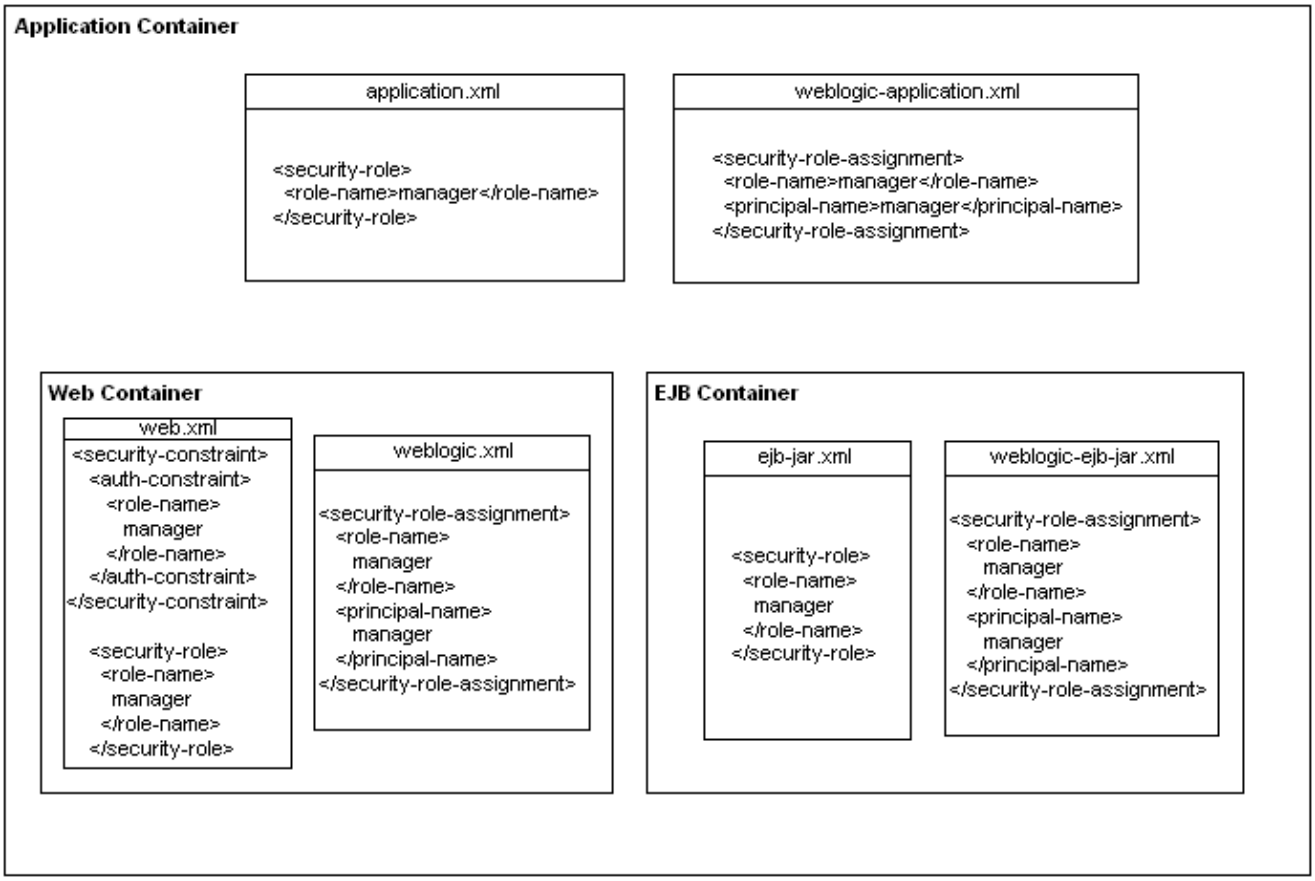
Scoped roles apply to a particular resource. WebLogic Workshop applications can have three different scopings:

1. Application scoped (defined in the application's application.xml / weblogic−application.xml files)
2. Web application scoped (defined in a project's web.xml / weblogic.xml files)
3. EJB scoped (defined in an EJB's ejb−jar.xml / weblogic−ejb−jar.xml files)

Application scoped roles can be used in an authorization process to protect any of the resources within the application, whereas web application scoped roles apply only to the resources within an individual web project and EJB scoped roles apply only to the resources within an individual EJB. For instance, if you want a security role to be defined just for a particular EJB, you make it EJB−scoped.

Note that EJB scoped roles do not exclusively protect WebLogic Workshop's EJB projects: they also can be used to protect Web Services, Java control extensions (JCX files), and JPD files. This is because all these files are *compiled into* EJBs at compile time.

The following diagram shows the three kinds of scoped roles, and corresponding deployment descriptors, that you can define with WebLogic Workshop.



Note. You can also define scoped security roles for other resources such as JDBC resources. For more information, see the WebLogic Server help topic Securing WebLogic Resources.

Role−Principal Mapping

Role−principal mappings define how principals map to security roles. A particular user can be mapped to one or more security roles or a group can be mapped to one or more security roles. Role−principal mappings for a scoped role are defined in the appropriate deployment descriptor configuration file (see the <security−role−assignment> fragments in the above picture; this is discussed in more detail in Implementing Role−Based Security).

For scoped roles, you can alternatively use the <externally−defined/> element to indicate that the role and role−principal mapping are defined elsewhere in the security realm. Specifically, when you use this element

for EJB−scoped or web application scoped roles, WebLogic Server first examines the application−scoped roles for a role with the same name and with a role−principal mapping definition. If no appropriate application−scoped roles are found, global roles are examined. For application−scoped roles with the <externally−defined/> element, global roles are examined for role−principal mappings.

Note. When you map a scoped role to a principal, the principal is assumed to exist in the security realm. Role−principal mapping does not have the side effect of defining the principal if it doesn't exist. For more inforrmation, see Creating Principals and Role−Principal Mappings.

Running Under Another Security Role

An EJB, Java control, or web service method can run under the security role of the invoking user, or it can run under a different security role and principal. This might for instance be necessary when the EJB or web service in turn use resources that have strict security requirements. For an example, see EJB Security Sample.

Related Topics

Authentication

Implementing Role−Based Security

How Do I: Create An Application−Scoped Security Role?

Implementing Role−Based Security

To restrict access to a resource (web services, page flows, Java controls, EJBs), you set up two kinds of tests that candidate users must pass to access some resources: an *authentication* process, which determines the user's identity and group membership, and an *authorization* process, which decides whether a user has the role membership necessary to access a particular resource. Once a user has access to a method and the method executes, the method can run under the security role of the user or, with the exception of page flows, under a different security role.

The current topic demonstrates how to implement role−based authorization. For an overview of these concepts, see An Overview of Role−Based Security. For more information on the authentication process, see Authentication.

Security Annotations

Within WebLogic Workshop, different security annotation are used for the different core components:

- ***@common:security***. The @common:security annotation can be used to place security role restrictions on web services (JWS files), java controls (JCX and JCS files) and business processes (JPD files). When you place this annotation on one of these components, WebLogic Workshop declares EJB−scoped roles in the corresponding deployment descriptors. The run−as attribute is used to run the web services under a different security role than the role of the invoking users.
- ***@jpf:controller and @jpf:action***. JPF files use the @jpf:controller and @jpf:action annotations to set up authorization processes. @jpf:controller is used to at the class level; @jpf:action is used at the method level. When @jpf:controller rolesAllowed is placed at the class level it *permits* the role referenced to access any method within the class. When @jpf:action rolesAllowed is placed above an individual method it *restricts* only the role reference to access the method. The @jpf:controller and @jpf:action annotations do not automatically declare corresponding roles and role mappings in the web application's deployment descriptors (web.xml and weblogic.xml). Instead these annotations assume that corresponding role declarations are present either at the web application scoping.
- ***@ejbgen:role−mapping***. To @ejbgen:role−mapping annotation is used to define EJB−scoped roles for the Enterprise JavaBeans in an EJB project. Any security role used in an EJB file must be defined in the EJB's project using this annotation. Other annotations are used to set role−based restrictions and constraints. Specifically, the run−as attribute on @ejbgen:session, @ejbgen:entity, and @ejbgen:message−driven tags specifies what security role the EJB must run as. The roles attribute on @ejbgen:local−home−method, @ejbgen:local−method, @ejbgen:remote−home−method, and @ejbgen:remote−method annotations as well as the @ejbgen:method−permission−pattern annotation define which security roles are permitted to invoke a method. The @ejbgen:security−role−ref maps a reference to a security role used in the bean code to an actual EJB−scoped security role.

More information on the various tags can be found by locating/creating a tag in source code and pressing F1 to bring up the context−sensitive help.

Authorization Implementation Examples

Web Services and Java Controls

The @common:security roles−allowed annotation is used to define the roles required to access a web service or an individual method within a web service. In the following example, all of the methods in the web service

HelloWorld can be accessed by those in the *Administrators* role, because *Administrators* are referred to in the class−level @common:security annotation.

```
/**
 * Users in the Administrators role may access any method in this web service.
 *
 * @common:security roles−allowed="Administrators"
 */
public class HelloWorld implements com.bea.jws.WebService
{
    /**
     * Only users in the Managers (or Administrators) role may access this method
     *
     * @common:operation
     * @common:security roles−allowed="Managers"
     */
    public String helloManagers()
    {
        return "Hello, Managers.";
    }

    /**
     * Only users in the Employees (or Administrators) role may access this method
     *
     * @common:operation
     * @common:security roles−allowed="Employees"
     */
    public String helloEmployees()
    {
        return "Hello, Employees.";
    }
}
```

The roles referred to in the @common:security annotation are EJB−scoped roles, applying to the EJB that is produced when your web service is compiled by WebLogic Workshop. When you place a @common:security roles−allowed annotation in a web service or Java control, each role referred to is automatically declared in the EJB's ejb−jar.xml deployment descriptor file and a role−principal mapping where the principal is given the same name as the role is automatically written to the EJB's weblogic−ejb−jar.xml deployment descriptor file. You must ensure that the principal actually exists in the security realm.

...in the ejb−jar.xml file

```
    <security−role>
        <role−name>Administrators</role−name>
    </security−role>
    <security−role>
        <role−name>Employees</role−name>
    </security−role>
    <security−role>
        <role−name>Managers</role−name>
    </security−role>

    <method−permission>
        <role−name>Administrators</role−name>
        <method>
            <ejb−name>StatelessContainer</ejb−name>
            <method−name>helloEmployees</method−name>
        </method>
```

```
        <method>
            <ejb-name>StatelessContainer</ejb-name>
            <method-name>helloManagers</method-name>
        </method>
    </method-permission>

    <method-permission>
        <role-name>Employees</role-name>
        <method>
            <ejb-name>StatelessContainer</ejb-name>
            <method-name>helloEmployees</method-name>
        </method>
    </method-permission>

    <method-permission>
        <role-name>Managers</role-name>
        <method>
            <ejb-name>StatelessContainer</ejb-name>
            <method-name>helloManagers</method-name>
        </method>
    </method-permission>
```

...in the weblogic-ejb-jar.xml file

```
    <security-role-assignment>
        <role-name>Administrators</role-name>
        <principal-name>Administrators</principal-name>
    </security-role-assignment>
    <security-role-assignment>
        <role-name>Employees</role-name>
        <principal-name>Employees</principal-name>
    </security-role-assignment>
    <security-role-assignment>
        <role-name>Managers</role-name>
        <principal-name>Managers</principal-name>
    </security-role-assignment>
```

Page Flows

The @jpf:controller roles-allowed annotation is used to define the roles required to access a page flow or an individual method within a pageflow. In the following example all methods in the the page flow securityController.jpf can be accessed by users granted the *Administrators* role. But it is not a requirement that the user be an Administrator to access any individual method. For example, Employees who aren't Administrators may access the employeeAction() method.

```
/**
 * @jpf:controller roles-allowed="Administrators"
 */
public class securityController extends PageFlowController
{

    /**
     * @jpf:action
     * @jpf:forward name="success" path="index.jsp"
     */
    protected Forward begin()
    {
```

```
        return new Forward( "success" );
    }

    /**
     * @jpf:action roles-allowed="Employees"
     * @jpf:forward name="success" path="employeesPage.jsp"
     */
    protected Forward employeeAction()
    {
        return new Forward("success");
    }

    /**
     * @jpf:action roles-allowed="Managers"
     * @jpf:forward name="success" path="managersPage.jsp"
     */
    protected Forward managerAction()
    {
        return new Forward("success");
    }
}
```

The roles referred to in the @jpf:controller and @jpf:action annotations are web application scoped roles, applying to the web application that is produced when the page flow is compiled by WebLogic Workshop. When you use the @jpf:controller and @jpf:action annotations to set up role restrictions, you must manually declare these roles and associated mappings as web application scoped roles in the project's web.xml and weblogic.xml files. Specifically, each role referred to must be declared in the web.xml deployment descriptor file and the role−principal mapping must be defined in weblogic.xml deployment descriptor file.

...in the web.xml file

```
    <security-role>
        <role-name>Administrators</role-name>
    </security-role>
    <security-role>
        <role-name>Employees</role-name>
    </security-role>
    <security-role>
        <role-name>Managers</role-name>
    </security-role>
```

Notice that the web.xml file will also contain a <security−constraint> fragment which describes authentication. In some cases authentication is done using security roles. For more information, see Authentication.

...in the weblogic−ejb−jar.xml file

```
    <security-role-assignment>
        <role-name>Administrators</role-name>
        <principal-name>weblogic</principal-name>
    </security-role-assignment>
    <security-role-assignment>
        <role-name>Employees</role-name>
        <principal-name>Employees</principal-name>
    </security-role-assignment>
    <security-role-assignment>
        <role-name>Managers</role-name>
```

```
        <externally-defined/>
    </security-role-assignment>
```

Notice in the above sample that one role is mapped to a principal with a different name, one role is mapped to a principal with the same name, and one role uses the <externally−defined/> element to refer to the role−principal mapping given for the role in the security realm. In the latter case you must ensure that this mapping is present in the security realm. In all other cases you must ensure that the principals mentioned in the role−principal mapping actually exist in the security realm.

Enterprise JavaBeans

The @ejbgen:role−mapping annotation is used to define the security roles and to do the role−principal mapping or to set the <externally−defined/> element. The following example shows the @ejbgen:local−method roles annotation to limit access to only authorized users.

```
 * @ejbgen:role-mapping principals="alfred" role-name="ceo"
 */
public class MySessionBean extends GenericSessionBean implements SessionBean
{
    public void ejbCreate() {
    }

    /**
     * @ejbgen:local-method roles="ceo"
     */
    public String authorizeProcedure(...)
    {
       ...
    }
}
```

When you place a @ejbgen:role−mapping annotation in an EJB, the role referred to is automatically declared in the EJB's ejb−jar.xml deployment descriptor file and the role−principal mapping is automatically written to the EJB's weblogic−ejb−jar.xml deployment descriptor file. You must ensure that the principal actually exists in the security realm.

...in the ejb−jar.xml file

```
    <security-role>
        <role-name>ceo</role-name>
    </security-role>
    <method-permission>
        <role-name>ceo</role-name>
        <method>
            <ejb-name>MySessionBean</ejb-name>
            <method-intf>Local</method-intf>
            <method-name>authorizeProcedure</method-name>
        </method>
    </method-permission>
```

...in the weblogic−ejb−jar.xml file

```
  <security-role-assignment>
      <role-name>ceo</role-name>
      <principal-name>alfred</principal-name>
  </security-role-assignment>
```

Related Topics

@common:security Annotation

@jpf:controller Annotation

@jpf:action Annotation

@ejbgen:role mapping Annotation

Creating Principals and Role−Principal Mappings

When you define a web service, Java control, page flow, or EJB scoped security role and the corresponding role−principal mapping, you do not actually define the principals in the security realm. Creating principals can be done in two ways: (1) through your application's ***Security Roles*** folder and (2) through the ***WebLogic Server console***.

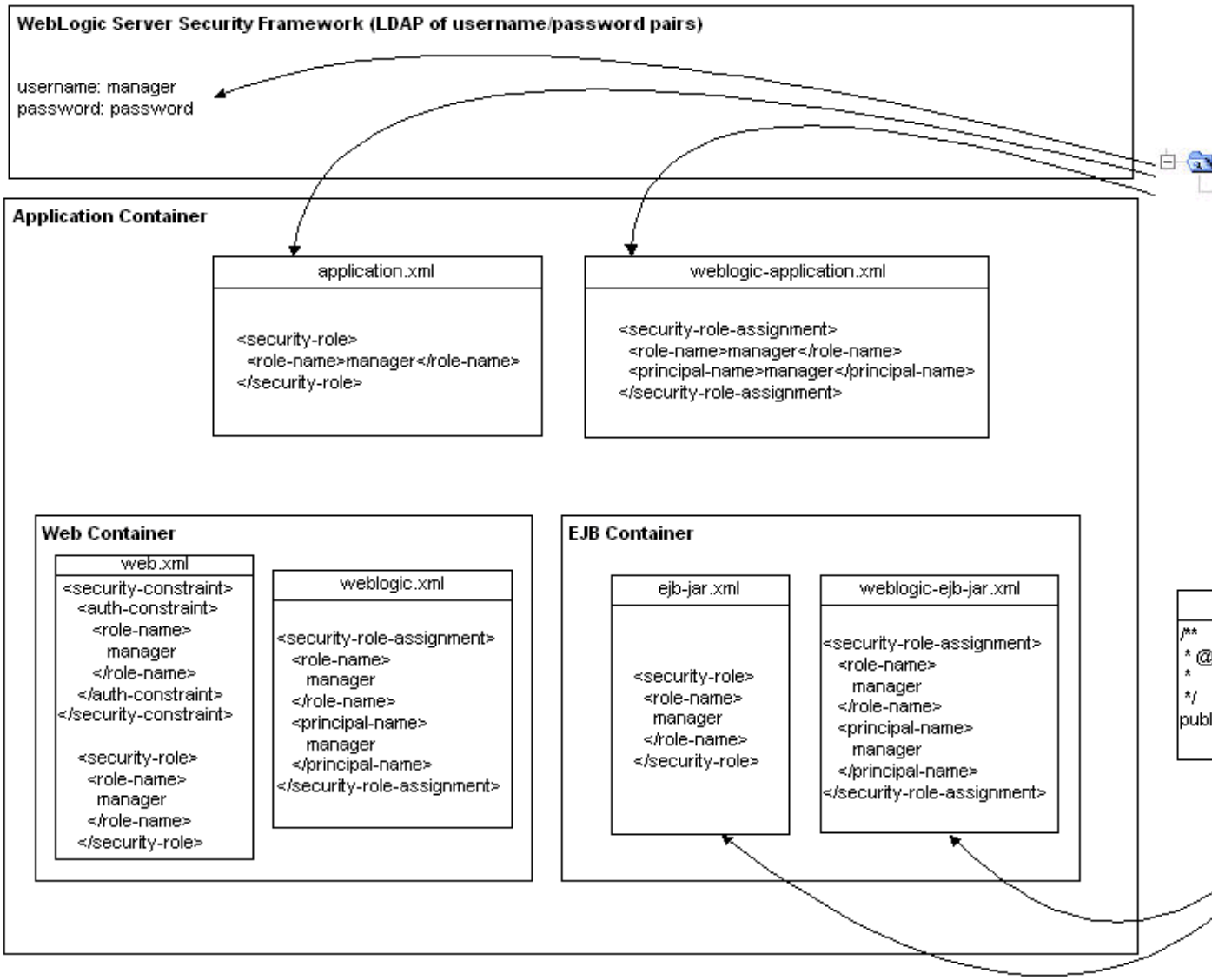
Note. Defining security roles is described in detail in Implementing Role−Based Security.

Using the Security Roles Folder

The Security Roles folder provides a convenient way to create principals in the security. The Security Roles folder appears as a project folder on the ***Application*** tab, as shown below.

To use the Security Roles folder to create a principal, right−click the folder and select ***Create a new role***, and make sure that you the ***Use role name*** option. Only when you select this option, a user with the same name as the security role is automatically created in the security realm and this user is mapped as the principal to the application−scoped role. The password for this user is password. (For more specific instructions and for more information on the other principal name options in the dialog, see How Do I: Create An Application−Scoped Security Role?) The diagram below shows how in this scenario a user is created in WebLogic Server's security framework and how application scoped roles and role−principal mappings are defined application's configuration files.

To provide a contrast, the diagram also shows how the @common:security roles−allowed annotation automatically defines EJB scoped roles and role−principal mappings in the ejb−jar.xml and weblogic−ejb−jar.xml files, but does not create the actual principals in the security realm. The same applies to the @ejbgen:role−mapping annotation for EJBs (not shown below). Note that the page flow annotations @jpf:action roles−allowed and @jpf:controller roles−allowed do not even automatically write web application scoped roles to the deployment descriptors. Instead, you must manually manage these roles in the web.xml and weblogic.xml deployment descriptor files. For more information, see Implementing Role−Based Security.



the WebLogic Server Console

You can also use the WebLogic Server console to define application−scoped or global roles, to create users and groups, and to map roles to principals. Below it is shown how to create users and how to create application−scoped roles and role−principal mapping. For a detailed description of all the WebLogic Server security functionality, see the WebLogic Server help topic Securing WebLogic Resources.

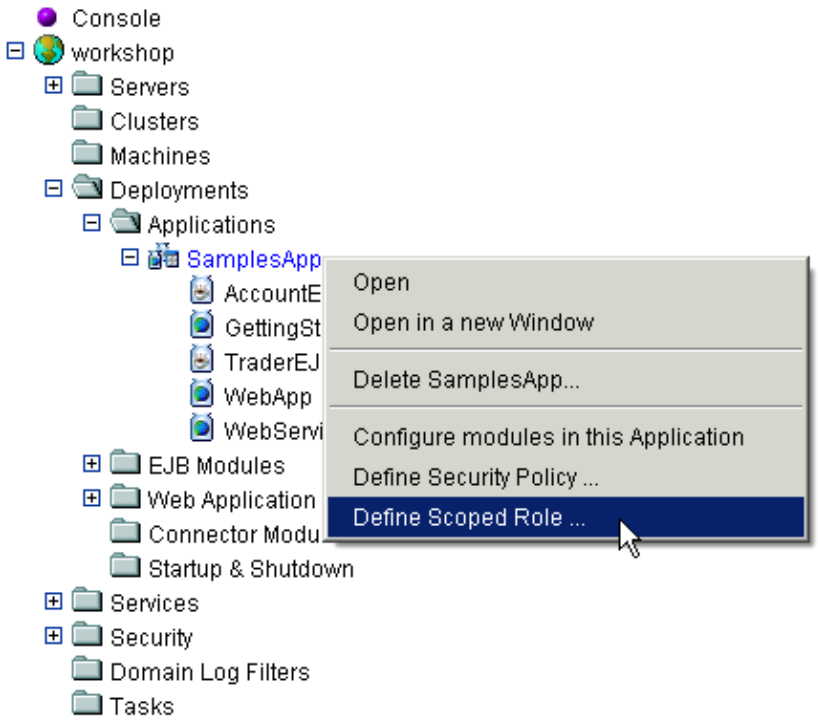
...to Create Test Users

To enter users through the WebLogic Server console, follow these steps.

1. Select ***Tools−−>WebLogic Server−−>WebLogic Console***.
2. Log on to the console. If you are using the workshop domain, use the username/password combination "weblogic/weblogic".
3. Navigate to ***Security−−>Realms−−>myrealm***.
4. By clicking on the ***Users*** folder and the ***Configure a New User*** link, you can enter a new individual user.

... to Define Application−Scoped Roles and Role Mapping

1. Select ***Tools−−>WebLogic Server−−>WebLogic Console***.
2. Log on to the console. If you are using the workshop domain, use the username/password combination "weblogic/weblogic".
3. Navigate to ***Deployments−−>Applications−−>[your application's name]*** and select ***Define Scoped Role***.



4. Click the ***Configure a new Scoped Role*** link.
5. In the ***General*** section define the security role and click the Apply button.
6. Click the ***Condition*** tag and define the role−principal mapping.

Related Topics

Role−Based Security

@common:security Annotation

@jpf:controller Annotation

@jpf:action Annotation

Authentication

Authentication establishes the identity of a user by challenging the user to provide a valid username/password pair: something only the intended user knows.

Authentication can be used to protect any web−accessible resource, including web applications, web services, page flow applications, and individual JSP pages.

If the protected web resource is intended for human clients, the application can be configured to redirect users to a login page, where they must enter a valid username and password before they can access the resource. If the web resource is intended for machine clients, the machine client can provided the required username and password through methods on the resource's control file. For detailed information on authenticating machine clients see Using Controls to Access Transport Secured Resources.

The username/password pair can be checked against a variety of authentication provider services. A default authentication provider is provided by WebLogic Server, but other providers can be supplied as needed. For details on changing the default authentication provider or adding addition providers, see Configuring Security Providers in the WebLogic Server 8.1 documentation.

If you want to restrict access to sensitive information, username/password authentication should always be used in conjunction with SSL. Without SSL the username and password to are transported over the HTTP protocol, which uses only 64−bit encryption to hide the username and password, making it relatively easy for a malicious party to intercept and decode the message. For this reason you should always use basic authentication in conjunction with SSL, which uses 128−bit encryption.

However, if the primary purpose of username/password authentication is tracking user behavior in an application, and there is no especially sensitive information at stake, you do not need to use SSL.

Below are three basic strategies for setting up a username/password challenge in a WebLogic Workshop application:

- ***Basic Authentication***: uses a standard login screen provided by the web browser.
- ***Form Authentication***: uses a custom login screen provided by the developer.
- ***Page flow authentication***: uses a page flow to authenticate a user.

Basic Authentication

Basic authentication has the advantage of being easy to implement, but, since the login page is provided by the browser software, the developer does not have control over the look and feel of the login page. For detailed information see Basic Authentication. (Also see Developing BASIC Authentication Web Applications in the WebLogic Server 8.1 documentation.)

Form Authentication

Form authentication is easy to implement and gives the developer control over the look and feel of the login screen, but it should not be used in all situations. In particular it should be used to secure (1) web services which have machine clients and (2) individual pages and methods within a page flow.

Web services with machine clients will encounter a problem interpreting the HTTP login page; instead use basic authentication for resources with machine clients.

Form authentication should not be used to secure individual pages and methods within a page flow. This is because form based authentication relies on redirecting the user from and back to the protected resource, but page flows do not support redirection from and back to the same location within a page flow. For this reason, you should only use form authentication to establish the identity of a user before he enters a page flow, not once he is within the page flow. If you want to allow a user to navigate within a page flow unauthenticated, but require authentication for other pages within the page flow use Page Flow Authentication.

For details on developing Form Authentication see Form Authentication. (Also see Developing FORM Authentication Web Applications in the WebLogic Server 8.1 documentation.)

Page Flow Authentication

Page Flow authentication uses a page flow to authenticate a user. The page flow can be a nested page flow, so it is appropriate to use this authentication technique when a user is already navigating within another page flow. For detailed information see Page Flow Authentication.

Topics Included in This Section

Basic Authentication

Form Authentication

Page Flow Authentication

Related Topics

WebLogic Server 8.1 Documentation

Developing BASIC Authentication Web Applications

Developing FORM Authentication Web Applications

Weblogic Workshop Samples

Login Samples

BasicAuthentication.jws Sample

Basic Authentication

The following topic explains how to secure a web resource using basic authentication.

Basic authentication involves a two part challenge to candidate users. (1) Basic authentication requires users to provide a valid username/password pair before they can access a web resource. (2) It also requires that users have been granted the appropriate role membership before they can access the resource. The web resource can be any web−accessible component, such as a web application, web service, or an individual JSP page.

When a user requests a web resource protected by basic authentication, first the user is redirected to a login window where he enters his username and password. If he fails to provide a valid username and password, then he is denied access to the resource. If he provides a valid username/password pair, he graduates to the second (role−based) challenge. If the user has not been granted the required role, then he is denied access to the resource (even if he provides a valid username and password). If he has been granted to required role, then he is granted access to the resource.

In basic authentication, the browser provides the login window and it cannot be customized. If you require a customizable login page use Form Authentication.

Note: you should not use basic authentication (or form authentication) to secure individual JSP pages or action methods within a page flow directory. After collecting a username and password, basic authentication redirects the user to the requested resource. But redirection into a page flow directory always fails, because the page flow's begin() action method is always invoked when a new user enters the page flow. However, it is appropriate to secure an entire page flow directory using basic or form authentication.

To secure a web resource using basic authentication, you must complete the following steps:

1. Specify the web resources to be protected
2. Specify the roles that can access the web resource
3. Specify basic authentication as the method of protection
4. Declare the security roles referenced in step 2
5. Assign security roles to principals (groups and individual users)

1. Specify the Web Resources to be Protected

You define a web resource, such as a web application or web service, as a *protected* resource by placing a security constraint on that resource. Security constraints are specified by <security−constraint> XML elements in the web.xml configuration file in the WEB−INF directory.

In the example below, the JSP page Administrators.jsp is defined as a protected web resource.

web.xml

```
<security−constraint>
    <web−resource−collection>
        <web−resource−name>Administrator's Page</web−resource−name>
        <url−pattern>/myProtectedWebResources/Administrators.jsp</url−pattern>
        <http−method>GET</http−method>
        <http−method>POST</http−method>
    </web−resource−collection>
```

```
</security-constraint>
```

The <url-pattern> Element

The child element <url-pattern> element defines which URLs in your web resources should be protected by a username/password challenge. If a user tries to access a protected URL, he is redirected to the login screen where he must enter a valid username/password pair before he can access the resource.

Using the <url-pattern> element you can restrict access to an entire web application, a folder or a particular file within the web application.

The following <url-pattern> element declares the entire project as protected.

```
<url-pattern>/*</url-pattern>
```

The following <url-pattern> element declares the /myProtectedWebResources folder as protected.

```
<url-pattern>/myProtectedWebResources*</url-pattern>
```

The following <url-pattern> element declares that the web service /myProtectedWebResources/Administrators.jsp should be protected.

```
<url-pattern>/myProtectedWebResources/Administrators.jsp</url-pattern>
```

You can place multiple <url-pattern> elements within a single security constraint.

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Administrator's Pages</web-resource-name>
        <url-pattern>/myProtectedWebResources/Administrators.jsp</url-pattern>
        <url-pattern>/myProtectedWebResources/Payroll.jsp</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
</security-constraint>
```

The <http-method> Element

The <http-method> element declares which HTTP methods (usually, GET or POST) are subject to the security constraint. If <http-method> element is omitted, the the security constraint is applied to all HTTP methods by default.

2. Specify the Roles that Can Access the Resource

It is common to use basic authentication in conjunction with role-based security. Once a user passes the username/password challenge, he can be mapped to one or more roles, allowing the developer a finer-grained control over access to web resources.

When specify basic authentication you must also set up a role-based constraint on the web resource. The authorization constraint allows access only to those user that have been granted a specific role. In the following example, the <auth-constraint> element specifies that users must be granted the Administrator role to access the resource.

web.xml

```
<security−constraint>
    <web−resource−collection>
        <web−resource−name>Administrator's Page</web−resource−name>
        <url−pattern>/myProtectedWebResources/Administrators.jsp</url−pattern>
        <http−method>GET</http−method>
        <http−method>POST</http−method>
    </web−resource−collection>
    <auth−constraint>
        <role−name>Administrator</role−name>
    </auth−constraint>
</security−constraint>
```

The <auth−constraint> Element

An <auth−constraint> child element (short for "authorization" constraint) specifies that a user must be a member of a certain role to access the web resource.

The role membership challenge can be heavy or light depending on your security needs. In some cases, requiring login is not intended to restrict access to web resources, but merely for the purpose of tracking individual users' behavior in the web application. In these cases, use a lightweight authorization challenge by requiring that users be members of the Users role. Provided that you assign the Users role to all visitors (see step 5 below), all visitors will be able to access the resource.

```
<security−constraint>
    <web−resource−collection>
        <web−resource−name>Administrator's Page</web−resource−name>
        <url−pattern>/myProtectedWebResources/Administrators.jsp</url−pattern>
        <http−method>GET</http−method>
        <http−method>POST</http−method>
    </web−resource−collection>
    <auth−constraint>
        <role−name>Users</role−name>
    </auth−constraint>
</security−constraint>
```

3. Specify Basic Authentication as the Method of Protection

To specify basic authentication at the method of protection, add a <login−config> element to the web.xml file. The <login−config> element should appear directly underneath the <security−constraint> element.

web.xml

```
<security−constraint>
    <web−resource−collection>
        <web−resource−name>Administrator's Page</web−resource−name>
        <url−pattern>/myProtectedWebResources/Administrators.jsp</url−pattern>
        <http−method>GET</http−method>
        <http−method>POST</http−method>
    </web−resource−collection>
    <auth−constraint>
        <role−name>Administrator</role−name>
    </auth−constraint>
</security−constraint>
```

```
<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>default</realm-name>
</login-config>
```

The value BASIC specifies that the browser should supply the login page shown to the user. If you with to provide a custom login page, use value FORM. For details on supplying custom login pages, see Form Authentication.

4. Declare Security Roles

The roles that you referenced in step 2 must also be declared in the web.xml file. Roles are declared using a <security−role> element.

web.xml

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Administrator's Page</web-resource-name>
        <url-pattern>/myProtectedWebResources/Administrators.jsp</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>Administrator</role-name>
    </auth-constraint>
</security-constraint>

<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>default</realm-name>
</login-config>

<security-role>
    <role-name>Administrator</role-name>
</security-role>
```

5. Assign Security Roles to Principals (Groups and Individual Users)

Finally, you must assign role membership to the groups and individuals users you want to grant access to. In the weblogic.xml file add a <security−role−assignment> element for each role to principal mapping. In the following example the group administrators_group is granted the Administrator role.

weblogic.xml

```
<security-role-assignment>
    <role-name>Administrator</role-name>
    <principal-name>administators_group</principal-name>
</security-role-assignment>
```

If you desire a lightweight authorization challenge, you can map the Users role to the users group. The users group is a pre−defined group in WebLogic Sever: everyone that successfully logs on is automatically a member of the user group.

<login−config>

As an alternative to using the <auth−constraint> element you can use the <login−config> element to require a username and password from users.

Note that if you use the <login−config> element, basic authentication is applied universally to all of the web resources protected by <security−constraint> elements within the web.xml file.

```
    <security−constraint>
        <display−name>
            Security Constraint for HelloWorldSecure.jws
        </display−name>
                <web−resource−collection>
            <web−resource−name>BasicAuthentication.jws</web−resource−name>
            <description>A web service secured by SLL and basic authentication</description>
            <!−−
            Defines the scope of the web resource to be secured with SSL.
            Secure all methods calls to the HelloWorldSecure web service.
            −−>
            <url−pattern>/security/transport/helloWorldSecure/HelloWorldSecure.jws/*</url−patte
            <http−method>GET</http−method>
            <http−method>POST</http−method>
                </web−resource−collection>
    </security−constraint>

<login−config>
                <auth−method>BASIC</auth−method>
        </login−config>
```

Related Topics

For more information about setting up an authentication procedure see Developing Secure Web Applications in the WebLogic Server 8.1 documentation.

For reference information on the web.xml file see web.xml Deployment Descriptor Elements in the WebLogic Server 8.1 documentation. See especially the documentation for the <security−constraint> element, the <login−config> element, and the <user−data−constraint> element.

WebLogic Workshop Documentation

Role−Based Security

Authorization

WebLogic Server 8.1 Documentation

Developing BASIC Authentication Web Applications

web.xml Deployment Descriptor Elements

<security−constraint>

<login−config>

<user−data−constraint>

Form Authentication

The following topic explains how to secure a web resource using form authentication.

Form authentication is similar to basic authentication: both require a candidate user to provide a valid username and password and both require that the candidate user be a member of an appropriate role before they can access a protect web resource. The major difference is that form authentication, unlike basic authentication, allows you to present a customized HTML login page to candidate users. In basic authentication, the login page is provided by the browser and cannot be customized according to your preferred look and feel.

Form authentication involves a two part challenge to candidate users. (1) It requires users to provide a valid username/password pair before they can access a web resource. (2) It also requires that users have been granted the appropriate role membership before they can access the resource. The web resource can be any web−accessible component, such as a web application, web service, or an individual JSP page.

When a user requests a web resource protected by form authentication, first the user is redirected to a login page where he enters his username and password. If he fails to provide a valid username and password, then he is denied access to the resource. If he provides a valid username/password pair, he graduates to the second (role−based) challenge. In the second challenge, WebLogic Server checks to see if the user has been granted the required role. If the user has not been granted the required role, then he is denied access to the resource (even if he provides a valid username and password). If he has been granted the required role, then he is granted access to the resource.

Note: you should not use form authentication (or basic authentication) to secure individual JSP pages or action methods within a page flow directory. After collecting a username and password, form authentication redirects the user to the requested resource. But redirection into a page flow directory always fails, because the page flow's begin() action method is always invoked when a new user enters the page flow. However, it is appropriate to secure an entire page flow directory using basic or form authentication.

To secure a web resource using form authentication, you must complete the following steps:

1. Specify the web resources to be protected
2. Specify the roles that can access the web resource
3. Specify form authentication as the method of protection
4. Declare the security roles referenced in step 2
5. Assign security roles to principals (groups and individual users)
6. Provide login and failover pages

1. Specify the Web Resources to be Protected

You define a web resource, such as a web application or web service, as a *protected* resource by placing a security constraint on that resource. Security constraints are specified by <security−constraint> XML elements in the web.xml configuration file in the WEB−INF directory.

In the example below, the JSP page Administrators.jsp is defined as a protected web resource.

web.xml

```
<security−constraint>
```

```
        <web-resource-collection>
            <web-resource-name>Administrator's Page</web-resource-name>
            <url-pattern>/myProtectedWebResources/Administrators.jsp</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
        </web-resource-collection>
    </security-constraint>
```

The <url-pattern> Element

The child element <url-pattern> element defines which URLs in your web resources should be protected by a username/password challenge. If a user tries to access a protected URL, he is redirected to the login screen where he must enter a valid username/password pair before he can access the resource.

Using the <url-pattern> element you can restrict access to an entire web application, a folder or a particular file within the web application.

The following <url-pattern> element declares the entire project as protected.

```
<url-pattern>/*</url-pattern>
```

The following <url-pattern> element declares the /myProtectedWebResources folder as protected.

```
<url-pattern>/myProtectedWebResources*</url-pattern>
```

The following <url-pattern> element declares that the web service /myProtectedWebResources/Administrators.jsp should be protected.

```
<url-pattern>/myProtectedWebResources/Administrators.jsp</url-pattern>
```

You can place multiple <url-pattern> elements within a single security constraint.

```
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Administrator's Pages</web-resource-name>
            <url-pattern>/myProtectedWebResources/Administrators.jsp</url-pattern>
            <url-pattern>/myProtectedWebResources/Payroll.jsp</url-pattern>
            <http-method>GET</http-method>
            <http-method>POST</http-method>
        </web-resource-collection>
    </security-constraint>
```

The <http-method> Element

The <http-method> element declares which HTTP methods (usually, GET or POST) are subject to the security constraint. If <http-method> element is omitted, the the security constraint is applied to all HTTP methods by default.

2. Specify the Roles that Can Access the Resource

It is common to use basic authentication in conjunction with role-based security. Once a user passes the username/password challenge, he can be mapped to one or more roles, allowing the developer a finer-grained control over access to web resources.

When specify basic authentication you must also set up a role−based constraint on the web resource. The authorization constraint allows access only to those user that have been granted a specific role. In the following example, the <auth−constraint> element specifies that users must be granted the Administrator role to access the resource.

web.xml

```
<security−constraint>
    <web−resource−collection>
        <web−resource−name>Administrator's Page</web−resource−name>
        <url−pattern>/myProtectedWebResources/Administrators.jsp</url−pattern>
        <http−method>GET</http−method>
        <http−method>POST</http−method>
    </web−resource−collection>
    <auth−constraint>
        <role−name>Administrator</role−name>
    </auth−constraint>
</security−constraint>
```

The <auth−constraint> Element

An <auth−constraint> child element (short for "authorization" constraint) specifies that a user must be a member of a certain role to access the web resource.

The role membership challenge can be heavy or light depending on your security needs. In some cases, requiring login is not intended to restrict access to web resources, but merely for the purpose of tracking individual users' behavior in the web application. In these cases, use a lightweight authorization challenge by requiring that users be members of the Users role. Provided that you assign the Users role to all visitors (see step 5 below), all visitors will be able to access the resource.

```
<security−constraint>
    <web−resource−collection>
        <web−resource−name>Administrator's Page</web−resource−name>
        <url−pattern>/myProtectedWebResources/Administrators.jsp</url−pattern>
        <http−method>GET</http−method>
        <http−method>POST</http−method>
    </web−resource−collection>
    <auth−constraint>
        <role−name>Users</role−name>
    </auth−constraint>
</security−constraint>
```

3. Specify Form Authentication as the Method of Protection

To specify basic authentication at the method of protection, add a <login−config> element to the web.xml file. The <login−config> element should appear directly underneath the <security−constraint> element.

web.xml

```
<security−constraint>
    <web−resource−collection>
        <web−resource−name>Administrator's Page</web−resource−name>
        <url−pattern>/myProtectedWebResources/Administrators.jsp</url−pattern>
        <http−method>GET</http−method>
        <http−method>POST</http−method>
```

```
    </web-resource-collection>
    <auth-constraint>
        <role-name>Administrator</role-name>
    </auth-constraint>
</security-constraint>

<login-config>
    <auth-method>FORM</auth-method>
    <realm-name>default</realm-name>
    <form-login-config>
        <form-login-page>/security/login/login.jsp</form-login-page>
        <form-error-page>/security/login/fail_login.jsp</form-error-page>
    </form-login-config>
</login-config>
```

The value FROM specifies that the developer has supplied the login page shown to the user. The path to the login page is specified in the <form−login−page> element. The login page you provide can be an HTML or JSP file. In the <form−error−page> element, you can supply a failover page, should the user fail the username/password challenge. See step 6 below for specific requirements on these pages.

4. Declare Security Roles

The roles that you referenced in step 2 must also be declared in the web.xml file. Roles are declared using a <security−role> element.

web.xml

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Administrator's Page</web-resource-name>
        <url-pattern>/myProtectedWebResources/Administrators.jsp</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>Administrator</role-name>
    </auth-constraint>
</security-constraint>

<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>default</realm-name>
</login-config>

<security-role>
    <role-name>Administrator</role-name>
</security-role>
```

5. Assign Security Roles to Principals (Groups and Individual Users)

Finally, you must assign role membership to the groups and individuals users you want to grant access to. In the weblogic.xml file add a <security−role−assignment> element for each role to principal mapping. In the following example the group administrators_group is granted the Administrator role.

weblogic.xml

```
<security-role-assignment>
    <role-name>Administrator</role-name>
    <principal-name>administators_group</principal-name>
</security-role-assignment>
```

If you desire a lightweight authorization challenge, you can map the Users role to the users group. The users group is a pre-defined group in WebLogic Sever: everyone that successfully logs on is automatically a member of the user group.

<login-config>

As an alternative to using the <auth-constraint> element you can use the <login-config> element to require a username and password from users.

Note that if you use the <login-config> element, basic authentication is applied universally to all of the web resources protected by <security-constraint> elements within the web.xml file.

```
<security-constraint>
    <display-name>
        Security Constraint for HelloWorldSecure.jws
    </display-name>
            <web-resource-collection>
        <web-resource-name>BasicAuthentication.jws</web-resource-name>
        <description>A web service secured by SLL and basic authentication</description>
        <!--
        Defines the scope of the web resource to be secured with SSL.
        Secure all methods calls to the HelloWorldSecure web service.
        -->
        <url-pattern>/security/transport/helloWorldSecure/HelloWorldSecure.jws/*</url-patte
        <http-method>GET</http-method>
        <http-method>POST</http-method>
            </web-resource-collection>
</security-constraint>

<login-config>
            <auth-method>BASIC</auth-method>
    </login-config>
```

6. Provide Login and Failover Pages

The custom login page must meet three requirements:

1. it must contain an HTML <form> with the attribute action="j_security_check"
2. it must contain an HTML <input> with the attribute name="j_username"
3. it must contain an HTML <input> with the attribute action="j_password"

You can use the following HTML <form> as a template for your login page.

```
<form method="POST" action="j_security_check">
    Username: <input type="text" name="j_username"><br>
    Password: <input type="password" name="j_password"><br>
    <input type=submit value="Submit">
</form>
```

The failover page does not have any special requirements. The failover page is shown to users who fail the username/password challenge. If the user passes the username/password challenge, but fails the role membership challenge, the failover page will not be shown. Error handling in this case must be provided by your web application.

Related Topics

For more information about setting up an authentication procedure see Developing Secure Web Applications in the WebLogic Server 8.1 documentation.

For reference information on the web.xml file see web.xml Deployment Descriptor Elements in the WebLogic Server 8.1 documentation. See especially the documentation for the <security−constraint> element, the <login−config> element, and the <user−data−constraint> element.

WebLogic Workshop Documentation

Role−Based Security

Authorization

WebLogic Server 8.1 Documentation

Developing BASIC Authentication Web Applications

web.xml Deployment Descriptor Elements

<security−constraint>

<login−config>

<user−data−constraint>

Page Flow Authentication

The following topic explains how to login in users using a nested page flow and then return the logged in user to the original Action in the nesting page flow.

You should *not* use a <security−constraint> in the WEB−INF/web.xml file to implement this form of security. Instead, use the @jpf:action login−required annotation to protect individual Action methods within the page flow file.

Requiring Login

You can protect an individual Action method within a page flow from non−logged in users with the @jpf:action login−required="true" annotation. If a non−logged in user tries to access this method, a NotLoggedInException will be thrown. To catch this exception and send the user to another (nested) page flow where they can be logged in, use the @jpf:catch annotation.

On the nesting (parent) page flow...

```
/**
 * If a non−logged in user calls this Action, a NotLoggedInException is thrown
 * (this is because login−required is set to "true").  The exception is caught
 * and the user is sent to the login page flow (a nested page flow).
 *
 * If the user successfully logs in, he is returned to this Action, which executes normally
 *
 * @jpf:action login−required="true"
 * @jpf:forward name="success" path="saved.jsp"
 * @jpf:catch type="com.bea.wlw.netui.pageflow.NotLoggedInException" path="/loginPageFlow/l
 */
protected Forward save()
{
    //
    // The code here can be executed only by logged in users.
    //

    return new Forward( "success" );
}
```

Logging In Users

Within the login page flow, you can use an standard <netui:form>, databound to a Form Bean, to collect the users username and password.

```
    <netui:form action="executeLogin">
        <table>
            <tr>
                <td>
                    Username:
                </td>
                <td>
                    <netui:textBox dataSource="{actionForm.username}"/>
                </td>
            </tr>
            <tr>
```

```
                <td>
                    Password:
                </td>
                <td>
                    <!-- The password attribute hides the user input in the browser -->
                    <netui:textBox dataSource="{actionForm.password}" password="true"/>
                </td>
            </tr>
        </table>
        <netui:button value="Submit"/>
    </netui:form>
```

To handle the user submission of a username and password, use the method login(String username, String password), a method on the com.bea.wlw.netui.pageflow.FlowController class. The login(String username, String password) method will consult WebLogic Server's authentication provider(s) to attempt to login the user.

If the login attempt fails, a javax.security.auth.login.LoginException is thrown. In the sample below, this exception is caught and the user is returned to the login page to try again.

If the login attempt succeeds, the user can be returned to the original Action method in the parent page flow using the @jpf:forward return−action attribute. The return−action attribute always refers to an Action in a parent page flow, not in the current, nested, page flow.

...on the nested (child) page flow

```
/**
 * @jpf:action
 * @jpf:catch type="javax.security.auth.login.LoginException" path="login.jsp"
 * @jpf:forward name="success" return−action="loginSuccess"
 */
protected Forward executeLogin( LoginForm form )
    throws LoginException
{
    /*
     * The login method below is a method of the com.bea.wlw.netui.pageflow.PageFlow
     * class.  It logs in the user against WebLogic Server's current authentication provide
     *
     * If the login succeeds (if user is known to the authentication provider), then
     * the user is returned to the originating Action method on
     * the nesting page flow.
     * Note that the user is returned to the save Action via the "loginSuccess" method on t
     * nesting page flow.
     *
     * If the login fails, a LoginException is returned and the user is returned to the
     * login page.
     */
    login( form.getUsername(), form.getPassword() );
    return new Forward( "success" );
}
```

Once the user has been logged in via the nested page flow, he is returned to a dummy method on the nesting page flow; this dummy method is used to return the user to the original method requiring login.

...on the nesting (parent) page flow

```
/**
```

```
     * @jpf:action
     * @jpf:forward name="prevAction" return-to="previousAction"
     */
    protected Forward loginSuccess()
    {
        return new Forward( "prevAction" );
    }
```

Related Topics

Login Samples

@jpf:action Annotation

@jpf:catch Annotation

Securing Portal Applications

This topic provides a basic overview of WebLogic Portal security. The other portal topics, listed in Topics Included in this Section, provide implementation instructions.

WebLogic Portal uses the underlying WebLogic Server security architecture to let you create secure portal applications. The ultimate goal of portal security is to restrict access to portal resources and administrative functions to only those users who need access to those resources and functions.

Topics Included in This Section

Overview

Provides overview information on authentication, user and group management, authorization, and WebLogic Portal security management tools and resources.

Portal Security Scenario

Provides a scenario describing a fictitious company's portal application.

Implementing the Portal Security Scenario

Uses the portal scenario to identify the security touch points and link to implementation information.

Implementing Authentication

Provides details on the authentication examples contained in the Tutorial Portal.

Using Multiple Authentication Providers in Portal Development

Describes how to develop applications with users stored in external authentication providers.

How WebLogic Portal Uses the WebLogic Server Security Framework

Discusses WebLogic portal support and limitations for working with multiple authentication providers.

Overview

Note: Implementing security in a portal requires a basic understanding of standard security concepts, many of which are outside the scope of WebLogic documentation; for example, encryption, injection of SQL statements at login, and secure socket layers (SSL). The Related Topics section contains, among other things, links to information that will help give you a broader, more complete view of security and the issues surrounding it.

WebLogic Portal provides built−in functionality for authentication ("Who are you?") and authorization ("What can you access?").

Authentication

WebLogic Portal provides many authentication samples that you can incorporate into your portals. WebLogic Portal also provides many tools for user/group management.

Samples

Implementing Authentication contains details about the authentication examples contained in the Tutorial Portal.

WebLogic Portal also provides two sample login portlets you can reuse in your portals to authenticate WebLogic users:

- Login to Portal portlet − Provides basic login functionality.
- Login Director portlet − Provides login and shows the user the first desktop to which he is entitled.

You can also build other types of authentication supported by WebLogic Server.

User/Group Management

The WebLogic Administration Portal provides tools for managing users, groups, and setting user/group properties. For information on managing users and groups, see:

- Creating User Proflie Properties
- User/Group Management JSP Tags
- Using Portal Controls
- WebLogic Portal Javadoc
- The WebLogic Administration Portal help system

Authorization

There are three fundamental categories of things that can be secured in portals:

- The WebLogic Administration Portal
- Portal Resources
- Java 2 Enterprise Edition (J2EE) Resources

Using the WebLogic Server concept of security roles, WebLogic Portal lets you dynamically match users to roles at login. Different roles are, in turn, assigned to different portal resources, administrative tools, and J2EE resources so users can access only the resources and tools that their assigned roles allow.

WebLogic Administration Portal

The WebLogic Administration Portal provides the tools for managing users, portal delegated administration roles, visitor entitlement roles, interaction management rules, content management, and portal resources.

You can lock down the WebLogic administration portal with ***delegated administration***, which provides secure administrative access to the WebLogic Administration Portal tools. Delegated administration security is based on the delegated administration roles you create.

Portal Resources

The WebLogic Workshop Portal Extensions and the WebLogic Administration Portal provide tools for creating and managing portals, desktops, shells, books, pages, layouts, look & feels, and portlets. You can control access to portal resources for two types of users: administrators and visitors.

Administrators − You can control the portal resources that can be managed by portal administrators using ***delegated administration***.

Visitors − You can control visitor access to portals and portal resources with ***visitor entitlements***. Visitor entitlements are based on the visitor entitlement roles you create.

J2EE Resources

J2EE resources are the application framework and logic (Web applications, JSPs, EJBs, and so on) for which you can control visitor access. Security on J2EE resources is based on global security roles set up in WebLogic Server and applied to the individual J2EE resources. Security roles for J2EE resources are different than security roles that users can belong to, though both types of roles use the same roles architecture.

Default Users

The portal sample domain <BEA_HOME>\<WEBLOGIC_HOME>\samples\domains\portal and any portal domain you create with the Configuration Wizard include the following default users. You can add these usernames and passwords to your existing domains.

Username	***Password***	***Belongs to these groups***
weblogic ***Note***: This is the username for the portal sample domain. In a new portal domain created with the Configuration Wizard, this can be whatever was used for the primary system administrator.	weblogic ***Note***: This is the password for the portal sample domain. In a new portal domain created with the Configuration Wizard, this can be whatever was used for the primary system administrator.	Administrators PortalSystemAdministrators
portaladmin This is a default user for managing and setting up delegated administration on portals. If	portaladmin	Administrators PortalSystemAdministrators

your domain does not contain portals, you can safely delete this user.		

Related Topics

WebLogic Server Security

The Open Web Application Security Project (OWASP)

Setting up Unified User Profiles

Creating User Profile Properties

Using Portal Controls (for user/group management)

User/Group Management JSP Tags

For details on managing users and groups, see the WebLogic Administration portal online help system, also available on e−docs.

Portal Security Scenario

The following scenario describes the portal implementation needs of a fictitious company called Avitek, many of which involve security considerations. The topic that follows, Implementing the Portal Security Scenario, describes the security touch points in the scenario and provides links to implementation information.

Avitek needs two types of portal−based Web presence: an internal site for its employees and partners called "Inweb," and a public portal for its customers called "Outweb." It needs authentication for both sites. Inweb must live behind a firewall.

Outweb is set up on a server cluster for load balancing and failover.

For Inweb, Avitek needs to cater to three different types of users: managers, regular employees, and partners.

For the three types of users, Avitek wants to create only two portals: one for managers and employees and one for partners. Since there are five different partners, each partner must have a separate view of Inweb.

Some of the partners also perform contract work for Avitek, so they must also be able to access the employee portal desktop.

Avitek wants all Inweb users to authenticate before seeing any view of the portals.

For Outweb, Avitek provides information and services on a subscription basis, so it wants to provide a portal that lets all users see unsecured company information and log in to see secure information.

Avitek has a staff of two to administer all portals, and it wants to grant limited administrative access to certain partners to let them maintain their partner portal.

There are two JSP−based administration portlets that can never be seen by anyone other than Avitek's in−house administrators.

Avitek also wants to use its existing content management system for delivering content to its portals. The content management system vendor has created an interface to connect to BEA's Virtual Content Repository.

Avitek will use two user databases: The Intranet site will use an existing user database, and the public site will use the default WebLogic Server LDAP user database and is gradually adding users to it.

Related Topics

Implementing the Portal Security Scenario

Securing Portal Applications

Implementing the Portal Security Scenario

This topic walks you through the Portal Security Scenario, highlights the security touch points in the scenario, and points to the tools and information for implementing security.

Scenario	***Security Touch Points***
Avitek needs two types of portal−based Web presence: an internal site for its employees and partners called "Inweb," and a public portal for its customers called "Outweb." It needs authentication for both sites. Inweb must live behind a firewall. Outweb is set up on a server cluster for load balancing and failover.	Because one site must reside behind a firewall and the other outside the firewall, two servers are needed. • See Using WebLogic Server Clusters in the WebLogic Server documentation, especially the section on Security Options for Cluster Architectures.
For Inweb, Avitek needs to cater to three different types of users: managers, regular employees, and partners.	The three different audiences can be identified by their user profile settings. For example, you can create a user profile property called "user_type" that has three possible values: manager, employee, and partner. Each user is assigned an appropriate value. While user profile properties are not a direct security feature, you can use the properties to define the roles that determine secure access to resources. • See Creating User Profile Properties in this help system. • For information on creating visitor entitlement roles and mapping those roles to portal resources to secure the resources, see the WebLogic Administration help system. Users will use VPN to gain access through the firewall.

For the three types of users, Avitek wants to create only two portals: one for managers and employees and one for partners. Since there are five different partners, each partner must have a separate view of Inweb.	This requirement is easily solved by the flexibility of the WebLogic Portal architecture. When you create a portal file with the WebLogic Workshop Portal Extensions, you can add books, pages, and portlets to that file. The portal file serves as a template with which portal administrators can create multiple portal instances, or desktops. Each desktop can contain any combination of books, pages, and portlets provided by the template, and those portal resource instances can be assigned their own visitor entitlement and delegated administration roles for security. To meet this requirement, then, only one portal is required. That portal can be used as a template to create separate desktops for managers, employees, and for each of the five partners. When any user logs in, the system knows which type of user they are (based on user profile properties), and they are shown only the desktop(s) they can access (based on visitor entitlement and delegated administration roles that are tied to the user profile properties). There are two other options for meeting this requirement: creating two portal files within a single Web application (project), or creating two projects with a portal in each. A decision to create two portal files is simply an organizational decision. Both portal files serve as templates for portal administrators to construct portal desktops, and the same mechanisms for applying visitor entitlements and delegated administration apply. If Avitek created separate projects for the portals (Web applications), they could secure the J2EE resources (such as JSPs) in each separately, since Web applications have separate security scopes in WebLogic Server. However, since you can secure individual resources in a single Web application, you can secure J2EE resources for different audiences in that application using WebLogic Server security. Using separate Web applications for each portal means Avitek might have to implement single

	sign−on as a convenience for partners who can access the employee portal and employees who can access the partner portal. Users will be able to see all desktops they are entitled to in both Web applications. • See Assembling Portal Applications in this help system for instructions on creating portals. • See Securing WebLogic Resources in the WebLogic Server documentation, especially the sections dealing with securing URL (Web) resources such as JSPs and Enterprise Java Beans (EJBs). • The Portal Samples contain an example implementation of single sign−on.
Some of the partners also perform contract work for Avitek, so they must also be able to access the employee portal desktop.	Because you need to identify someone by certain characteristics, you could again use a user profile property to identify a user as both a partner and an employee. Instead of making the property a "choose one value" type (single, restricted), you could make the property a "choose multiple values" type (multiple, restricted). You can handle security access to portal desktops and other resources with visitor entitlements. If the employee and partner portals are located in separate Web applications, you can provide single sign−on for partners as a convenience, then handle security access to portal desktops and other resources with visitor entitlements. • See Creating User Profile Properties in this help system. • For information on creating visitor entitlement roles and mapping those roles to portal resources to secure the resources, see the WebLogic Administration help system. • The Portal Samples, especially the Tutorial Portal, contain example implementations of authentication, including single sign−on.
Avitek wants all Inweb users to authenticate before seeing any view of the portals.	You can set up the Intranet portal to use a front−end login JSP. After successful login, users are taken to the portal desktop to which they have access. • Designation of a login JSP occurs in the WebLogic Administration Portal. When

	creating a portal, enter the name of the login JSP file in the optional ***Portal URI*** field of the portal properties window. For information on creating portals, see the WebLogic Administration Portal help system. • You can also secure J2EE resources, including specific URL patterns, using your application's deployment descriptors. See Securing WebLogic Resources in the WebLogic Server documentation, especially the sections dealing with securing URL (Web) resources. • The Portal Samples, especially the Tutorial Portal, contain example implementations of authentication.
For Outweb, Avitek provides information and services on a subscription basis, so it wants to provide a portal that lets all users see unsecured company information and log in to see secure information.	Unlike the previous requirement in the scenario where a separate login JSP was required to access the portal, this requirement lets users access a portal without authenticating. The only time they must authenticate is when they want to view the protected information. Providing authentication based on whether or not users are subscribers is another instance where user properties are useful. For example, you could create a "subscriber" property and set it to "true" or "false." You could create a "subscriber" role that allows only subscribers view protected information at login. A best practices way of providing secure information is by putting secure portlets on a dedicated page. The page itself could even be secured (entitled). • See Creating User Profile Properties in this help system. • The Portal Samples contain a Login to Portal Portlet you can reuse in your portals. The samples also contain example implementations of authentication. • For information on creating pages and adding portlets to them, see Assembling Portal Applications in this help system and the Portal Management topics in the WebLogic Administration Portal help system.

Avitek has a staff of two to administer all portals, and it wants to grant limited administrative access to certain partners to let them maintain their partner portal.	Delegated administration for WebLogic Portal is set up in the WebLogic Administration Portal. A system administrator or super portal administrator can set up other administrators and delegate different levels of access to them. Delegated administration for tools and portal resources in the WebLogic Administration Portal can be defined by roles, users, and groups. In this part of the scenario you could create an administrator role based on user properties and define delegated administration accordingly. Or you could create two groups: "local administrators" and "partner administrators," add users to those groups, and set up delegated administration with those groups. With delegated administration roles set up, you can apply those roles to individual portal resources, giving the staff administrators full access and the partner administrators access to only their portal resources. • For information on creating users and groups and setting up delegated administration, see the WebLogic Administration Portal help system. • See Creating User Profile Properties in this help system.
There are two JSP−based administration portlets that can never be seen by anyone other than Avitek's in−house administrators.	Securing portlets is simple, straightforward, and powerful using visitor entitlements. In this requirement of the scenario, there may be a need for backup security assurance. This can be accomplished by securing the JSPs in question with WebLogic Server security policies. Security policies are server−level global roles that are applied to J2EE resources. • See Securing WebLogic Resources in the WebLogic Server documentation, especially the sections dealing with securing URL (Web) resources such as JSPs and Enterprise Java Beans (EJBs).
Avitek also wants to use its existing content management system for delivering content to its portals. The content management system vendor has created an interface to connect to BEA's Virtual	By adding compatible third−party content management systems into the Virtual Content Repository, content security in those third−party systems is maintained in the Virtual Content Repository.

Content Repository.	WebLogic Portal's Virtual Content Repository also provides limited content security beyond the security provided by compatible third−party content management systems. • See To control user access in the My Content Portlet topic. • See the Content Management topics in the WebLogic Administration Portal help system.
Avitek will use two user databases: The Intranet site will use an existing user database, and the public site will use the default WebLogic Server LDAP user database and is gradually adding users to it.	Inweb − Uses existing RDBMSRealm in the existing domain. This can remain the authentication provider/user database or Avitek can migrate the user database to the WebLogic Server LDAP user database. Outweb − Uses WebLogic Server's default LDAP user database. WebLogic Server's default LDAP user database provides all the power and functionality of the WebLogic Server security architecture. • See Managing WebLogic Security in the WebLogic Server documentation.

Summary

Following is a summary of the configuration Avitek will use for its Inweb and Outweb portal sites:

Inweb	***Outweb***
Inweb can experience minor down time, so it can be set up on a single server. Because there are many existing internal users Avitek wants to retain, it will use its existing RDBMSRealm for user storage and authentication. Inweb is set up behind a firewall, so users will use VPN to gain access from outside the firewall. Because of the flexibility of the portal framework, only one portal is needed. Multiple desktops can be created and entitled based on that single portal. Avitek will use BEA's Virtual Content	Outweb cannot experience down time, so it must be set up on a cluster. It will use the default WebLogic LDAP user database and authentication provider. Because of the flexibility of the portal framework, only one portal is needed. Multiple desktops can be created and entitled based on that single portal. Avitek will use BEA's Virtual Content Repository to connect to its BEA−compatible third−party content management system.

Repository to connect to its BEA−compatible third−party content management system.	

Samples

Portal Samples

Login to Portal Portlet

Related Topics

Securing Portal Applications

Portal Security Scenario

Implementing Authentication

WebLogic Portal provides many different ways to implement user login and authentication against any available authentication providers. The following sections provide nine authentication samples to help you better understand the choices you have in implementing authentication. These sections are taken directly from the samples in the Tutorial Portal.

Note: This topic describes how to implement authentication after authentication providers have been configured for use with WebLogic Server. For information on setting up authentication providers, see Using Multiple Authentication Providers In Portal Development.

In these samples, the portal Web project root is <WEBLOGIC_HOME>/samples/portal/portalApp/tutorial. Paths in the samples are relative to this root. All resources and configuration for these samples is included in the tutorial portal Web project which uses the <WEBLOGIC_HOME>/samples/domains/portal/config.xml server (called portalServer). To use these samples in your own domains and portal Web projects, import or mimic the files and configurations used in the samples.

This topic includes the following samples:

1. Form Based
2. Client Certificate
3. Backing File
4. Multi−Page User Registration Using Page Flow
5. Single Sign−on Within WebLogic with a Second Application
6. Auto Login
7. Basic Authentication
8. Portal Access that Requires Login
9. Perimeter Login
10. User Login Control

1. Form Based

Source Location: /portlets/login/formLogin/

The example /WEB−INF/web.xml specifies a CONFIDENTIAL transport−guarantee so the /portlets/login/formLogin/login_link.jsp must build a HTTPS URL to access the redirect.jsp. The redirect.jsp simply redirects back to the portal. This example leaves the protocol in HTTPS, but you could switch back to HTTP in redirect.jsp if you only wanted HTTPS to protect your username/password during login.

Note: The <form−login−page> URLs specified in web.xml will be different based on whether you are running the portal from a .portal file in WebLogic Workshop or assembled from the database when you create a desktop in the WebLogic Administration Portal. For example, when running a file based portal such as sample.portal (in development) the <form−login−page> element might be specified with /samplel.portal?_nfpb=true&_pageLabel=login. When running an assembled portal (in production) the <form−login−page> element might be specified with /appmanager/samplePortal/sampleDesktop?_nfpb=true&_pageLabel=login.

2. Client Certificate

Source Location: /portlets/login/clientCert/

These are the steps for using client certificate authorization:

1. Comment out the FORM or BASIC <login−config> in /WEB−INF/web.xml and uncomment the CLIENT−CERT <login−config>. This is necessary because the Web application can have only a single login−config.
2. Next you can do one of 2 things:
   - Import the democlient−cert.p12 client certificate into your browser, located in /portlets/login/clientCert/.

     or
   - Generate your own certificate using openssl.

   Note: democlient−cert.p12 was created for demonstration purposes and is not meant for production use. If you choose to generate your own certificate using openssl, they have instructions at their Web site at www.openssl.org.
3. If you import democlient−cert.p12, the following is for importing into IE version 6:
   a. Double−click the democlient−cert.p12 file.
   b. Click ***Next*** when the Certificate Import Wizard appears.
   c. democlient−cert.p12 should be displayed in the filename textbox. Click ***Next***.
   d. Do not type a password for the private key. Click ***Next***.
   e. You can select a certificate store or not. Click ***Next***.
   f. Click ***Finish***.

The following steps are for configuring WebLogic Server to use SSL and the democlient−cert correctly.

4. With a running portalServer, open the WebLogic Administration Console (http://<server>:<port>/console).
5. Using the tree view pane, navigate to ***Security*** > ***Realms*** > ***realmName*** > ***Providers*** > ***Authentication*** > ***DefaultIdentityAsserter***.
6. In the User Name Mapper Class Name textbox, enter examples.login.ExampleUserNameMapper.
7. Move the X.509 certificate type to the Chosen box and click ***Apply***.
8. ***Navigate to Security*** > ***Realms*** > ***realmName*** > ***Users*** and create a new user called support with password of password.
9. Navigate to ***Servers*** > ***portalServer*** and click the ***Keystores and SSL*** tab.
10. Click the ***Show*** link for the Advanced Options at the bottom of the page.
11. Select ***Client Certs Requested But Not Enforced*** from the Two Way Client Cert Behavior drop down (or enforced depending on the desired behavior) and click ***Apply***.
12. Add the examples.login.ExampleUserNameMapper.class to the system classpath. This can be accomplished by adding the class to netuix_system.jar. The ExampleUserNameMapper extracts the username from the e−mail of the Subject DN in the X.509 certificate. For example, the democlient−cert.p12 has a Subject DN with an e−mail of support@bea.com and the resulting username is "support". This is why the support user was added to the realm in the previous steps.

Because the form−based login example uses SSL, the one−way SSL was already configured for the server. If you need to enable client−certificate authentication for any server, it is a prerequisite to configure one−way SSL (see "Configuring SSL" in "Managing WebLogic Security" at http://e−docs.bea.com/wls/docs81/secmanage/ssl.html).

After you take these steps, you can access the portal and /portlets/login/formLogin/login_link.jsp to use your client certificate to get logged in to the portal Web application. See the form−based login example for an explanation of the use of the login link to access a protected resource.

3. Backing file

Source Location: /portlets/login/backingFileLogin/

This example uses portal personalization code and a backing file to log in (/WEB−INF/src/portlet/login/LoginBacking.java). The backing file also redirects back to the portal so that database state is not clobbered by control state.

4. Multi−Page User Registration Using Page Flow

Source Location: /portlets/login/pageflowLogin/

This example uses Java Page Flow to show how a multi−page user registration portlet can be accomplished. This example has 4 pages:

1. The first displays a simple user registration page.
2. The second page gathers more user information that could be stored in user properties (personalization code).
3. The third page will optionally authenticate the user or show a summary page (2 links).
4. The fourth page is either the logged in status of the user or the summary page.

5. Single Sign−on Within WebLogic with a Second Application

Source Location: /portlets/login/ssoLogin/

This is an example of single sign−on between two Web applications. For single sign−on to work, both Web applications must have matching cookie name entries in web.xml. Since by default WebLogic sets the cookie names to be identical if unspecified for a Web application, this behavior should work by default.

6. Auto Login

Source Location: /portlets/login/autologin/

Note: This example uses cookies, which is an insecure authentication method.

This example shows how to use cookies and encoding for autologin. When you login and select the "autologin" checkbox, it will encode your username and password. The username and password will be added as a cookie to the response for a life of 1 day. After this point, if you leave the portal and come back, you will be logged in automatically, including when you exit the browser. If you log out, the cookies will be deleted and you will no longer be automatically logged in when you revisit the portal.

This example uses a backing file (/WEB−INF/src/portlet/login/AutoLoginBacking.java).

7. Basic Authentication

Source Location: /portlets/login/basicLogin/

This example uses the same principles as the form−based login. To use basic authentication, simply uncomment the FORM or CLIENT−CERT based authentication methods in web.xml, and use the basic method of authentication. You can use one of the default users in the realm such as visitor1/password to log in.

8. Portal Access that Requires Login

Source Location: /portlets/login/loginRequiredPortal/

This example shows a portal that is only accessible after user authentication. To enable this, simply add a security constraint entry in web.xml for all URL resources. For Example:

```
<security−constraint>
    <web−resource−collection>
        <web−resource−name>login</web−resource−name>
        <description>Security constraint for the whole portal</description>
        <url−pattern>/*</url−pattern>
    </web−resource−collection>
    <auth−constraint>
        <description>all users</description>
        <role−name>AnonymousRole</role−name>
    </auth−constraint>
</security−constraint>
```

9. Perimeter Login

Source Location: /portlets/login/perimeterLogin/

See the following WebLogic Server documentation topics:

- "The Authentication Process" in "WebLogic Security Service Architecture" at http://e−docs.bea.com/wls/docs81/secintro/archtect.html.
- "Configuring Security Providers" at http://edocs/wls/docs81/secmanage/providers.html, especially the following sections:
    - ♦ Configuring a WebLogic Authentication Provider
    - ♦ Configuring a WebLogic Identity Assertion Provider

Related Topics

Tutorial Sample

How WebLogic Portal Uses the WebLogic Server Security Framework

How WebLogic Portal Uses the WebLogic Server Security Framework

When you build portal applications, you secure them using authentication (Who are you?) and authorization (What can you access?).

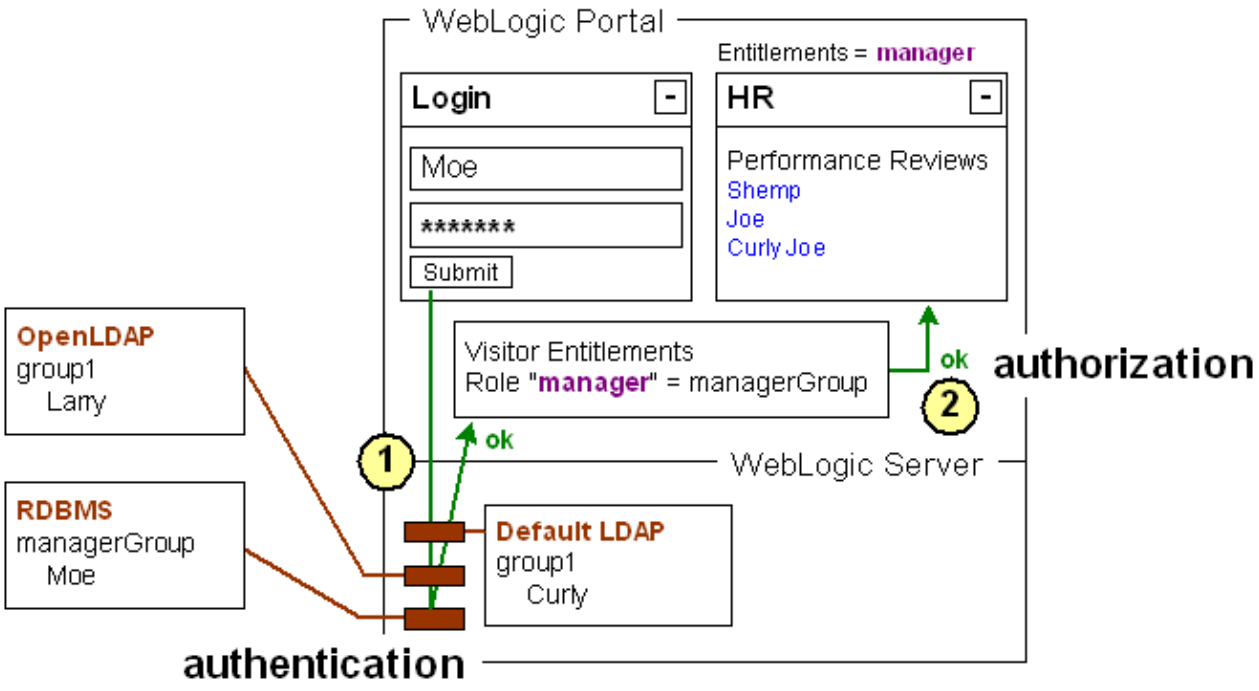
Authentication − WebLogic Portal uses WebLogic Server for login authentication, whether you are using only WebLogic Server's default LDAP user store, one or more external authentication providers configured for use with WebLogic Server, or both. The WebLogic Portal user and group management framework communicates with multiple authentication providers through WebLogic Server for basic user and group operations.

See: Using Multiple Authentication Providers in Portal Development | Implementing Authentication

Authorization − WebLogic Portal has its own authorization framework that lets you define roles for who can access which portal resources. You can define delegated administration roles for your portal administrators, and you can define entitlement roles for visitors to your portals. When a portal administrator logs in to the WebLogic Administration portal, the administrator sees only the areas he can administer. When a visitor logs in to a portal, the visitor sees only the books, pages, and portlets to which he is entitled.

See: Delegated Administration Overview | Overview of Visitor Entitlements

The following illustration shows authentication and authorization in more detail.



In this example, three authentication providers are being used by WebLogic Server: an OpenLDAP provider, an RDBMS (relational database) provider, and WebLogic Server's default LDAP provider. The two external authentication provider servers are running, and they have been added to WebLogic Server as authentication providers in the WebLogic Server Administration Console.

1	The user logging in is authenticated against all available authentication providers. Moe belongs to the RDBMS authentication provider and can log in successfully (unless authentication is set to "REQUIRED" on more than one provider).
2	On successful login to a portal, WebLogic Portal uses its DefaultRoleMapper to see if the user belongs to any delegated administration and visitor entitlement roles, and the user is granted access to only the resources he is allowed to access. The role called "manager" is defined so that anyone belonging to the group called "managerGroup" is part of that role. The HR portlet is set up so that only members of the "manager" role can view it.

If you are using more than the LDAP authentication provider supplied by WebLogic Server, see Using Multiple Authentication Providers in Portal Development.

Related Topics

Implementing Authentication

Using Multiple Authentication Providers in Portal Development

WebLogic Portal supports the use of multiple authentication providers in a Portal domain, which means that users in external providers can log in to your portal applications. It also means that in your code you have access to potentially multiple user stores.

In Portal Controls, JSP tags, and classes in the Portal API that deal with user and group management, you can specify the authentication provider you want. (Use the WebLogic Administration Portal to see a list of available authentication providers.)

For user and group management, the roles you specify in the WebLogic Administration Portal Authentication Security Provider Service determine who can perform certain management tasks in your portal applications.

Note: It is possible (but not recommended) to store an identical username or group name in more than one authentication provider. For example, user "foo" can reside in the default WebLogic Server LDAP provider and in an external RDBMS provider. In that case, WebLogic Portal uses only one user profile for user "foo."

If you are using an RDBMS authentication provider, be aware of case sensitivity when entering names for users and groups. For example, user "Bob" is different than user "bob."

Setting up Multiple Authentication Providers

For information on setting up and configuring multiple authentication providers, see Using Multiple Authentication Providers with WebLogic Portal in the WebLogic Administration Portal help system.

Related Topics

How WebLogic Portal Uses the WebLogic Server Security Framework

Unified User Profiles Overview

If you have an existing store of users, groups, and additional properties (such as address, e−mail address, phone number, and so on), unified user profiles are a necessary part of bringing those user properties into the WebLogic Portal environment, where they can be used for retrieving and editing property values and setting up personalization, delegated administration, and visitor entitlements.

This topic describes the unified user profile, when to use it, and when not to use it.

Note: This topic contains the terms "user store" and "data store." A user store can contain users and groups, as well as additional properties. A data store implies that the store does not have to contain users and groups. It can simply contain properties.

What is a Unified User Profile?

Here is an example that explains what a unified user profile is and does:

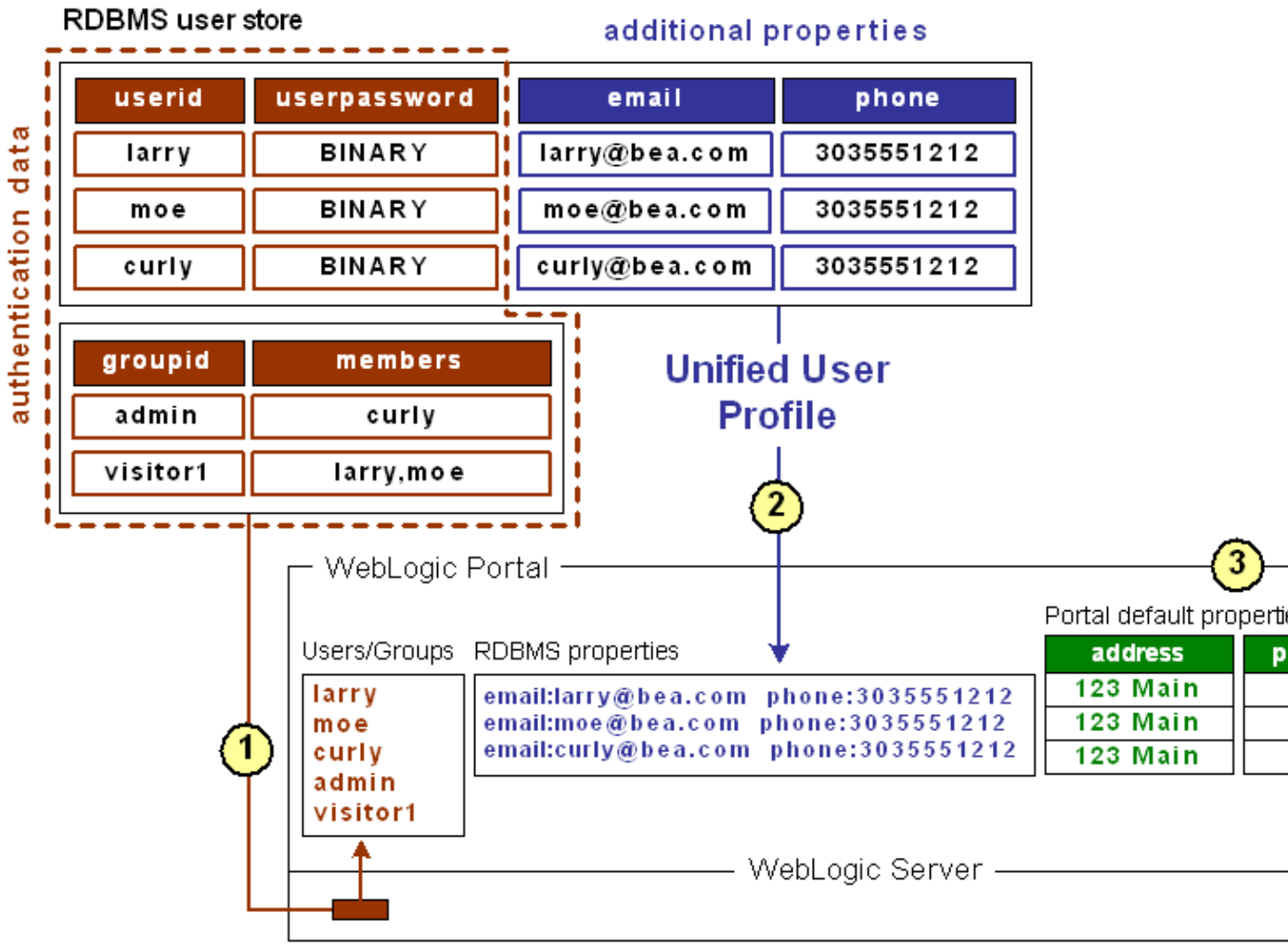
Let's say you're creating a new portal application that you want users to be able to log in to. Let's also say your users are stored in an RDBMS user store outside of the WebLogic environment. You could connect WebLogic Server (your portal application's domain server instance) to your RDBMS system, and your users could log in to your portal application as if their usernames and passwords were stored in WebLogic Server. If authentication was all you wanted to provide through your RDBMS user store, you could stop here without needing a unified user profile.

However, let's say you also stored e−mail and phone number information (properties) for users in your RDBMS user store, and you wanted to be able to access those properties in your portal applications. In this case, you need to create a unified user profile for your RDBMS user store that lets you access those additional properties from your code.

Technically speaking, a unified user profile is a stateless session bean you create (with associated classes) that lets WebLogic Portal read property values stored in external data stores, such as LDAP servers and databases. Once connected to an external data store with a unified user profile, you can use portal JSP tags, controls, and the WebLogic Portal API to retrieve user property values from that store. You can also take the extra step of surfacing these external properties in the WebLogic Administration portal, where the properties can be used to define rules for personalization, visitor entitlements, and delegated administration.

Whether or not you have additional properties stored in your external user store, the external users and groups you connect to WebLogic Server are automatically assigned the default user property values you have set up in WebLogic Portal, without the use of a unified user profile. With the WebLogic Administration Portal, you can change the default WebLogic Portal property values for those users. These values are stored in WebLogic Portal's RDBMS data store using the Portal schema.

The following figure shows where a unified user profile fits between an external user store and the WebLogic environment.



1	This external RDBMS user store, which supports authentication, contains users (principals) and passwords in one database table and groups (principals) in another. Giving a user store authentication capabilities (as an authentication provider or identity asserter) involves configuration steps not associated with the unified user profile configuration process. (See Developing Security Providers for WebLogic Server.) Unified user profile configuration is not dependent on the authentication provider configuration and vice versa. Once the RDBMS authentication provider is connected to WebLogic Server, WebLogic Server (and WebLogic Portal) can see those users and groups. Those users can log in to your portal applications, and you can include those users and groups in your rules for personalization, delegated administration, and visitor entitlements. Also, WebLogic Portal's ProfileWrapper maps the principals to properties kept in the Portal schema, thereby establishing the user profile.
2	***Unified User Profile*** − The same external table that contains users and passwords also contains additional properties (email and phone) for each user. These additional properties are not part of authentication; but they are

	part of each user's profile. If you want to access these properties in your portal applications (with the WebLogic Portal JSP tags, controls, or API), you must create a unified user profile for the RDBMS user store. When you create the unified user profile, the ProfileWrapper includes the external properties in the user profile. The unified user profile consists of a stateless session bean and associated classes that you create. If you want to surface any of these properties in the WebLogic Administration Portal to be used in defining rules for personalization, delegated administration, or visitor entitlements, create a user profile property set for the external user store in addition to implementing your unified user profile session bean. The property set provides metadata about your external properties so that WebLogic Workshop and the WebLogic Administration Portal know how to display them. Properties from an external data store are typically read only in the WebLogic Administration Portal.
3	WebLogic Portal lets you create user/group properties and set default values for those properties. Any user or group in WebLogic Server, whether created in the default LDAP store or brought in through a connection to an external user store, is automatically assigned those default property values; and you can change the default values for each user or group, programmatically or in the WebLogic Administration Portal. This does not involve unified user profiles, because the properties to be retrieved are local, not stored in an external data store. In the illustration, after the authentication provider or identity asserter provides the principals, the ProfileWrapper combines the principals with the external properties of email and phone (retrieved by the unified user profile) and the default WebLogic Portal properties of address and postal code, all of which make up the full user profile.

What a Unified User Profile is Not

A user profile is not a security realm, and it does not provide authentication. It is not even the external user store itself. It is the connection (stateless session bean with associated classes) that lets you read properties in the external user store.

When Should You Create a Unified User Profile?

Create a unified user profile for an external data store if you want to do any of the following:

- Use WebLogic Portal's JSP tags, controls, or API to retrieve property values from that external store.
- Surface external properties in the WebLogic Administration Portal for use in defining rules for personalization, delegated administration, or visitor entitlements. Users and groups are not considered properties.

When Don't You Need a Unified User Profile?

You do not need to create a unified user profile for an external data store if you only want to:

- Provide authentication for users in the external user store.
- Define rules for personalization, delegated administration, or visitor entitlements based only on users or groups in an external user store, not on user properties.
- Define rules for personalization, delegated administration, or visitor entitlements based on the WebLogic Portal user profile properties you create in WebLogic Workshop, which are kept in the Portal schema.

Setting up a Unified User Profile

See Setting up Unified User Profiles.

Related Topics

Using Multiple Authentication Providers in Portal Development (external user stores)

Setting up Unified User Profiles

This topic provides guidelines and instructions on creating a unified user profile to access user/group properties from an external user store. (See Unified User Profiles Overview for overview information.)

Best Practices: When possible, use WebLogic Portal's user profile functionality (default UserProfileManager) to assign properties to users and groups. Given the choice between creating and storing additional properties in an external user store (which requires write access to that external store, which must be implemented) and creating and storing them in WebLogic Portal, doing so in WebLogic Portal can greatly improve performance on accessing property values. If you are storing users and groups in an external store, the ideal configuration is storing only users, groups, and passwords in the external store and creating and setting additional properties in WebLogic Portal. With that configuration, performance is optimal and you do not have to create a unified user profile.

To create a UUP to retrieve user data from external sources, complete the following tasks:

Create an EntityPropertyManager EJB to Represent External Data

Deploy a ProfileManager That Can Use the New EntityPropertyManager

If you have an LDAP server for which you want to create a unified user profile, WebLogic Portal provides a default unified user profile you can modify. See Retrieving User Profile Data from LDAP.

Create an EntityPropertyManager EJB to Represent External Data

To incorporate data from an external source, you must first create a stateless session bean that implements the methods of the com.bea.p13n.property.EntityPropertyManager remote interface. EntityPropertyManager is the remote interface for a session bean that handles the persistence of property data and the creation and deletion of profile records. By default, EntityPropertyManager provides read−only access to external properties.

In addition, the stateless session bean should include a home interface and an implementation class. For example:

```
MyEntityPropertyManager
extends com.bea.p13n.property.EntityPropertyManager

MyEntityPropertyManagerHome
extends javax.ejb.EJBHome
```

Your implementation class can extend the EntityPropertyManagerImpl class. However the only requirement is that your implementation class is a valid implementation of the MyEntityPropertyManager remote interface. For example:

```
MyEntityPropertyManagerImpl extends
com.bea.p13n.property.internal.EntityPropertyManagerImpl
```

or

```
MyEntityPropertyManagerImpl extends
javax.ejb.SessionBean
```

Recommended EJB Guidelines

We recommend the following guidelines for your new EJB:

- Your custom EntityPropertyManager is not a default EntityPropertyManager. A default EntityPropertyManager is used to get/set/remove properties in the Portal schema. Your custom EntityPropertyManager does not have to support the following methods. It can throw java.lang.UsupportedOperationException instead:
  - getDynamicProperties()
  - getEntityNames()
  - getHomeName()
  - getPropertyLocator()
  - getUniqueId()
- If you want to be able to use the portal framework and tools to create and remove users in your external data store, you must support the createUniqueId() and removeEntity() methods. However, your custom EntityPropertyManager is not the default EntityPropertyManager so your createUniqueId() method does not have to return a unique number. It must create the user entity in your external data store and then it can return any number, such as −1.
- The following recommendations apply to the EntityPropertyManager() methods that you must support:
  - getProperty() − Use caching. You should support the getProperties() method to retrieve all properties for a user at once, caching them at the same time. Your getProperty() method should use getProperties().
  - setProperty() − Use caching.
  - removeProperties(), removeProperty() − After these methods are called, a call to getProperty() should return null for the property. Remove properties from the cache, too.
- Your implementations of the getProperty(), setProperty(), removeProperty(), and removeProperties() methods must include any logic necessary to connect to the external system.
- If you want to cache property data, the methods must be able to cache profile data appropriately for that system. (See the com.bea.p13n.cache package in the WebLogic Portal Javadoc.)
- If the external system contains read−only data, any methods that modify profile data must throw a java.lang.UnsupportedOperationException. Additionally, if the external data source contains users that are created and deleted by something other than your WebLogic Portal createUniqueId() and removeEntity() methods can simply throw an UnsupportedOperationException.
- To avoid class loader dependency issues, make sure that your EJB resides in its own package.
- For ease of maintenance, place the compiled classes of your custom EntityPropertyManager bean in your own JAR file (instead of modifying an existing WebLogic Portal JAR file).

Before you deploy your JAR file, follow the steps in the next section.

Deploy a ProfileManager That Can Use the New EntityPropertyManager

A "user type" is a mapping of a ProfileType name to a particular ProfileManager. This mapping is done in the UserManager EJB deployment descriptor.

To access the data in your new EntityPropertyManager EJB, you must do ***one*** of the following:

- Modifying the Existing ProfileManager Deployment Configuration − In most cases you will be able to use the default deployment of ProfileManager, the UserProfileManager. You will modify the UserProfileManager's deployment descriptor to map a property set and/or properties to your custom EntityPropertyManager. If you support the createUniqueId() and removeEntity() methods in your

custom EntityPropertyManager, you can use WebLogic Administration Portal to create a user of type "User" with a profile that can get/set properties using your custom EntityPropertyManager.

- Configuring and Deploying a New ProfileManager − In some cases you may want to deploy a newly configured ProfileManager that will be used instead of the UserProfileManager. This new ProfileManager is mapped to a ProfileType in the deployment descriptor for the UserManager. If you support the createUniqueId() and removeEntity() methods in your custom EntityPropertyManager, you can use the WebLogic Administration Portal (or API) to create a user of type "MyUser" (or anything else you name it) that can get/set properties using the customized deployment of the ProfileManager that is, in turn, configured to use your custom EntityPropertyManager.

ProfileManager is a stateless session bean that manages access to the profile values that the EntityPropertyManager EJB retrieves. It relies on a set of mapping statements in its deployment descriptor to find data. For example, the ProfileManager receives a request for the value of the "DateOfBirth" property, which is located in the "PersonalData" property set. ProfileManager uses the mapping statements in its deployment descriptor to determine which EntityPropertyManager EJB contains the data.

Modifying the Existing ProfileManager Deployment Configuration

If you use the existing UserProfileManager deployment to manage your user profiles, perform the following steps to modify the deployment configuration.

Under most circumstances, this is the method you should use to deploy your UUP. An example of this method is the deployment of the custom EntityPropertyManager for LDAP property retrieval, the LdapPropertyManager. The classes for the LdapPropertyManager are packaged in p13n_ejb.jar. The deployment descriptor for the UserProfileManager EJB is configured to map the "ldap" property set to the LdapPropertyManager. The UserProfileManager is deployed in p13n_ejb.jar.

1. Back up the p13n_ejb.jar file in your enterprise application root directory.
2. From p13n_ejb.jar, extract META−INF/ejb−jar.xml and open it for editing.
3. In ejb−jar.xml, find the <env−entry> element, as shown in the following example:
   ```
   <!−− map all properties in property set ldap to ldap server −−>
   <env−entry>
       <env−entry−name>PropertyMapping/ldap</env−entry−name>
       <env−entry−type>java.lang.String</env−entry−type>
       <env−entry−value>LdapPropertyManager</env−entry−value>
   </env−entry>
   ```

   and add an <env−entry> element after this to map a property set to your custom EntityPropertyManager, a shown in the following example:
   ```
   <!−− map all properties in UUPExample property set to MyEntityPropertyManager −−>
   <env−entry>
       <env−entry−name>PropertyMapping/UUPExample</env−entry−name>
       <env−entry−type>java.lang.String</env−entry−type>
       <env−entry−value>MyEntityPropertyManager</env−entry−value>
   </env−entry>
   ```
4. In ejb−jar.xml, find the <ejb−ref> element shown in the following example:
   ```
   <!−− an ldap property manager −−>
   <ejb−ref>
       <ejb−ref−name>ejb/LdapPropertyManager</ejb−ref−name>
       <ejb−ref−type>Session</ejb−ref−type>
       <home>com.bea.p13n.property.EntityPropertyManagerHome</home>
       <remote>com.bea.p13n.property.EntityPropertyManager</remote>
   </ejb−ref>
   ```

and add an <ejb−ref> element after this to map a reference to an EJB that matches the name from the previous step with ejb/ prepended as shown in the following example:

```
<!−− an example property manager −−>
<ejb−ref>
    <ejb−ref−name>ejb/MyEntityPropertyManager</ejb−ref−name>
    <ejb−ref−type>Session</ejb−ref−type>
    <home>examples.usermgmt.MyEntityPropertyManagerHome</home>
    <remote>examples.usermgmt.MyEntityPropertyManager</remote>
</ejb−ref>
```

The home and remote class names match the classes from your EJB JAR file for your custom EntityPropertyManager.

5. If your EntityPropertyManager implementation handles creating and removing profile records, you must also add Creator and Remover entries. For example:

```
<env−entry>
    <env−entry−name>Creator/Creator1</env−entry−name>
    <env−entry−type>java.lang.String</env−entry−type>
    <env−entry−value>MyEntityPropertyManager</env−entry−value>
</env−entry>

<env−entry>
    <env−entry−name>Remover/Remover1</env−entry−name>
    <env−entry−type>java.lang.String</env−entry−type>
    <env−entry−value>MyEntityPropertyManager</env−entry−value>
</env−entry>
```

This instructs the UserProfileManager to call your custom EntityPropertyManager when creating or deleting user profile records. The names "Creator1" and "Remover1" are arbitrary. All Creators and Removers will be iterated through when the UserProfileManager creates or removes a user profile. The value for the Creator and Remover matches the ejb−ref−name for your custom EntityPropertyManager without the ejb/ prefix.

6. From p13n_ejb.jar, extract META−INF/weblogic−ejb−jar.xml and open it for editing.
7. In weblogic−ejb−jar.xml, find the elements shown in the following example:

```
<weblogic−enterprise−bean>
    <ejb−name>UserProfileManager</ejb−name>
    <reference−descriptor>
        <ejb−reference−description>
            <ejb−ref−name>ejb/EntityPropertyManager</ejb−ref−name>
            <jndi−name>${APPNAME}.BEA_personalization. EntityPropertyManager</jndi−name>
        </ejb−reference−description>
```

and add an ejb−reference−description to map the ejb−ref for your custom EntityPropertyManager to the JNDI name. This JNDI name must match the name you assigned in weblogic−ejb−jar.xml in the JAR file for your customer EntityPropertyManager. It should look like the following example:

```
<weblogic−enterprise−bean>
    <ejb−name>UserProfileManager</ejb−name>
    <reference−descriptor>
        <ejb−reference−description>
            <ejb−ref−name>ejb/EntityPropertyManager</ejb−ref−name>
            <jndi−name>${APPNAME}.BEA_personalization. EntityPropertyManager</jndi−name>
        </ejb−reference−description>
        <ejb−reference−description>
            <ejb−ref−name>ejb/MyEntityPropertyManager</ejb−ref−name>
            <jndi−name>${APPNAME}.BEA_personalization. MyEntityPropertyManager</jndi−nam
        </ejb−reference−description>
```

Note the ${APPNAME} string substitution variable. The WebLogic EJB container automatically substitutes the enterprise application name to scope the JNDI name to the application.

8. Update p13n_ejb.jar for your new deployment descriptors. You can use the jar uf command to update the modified META−INF/ deployment descriptors.
9. Edit your application's META−INF/application.xml to add an entry for your custom EntityPropertyManager EJB module as shown in the following example:
```
<module>
    <ejb>UUPExample.jar</ejb>
</module>
```
10. If you are using an application−wide cache, you can manage it from the WebLogic Administration Console if you add a <Cache> tag for your cache to the META−INF/application−config.xml deployment descriptor for your enterprise application like this:
```
<Cache Name="UUPExampleCache" TimeToLive="60000"/>
```
11. Verify the modified p13n_ejb.jar and your custom EntityPropertyManager EJB JAR archive are in the root directory of your enterprise application and start WebLogic Server.
12. Use the WebLogic Server Administration Console to verify your EJB module is deployed to the enterprise application and then use the console to add your server as a target for the EJB module. You need to select a target to have your domain's config.xml file updated to deploy your EJB module to the server.
13. Use the WebLogic Workshop Property Set Designer to create a User Profile (property set) that matches the name of the property set that you mapped to your custom EntityPropertyManager in ejb−jar.xml for the UserProfileManager (in p13n_ejb.jar). You could also map specific property names in a property set to your custom EntityPropertyManager, which would allow you to surface the properties and their values in the WebLogic Administration Portal for use in creating rules for personalization, delegated administration, and visitor entitlements.

Your new Unified User Profile type is ready to use. You can use the WebLogic Administration Portal to create a user, and it will use your UUP implementation when the "UUPExample" property set is being modified. When you call createUser("bob", "password") or createUser("bob", "password", null) on the UserManager, several things will happen:

- A user named "bob" is created in the security realm.
- A WebLogic Portal Server profile record is created for "bob" in the user store.
- If you set up the Creator mapping, the UserManager will call the default ProfileManager deployment (UserProfileManager) which will call your custom EntityPropertyManager to create a record for Bob in your data source.
- Retrieving Bob's profile will use the default ProfileManager deployment (UserProfileManager), and when you request a property belonging to the "UUPExample" property set, the request will be routed to your custom EntityPropertyManager implementation.

Configuring and Deploying a New ProfileManager

If you are going to deploy a newly configured ProfileManager instead of using the default ProfileManager (UserProfileManager) to manage your user profiles, perform the following steps to modify the deployment configuration. In most cases, you will not have to use this method of deployment. Use this method only if you need to support multiple types of users that require different ProfileManager deployments deployments that allow a property set to be mapped to different custom EntityPropertyManagers based on ProfileType.

An example of this method is the deployment of the custom CustomerProfileManager in customer.jar. The CustomerProfileManager is configured to use the custom EntityPropertyManager (CustomerPropertyManager) for properties in the "CustomerProperties" property set. The UserManager EJB

in p13n_ejb.jar is configured to map the "WLCS_Customer" ProfileType to the custom deployment of the ProfileManager, CustomerProfileManager.

To configure and deploy a new ProfileManager, use this procedure.

1. Back up the p13n_ejb.jar file in your enterprise application root directory.
2. From p13n_ejb.jar, extract META−INF/ejb−jar.xml, and open it for editing.
3. In ejb−jar.xml, copy the entire <session> tag for the UserProfileManager, and configure it to use your custom implementation class for your new deployment of ProfileManager.
   In addition, you could extend the UserProfileManager home and remote interfaces with your own interfaces if you want to repackage them to correspond to your packaging (for example., examples.usermgmt.MyProfileManagerHome, examples.usermgmt.MyProfileManager).
   However, it is sufficient to replace the bean implementation class:
   You must create an <env−entry> element to map a property set to your custom EntityPropertyManager. You must also create a <ejb−ref> element to map a reference to an EJB that matches the name from the PropertyMapping with ejb/ prepended. The home and remote class names for your custom EntityPropertyManager match the classes from your EJB JAR file for your custom EntityPropertyManager.

   Also, if your EntityPropertyManager implementation handles creating and removing profile records, you must also add Creator and Remover entries. This instructs your new ProfileManager to call your custom EntityPropertyManager when creating or deleting user profile records.

   Note: The name suffixes for the Creator and Remover, "Creator1" and "Remover1", are arbitrary. All Creators and Removers will be iterated through when your ProfileManager creates or removes a user profile. The value for the Creator and Remover matches the <ejb−ref−name> for your custom EntityPropertyManager without the ejb/ prefix.
4. In ejb−jar.xml, you must add an <ejb−ref> to the UserManager EJB section to map your ProfileType to your new deployment of the ProfileManager, as shown in the following example:
   ```
   <ejb−ref>
       <ejb−ref−name>ejb/ProfileType/UUPExampleUser</ejb−ref−name>
       <ejb−ref−type>Session</ejb−ref−type>
       <home>com.bea.p13n.usermgmt.profile.ProfileManagerHome</home>
       <remote>com.bea.p13n.usermgmt.profile.ProfileManager</remote>
   </ejb−ref>
   ```

   The <ejb−ref−name> must start with ejb/ProfileType/ and must end with the name that you want to use as the profile type as an argument in the createUser() method of UserManager.
5. From p13n_ejb.jar, extract META−INF/weblogic−ejb−jar.xml and open it for editing.
6. In weblogic−ejb−jar.xml, copy the <weblogic−enterprise−bean> tag, shown in the following example, for the UserProfileManager and configure it for your new ProfileManager deployment:
   ```
   <weblogic−enterprise−bean>
       <ejb−name>MyProfileManager</ejb−name>
       <reference−descriptor>
           <ejb−reference−description>
               <ejb−ref−name>ejb/EntityPropertyManager</ejb−ref−name>
               <jndi−name>${APPNAME}.BEA_personalization. EntityPropertyManager</jndi−name>
           </ejb−reference−description>
           <ejb−reference−description>
               <ejb−ref−name>ejb/PropertySetManager</ejb−ref−name>
               <jndi−name>${APPNAME}.BEA_personalization. PropertySetManager</jndi−name>
           </ejb−reference−description>
           <ejb−reference−description>
               <ejb−ref−name>ejb/MyEntityPropertyManager</ejb−ref−name>
   ```

```
            <jndi-name>${APPNAME}.BEA_personalization. MyEnitityPropertyManager</jndi-nam
        </ejb-reference-description>
    </reference-descriptor>
    <jndi-name>${APPNAME}.BEA_personalization. MyProfileManager</jndi-name>
</weblogic-enterprise-bean>
```

You must create an <ejb−reference−description> to map the <ejb−ref> for your custom EntityPropertyManager to the JNDI name. This JNDI name must match the name you assigned in weblogic−ejb−jar.xml in the JAR file for your custom EntityPropertyManager.
Note the ${APPNAME} string substitution variable. The WebLogic Server EJB container automatically substitutes the enterprise application name to scope the JNDI name to the application.

7. In weblogic−ejb−jar.xml, copy the <transaction−isolation> tag for the UserProfileManager, shown in the following example, and configure it for your new ProfileManager deployment:
```
<transaction-isolation>
    <isolation-level>TRANSACTION_READ_COMMITTED</isolation-level>
    <method>
        <ejb-name>MyProfileManager</ejb-name>
        <method-name>*</method-name>
    </method>
</transaction-isolation>
```
8. Create a temporary p13n_ejb.jar for your new deployment descriptors and your new ProfileManager bean implementation class. This temporary EJB JAR archive should not have any container classes in it. Run ejbc to generate new container classes.
9. Edit your application's META−INF/application.xml to add an entry for your custom EntityPropertyManager EJB module, as shown in the following example:
```
<module>
    <ejb>UUPExample.jar</ejb>
</module>
```
10. If you are using an application−wide cache, you can manage it from the WebLogic Server Administration Console if you add a <Cache> tag for your cache to the META−INF/application−config.xml deployment descriptor for your enterprise application as shown in the following example:
```
<Cache Name="UUPExampleCache" TimeToLive="60000"/>
```

    Verify the modified p13n_ejb.jar and your custom EntityPropertyManager EJB JAR archive are in the root directory of your enterprise application and start your server.
11. Use the WebLogic Server Administration Console to verify your EJB module is deployed to the enterprise application and add your server as a target for the EJB module. You must select a target to have your domain's config.xml file updated to deploy your EJB module to the server.
12. Use the WebLogic Workshop Property Set Designer to create a User Profile (property set) that matches the name of the property set that you mapped to your custom EntityPropertyManager in ejb−jar.xml for the UserProfileManager (in p13n_ejb.jar). You could also map specific property names in a property set to your custom EntityPropertyManager, which would allow you to surface the properties and their values in the WebLogic Administration Portal for use in creating rules for personalization, delegated administration, and visitor entitlements.

Your new Unified User Profile type is ready to use. You can use the WebLogic Administration Portal to create a user, and it will use your UUP implementation when the "UUPExample" property set is being modified. That is because you mapped the ProfileType using an <ejb−ref> in your UserManager deployment descriptor, ejb/ProfileType/UUPExampleUser.

Now, when you call createUser("bob", "password", "UUPExampleUser") on the UserManager, several things will happen:

- A user named "bob" is created in the security realm.
- A WebLogic Portal Server profile record is created for "bob" in the WebLogic Portal RDBMS repository.
- If you set up the Creator mapping, the UserManager will call your new ProfileManager deployment, which will call your custom EntityPropertyManager to create a record for Bob in your data source.
- Retrieving Bob's profile will use your new ProfileManager deployment, and when you request a property belonging to the "UUPExample" property set, the request will be routed to your custom EntityPropertyManager implementation.

Retrieving User Profile Data from LDAP

WebLogic Portal provides a default unified user profile for retrieving properties from an LDAP server. Use this procedure to implement the LDAP unified user profile for retrieving properties from your LDAP server.

The LdapRealm security realm and the LdapPropertyManager unified user profile (UUP) for retrieving user properties from LDAP are independent of each other. They do not share configuration information and there is no requirement to use either one in conjunction with the other. A security realm has nothing to do with a user profile. A security realm provides user/password data, user/group associations, and group/group associations. A user profile provides user and group properties. A password is not a property.

In order to successfully retrieve the user profile from the LDAP server, ensure that you've done the following:

1. If you have already deployed the application on a WebLogic Portal instance, stop the server.
2. Extract p13n_ejb.jar from your application root to a temporary directory.
3. In the temporary directory, open META−INF/ejb−jar.xml, which contains a commented block called "Ldap Property Manager." Uncomment and reconfigure this section using the following steps:
    a. Remove the closing comment mark (−−>) from the end of the "Ldap Property Manager" block, just before the "Property Set Web Service EJB" block, and add it to the end of the first paragraph of the Ldap Property Manager block, like this:

```
<!−− Ldap Property Manager
     To use this, uncomment it here as well as in weblogic−ejb−jar.xml.
     Configure the LDAP connection and settings using the env−entry values
     Do not forget to uncomment the ejb−link and method−permission tags for
     An easy way to ensure you don't miss anything is to search for "ldap"
     weblogic−ejb−jar.xml. Search from the beginning to the end of the file
−−>
```

    b. In the "Ldap Property Manager" block, look for the following default settings and replace them with your own:

ldap://server.company.com:389	Change this to the value of your LDAP server URL.
uid=admin, ou=Administrators, ou=TopologyManagement, o=NetscapeRoot	Change this to the value of your LDAP server's principal.
<env−entry−value>weblogic</env−entry−value>	Change "weblogic" to your LDAP server's principalCredential.

ou=People,o=company.com	Change this to your LDAP server's UserDN.
ou=Groups,o=company.com	Change this to your LDAP server's GroupDN.
<env−entry−value>uid</env−entry−value>	Change "uid" to your LDAP server's usernameAttribute setting.
<env−entry−value>cn</env−entry−value>	Change "cn" to your LDAP server's groupnameAttribute setting.

c. In the "User Profile Manager" and "Group Profile Manager" sections, find the following lines:

```
<!−− <ejb−link>LdapPropertyManager</ejb−link> −−>
<ejb−link>EntityPropertyManager</ejb−link>
```

Uncomment the LdapPropertyManager line and delete the EntityPropertyManager line in both sections.

d. In the <method−permission> and <container−transaction> sections, find and uncomment the following:

```
<!−−
<method>
    <ejb−name>LdapPropertyManager</ejb−name>
    <method−name>*</method−name>
</method>
−−>
```

e. Check to see that you have uncommented all Ldap configurations by doing a search for "Ldap" in the file.

f. Save and close the file.

4. In the temporary directory, open META−INF/weblogic−ejb−jar.xml and perform the following modifications:

a. Uncomment the "LdapPropertyManager" block:

```
LdapPropertyManager
<weblogic−enterprise−bean>
    <ejb−name>LdapPropertyManager</ejb−name>
    <enable−call−by−reference>True</enable−call−by−reference>
    <jndi−name>${APPNAME}.BEA_personalization.LdapPropertyManager</jndi−nam
</weblogic−enterprise−bean>
```

b. In the "Security configuration" section of the file, uncomment the LdapPropertyManager method:

```
<method>
    <ejb−name>LdapPropertyManager</ejb−name>
    <method−name>*</method−name>
</method>
```

c. Check to see that you have uncommented all Ldap configurations by doing a search for "Ldap" in the file.
d. Save and close the file.

5. Replace the original p13n_ejb.jar with the modified version.

   a. Rename the original p13n_ejb.jar to use it as a backup. For example, rename it to p13n_ejb.jar.backup.
   b. JAR the temporary version of p13n_ejb.jar to which you made changes. Name it p13n_ejb.jar.
   c. Copy the new JAR to your application's root directory.
6. Start the server and re−deploy the application.
7. The properties from your LDAP server are now accessible through the WebLogic Portal API, JSP tags, and controls.

   If you want to surface the properties from your LDAP server in the WebLogic Administration Portal (for use in defining rules for personalization, delegated administration, and visitor entitlements), create a user profile property set called ldap.usr, and create properties in the property set that exactly match the names of the LDAP properties you want to surface.

Enabling SUBTREE_SCOPE Searches for Users and Groups

The LdapPropertyManager EJB in p13n_ejb.jar allows for the inspection of the LDAP schema to determine multi−valued versus single−value LDAP attributes, to allow for multiple userDN/groupDN, and to allow for SUBTREE_SCOPE searches for users and groups in the LDAP server. Following are more detailed explanations:

The determination of multi−value versus single−value LDAP attributes allows a developer to configure the ejb−jar.xml deployment descriptor for the LdapPropertyManager EJB to specify that the LDAP schema be used to determine if a property is single− or multi−value.

To enable SUBTREE−SCOPE for users and groups:

1. Stop the server.
2. Extract p13n_ejb.jar from your application root directory to a temporary directory and edit the temporary META−INF/ejb−jar.xml by setting the following env−entries.

```
<!−− Flag to specify if LDAP attributes will be determined to be single value
or multi−value via the schema obtained from the attribute. If false,
then the attribute is stored as multi−valued (a Collection) only if it has
more than one value. Leave false unless you intend to use multi−valued LDAP
attributes that may have only one value. Using true adds overhead to check
the LDAP schema. Also, if you use true beware that most LDAP attributes are
multi−value. For example, iPlanet Directory Server 5.x uses multi−value for
givenName, which you may not expect unless you are familiar with LDAP schemas.
This flag will apply to property searches for all userDNs and all groupDNs. −−>

<env−entry>
    <env−entry−name>config/detectSingleValueFromSchema</env−entry−name>
    <env−entry−type>java.lang.Boolean</env−entry−type>
    <env−entry−value>true</env−entry−value>
</env−entry>

<!−− Value of the name of the attribute in the LDAP schema that is used
to determine single value or multi−value (RFC2252 uses SINGLE−VALUE).
This attribute in the schema should be true for single value and false
```

```
or absent from the schema otherwise. The value only matters if
config/detectSingleValueFromSchema is true. -->

<env-entry>
    <env-entry-name>config/singleValueSchemaAttribute</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>SINGLE-VALUE</env-entry-value>
</env-entry>
```

It is not recommended that true be used for config/detectSingleValueFromSchema unless you are going to write rules that use multi−valued LDAP attributes that have a single value. Using config/detectSingleValueFromSchema = true adds the overhead of checking the LDAP schema for each attribute instead of the default behavior (config/detectSingleValueFromSchema = false), which only stores an attribute as multi−valued (in a Collection) if it has more than one value.

This feature also implements changes that allow you to use SUBTREE_SCOPE searches for users and groups. It also allows multiple base userDN and groupDN to be specified. The multiple base DN can be used with SUBTREE_SCOPE searches enabled or disabled.

A SUBTREE_SCOPE search begins at a base userDN (or groupDN) and works down the branches of that base DN until the first user (or group) is found that matches the username (or group name).

To enable SUBTREE_SCOPE searches you must set the Boolean config/objectPropertySubtreeScope env−entry in the ejb−jar.xml for p13n_ejb.jar.jar to true and then you must set the config/userDN and config/groupDN env−entry values to be equal to the base DNs from which you want your SUBTREE_SCOPE searches to begin.

For example, if you have users in ou=PeopleA,ou=People,dc=mycompany,dc=com and in ou=PeopleB,ou=People,dc=mycompany,dc=com then you could set config/userDN to ou=People,dc=mycompany,dc=com and properties for these users would be retrieved from your LDAP server because the user search would start at the "People" ou and work its way down the branches (ou="PeopleA" and ou="PeopleB").

You should not create duplicate users in branches below your base userDN (or duplicate groups below your base groupDN) in your LDAP server. For example, your LDAP server will allow you to create a user with the uid="userA" under both your PeopleA and your PeopleB branches. The LdapPropertyManager in p13n_ejb.jar.jar will return property values for the first userA that it finds.

It is recommended that you do not enable this change (by setting config/objectPropertySubtreeScope to true) unless you need the flexibility offered by SUBTREE_SCOPE searches.

An alternative to SUBTREE_SCOPE searches (with or without multiple base DNs) would be to configure multiple base DNs and leave config/objectPropertySubtreeScope set to false. Each base DN would have to be the DN that contains the users (or groups) because searches would not go any lower than the base DN branches. The search would cycle from one base DN to the next until the first matching user (or group) is found.
The new ejb−jar.xml deployment descriptor is fully commented to explain how to set multiple DNs, multiple usernameAttributes (or groupnameAttributes), and how to set the objectPropertySubtreeScope flag.

3. Save and close the file.
4. Replace the original p13n_ejb.jar with the modified version:

   a. Rename the original p13n_ejb.jar to use it as a backup. For example, rename it to p13n_ejb.jar.backup.
   b. JAR the temporary version of p13n_ejb.jar to which you made changes. Name it p13n_ejb.jar.
   c. Copy the new JAR to your application's root directory.
5. Start the server and re−deploy the application.

Related Topics

Using Multiple Authentication Providers in Portal Development