

INTEGRATING PASSWORD-PROTECTED APPLICATIONS WITH ORACLE9iAS PORTAL

OVERVIEW

The purpose of this paper is to describe, in detail, how to integrate an existing password-protected application with Oracle9iAS Portal. This is an advanced paper that assumes you are already familiar with Oracle9iAS Portal, Oracle9iAS Single Sign-On Server, and Web Providers.

The paper includes several sections. The first section introduces the various integration approaches and helps you determine which approach is right for your application. The following sections describe the approaches in detail. The final section discusses the advanced topics of deep links and session sharing.

Samples are included in several sections. The samples are based on the Portal Development Kit for Java (PDK-Java) which can be downloaded from Oracle. The PDK-Java includes the complete source code for the samples.

WHERE DO I START?

To integrate an existing application with Oracle9iAS Portal you must create a “provider”. A provider is an application that you register with Oracle9iAS Portal. Each provider manages a set of portlets. Portlets act as windows into your application, display summary information, and provide a way to access the full functionality of the application. Portlets expose application functionality directly in the portal or provide “deep links” that take you to the application itself to perform a task. Since portlets format information for display in a web page the underlying application does not need to be web enabled to be integrated with Oracle9iAS Portal. An application that is not web enabled can be integrated into Oracle9iAS Portal using the external application approach described in this paper.

There are three basic approaches for integrating your application. Each approach is suitable for certain situations or scenarios. The approaches are:

- Partner Application Provider
- External Application Provider
- URL Services Provider

PARTNER APPLICATION PROVIDER

Partner applications are web applications that are tightly integrated with the Oracle9iAS Single Sign-On Server. When a user attempts to access a partner application, the partner application delegates the authentication of the user to the Oracle9iAS Single Sign-On Server. Once a user is authenticated (i.e. has provided a valid username and password) for one partner application, the user does not need to provide a username or password when accessing other partner applications that share the same Single Sign-On Server instance. The Single Sign-On Server determines that the user was successfully authenticated and indicates successful authentication to the new partner application.

A partner application provider is a provider that integrates with a partner application.

ADVANTAGES

1. Provides the tightest integration with Oracle9iAS Portal and Oracle9iAS Single Sign-On Server.
2. Provides the best Single Sign-on experience to users.
3. The application and the portal share the same user repository. This reduces user maintenance.
4. Provides the most secure form of integration because usernames and passwords are not transmitted between the

portal and the provider.

DISADVANTAGES

1. The application must be written using a technology that can be easily integrated with Java or PL/SQL.
2. The application must share the same user repository as the portal even though the application's user community may be a subset of the portal user community. This is a minor issue because the portal pages that expose the application can easily be restricted to the application's user community.
3. The application can only be tightly integrated to one or more Single Sign-On Servers, if they share the same user repository.

EXTERNAL APPLICATION PROVIDER

Applications that manage the authentication of users can be loosely integrated with the Oracle9iAS Single Sign-On Server by registering the applications as external applications. When a user who has been previously authenticated by the Single Sign-On Server accesses an external application for the first time, the Single Sign-On Server attempts to authenticate the user with the external application. The authentication is performed by submitting an HTTP request that combines the registration information and the user's username and password for the application. If the user has not yet registered their username and password for the external application, the Single Sign-On Server prompts the user for the required information before making the authentication request. When a user supplies a username and password for an external application, the Single Sign-On Server maps the new username and password to the user's portal username and stores them. They will be used the next time the user needs authenticating with the external application.

An external application provider is a provider that integrates with an external application. Ideally the external application should have an authentication API you can call from your provider.

ADVANTAGES

1. Allows integration with many portals. However, if there is a "preferred" portal, the application could be integrated as a partner application of that portal and an external application of other portals.
2. Provides a Single Sign-On experience for users. However, users still need to maintain different usernames and passwords. In addition, the external application username mapping must be maintained.
3. Allows integration with multiple portals independent of their user repositories and Single Sign-On Servers.

DISADVANTAGES

1. External applications do not share the same user repository as the portal. There is additional maintenance of user information.
2. The username and password must be transmitted to the provider. This approach is not as secure as a partner application.
3. The application must be written using a technology that can be easily integrated with Java or PL/SQL.

URL SERVICES PROVIDER

A URL Services Provider is a provider that allows you to create portlets by accessing existing content using a URL. Since a URL is used to access the content the technology used to generate the content is not important. The resulting content, which most likely represents a complete HTML page (including HTML headers), can be clipped or filtered. Using filters, you can eliminate the parts of the page you do not want to display in your portlet.

ADVANTAGES

1. The easiest form of integration and the fastest to implement.
2. Allows you to integrate content from any web enabled application regardless of the technology.

DISADVANTAGES

1. Provides the weakest integration with Oracle9iAS Portal.
2. Since authentication occurs by submitting a URL, it is not possible to determine if authentication is successful or not. Error detection is also extremely difficult.
3. Provides the slowest form of integration because multiple requests must be made to return content. Also, content must be filtered to remove at least the HTML header and body tags before it can be included in a portlet.

PREFERRED INTEGRATION APPROACH

In terms of general desirability, the order of preference in most situations is:

1. Partner Application Provider
2. External Application Provider
3. URL Services

However, every situation is different so you should consider the advantages and disadvantages of each approach very carefully before making your decision. To simplify the decision-making task, the following section describes the situations in which each approach is viable.

CHOOSING AN INTEGRATION METHOD

Some of the factors that influence your choice of implementation style may be evident from the brief descriptions in the preceding section. If you are still unsure, the following questions and matrix should guide you towards the appropriate approach for your situation. Each question should be answered either “Yes” or “No”. When you have answered all of the questions, compare your answers to those in the following matrix.

The matrix is intended to guide you in selecting the best approach based on your requirements. However, your requirements may not exactly match one approach in the matrix. For example, you may have easy access to the source code of the application, but URL Services is still the most appropriate integration approach based on your answers to other questions.

Question	Partner Application Provider	External Application Provider	URL Services
Can you easily access the source code of the application?	YES	YES OR NO	NO
Is your application already a partner application? Do you want it to be?	YES	NO	NO
If your application is a partner application, does it share the same user repository as the portal?	YES	NO	NO
Do you need to expose your application in multiple portals (Oracle9iAS Portal or otherwise) that have different user repositories?	NO	YES	YES
Do you need to filter existing content pages to create your portlets?	NO	NO	YES
Is your application written using a technology that allows it to be accessed from Java or PL/SQL?	YES	YES	NO
Can you access an authentication API via a method call?	N/A	YES	NO
Can you access an authentication API via a URL?	N/A	YES	YES
Is your application web enabled?	YES	YES OR NO	YES

Table 1. Provider Selection Matrix

Based on your answers to these questions you should be able to select the integration approach that is most suitable for your specific situation. Having chosen an integration approach, the next step is to examine how the integration works. The following sections describe each approach in detail, focusing primarily on authentication. The sections can be read in isolation. You do not need to read the sections in sequence or understand one section to understand another. This allows you to focus on the implementation approach that is most appropriate for you.

The sample code in each section is based on a web provider implemented using the PDK-Java. Partner application providers and external application providers can also be implemented as database providers using the PL/SQL version of the provider APIs. However, URL Services is only available for web providers implemented using Java.

IMPLEMENTING A PARTNER APPLICATION PROVIDER

This section explains the concepts behind implementing a partner application provider and includes a sample implementation to illustrate the concepts. The code samples are taken directly from the partner application sample that is included with the PDK-Java. Key methods have been selected from the sample implementation to illustrate important concepts. To review the complete source for this application, please download the PDK from Oracle.

SAMPLE APPLICATION FUNCTIONALITY

To understand the flow of logic it helps to understand the functionality of the application. “Flights of Fancy” was originally written as a standalone Java Servlet. Once the standalone version was complete, it was modified so that it could be integrated into Oracle9iAS Portal. The sample was written this way to simulate the process of integrating a real application into Oracle9iAS Portal.

STANDALONE APPLICATION FUNCTIONALITY

The standalone application consists of three screens, a login screen, a summary screen, and a detail screen. When a user first accesses the application, the application determines if the user is authenticated. If the user is not authenticated, the login screen displays. When a user successfully logs in to the application, the application creates a cookie so that it knows the user is logged in when it processes subsequent requests.

After the user logs in to the application, a screen showing a summary of flight information displays. Clicking a flight link takes the user to the detail screen that displays the details associated with the selected flight.

PARTNER APPLICATION PROVIDER FUNCTIONALITY

The partner application provider consists of a single portlet that displays the same content as the summary screen in the standalone application. When a user views the portlet for the first time, the provider’s `initSession` method is called to authenticate the user. If the authentication is successful, the summary screen displays in the form of a portlet. Clicking a flight within the portlet takes the user to the application itself, which displays the detail page as a full screen, not as a portlet. Links that take the user from a portlet to an application in this manner are referred to as “deep links”.

AUTHENTICATION OVERVIEW

Authentication of users in partner applications is somewhat different than in conventional applications. Partner applications delegate user authentication to a third party, the Single Sign-On Server. If the user has not been authenticated, the Single Sign-On Server displays a login page prompting the user to enter a username and password. The login page submits the username and password back to the Single Sign-On Server.

If successfully authenticated, the Single Sign-On Server creates a special cookie containing information about the user. For security, the application may encrypt the contents of the cookie. The cookie is sent back to the user’s browser, but is scoped so that it is only visible to the Single Sign-On Server. It is not passed to other listeners. After creating the cookie, the Single Sign-On Server re-directs the web browser to the “success URL” specified by the partner application. At this point, the partner application creates an application session cookie which contains information the application needs to re-establish the session later. The contents may be encrypted using Single Sign-On Server APIs (SSO SDK). Upon making subsequent requests to the partner application, the partner application detects the presence of the partner application session cookie and from the cookie knows that the user is already authenticated.

If the user later accesses another partner application, that application looks for its application specific session cookie. If the cookie is not found the application redirects the request to the Single Sign-On Server as described previously. However, this time the Single Sign-On Server detects the presence of the user's Single Sign-On Server cookie. This indicates that the user is already authenticated so the Single Sign-On Server redirects the browser to the "success URL" of the second partner application without prompting the user to enter a username and password again. At this point, the partner application creates its own application specific session cookie.

To secure the application session cookies, the content may be encrypted. The Single Sign-On SDK (SSOSDK) includes APIs to encrypt and decrypt strings. These APIs can be used to encrypt the contents of the application session cookie.

PARTNER APPLICATION PROVIDERS AND THE SINGLE SIGN-ON SERVER

Partner application providers, similar to partner applications, must create a partner application session cookie. However, a partner application provider does not need to contact the Single Sign-On Server. Instead, the partner application provider "trusts" the portal to authenticate the user on the provider's behalf. This is possible because the portal is, itself, a partner application.

When the provider receives a call to establish a partner application session, the provider should first determine if a partner application session already exists. This is possible if the provider's session has expired and the portal calls the provider to re-establish the session. If a session cookie already exists, you may need to take action to ensure that the session is still active and has not expired. If the session has expired, you should establish a new one.

If a session cookie does not exist, you should create one. It should be noted that it is possible that the current user has not logged into the portal, but is accessing the portlet from a public page that does not require authentication. If this is the case, you should still establish a session, but ensure that any sensitive content is not displayed when the portlet is rendered. If the user subsequently logs in to the portal, the portal discards any existing cookies and contacts the provider to establish new sessions for each provider the next time the user views a portlet from that provider.

Partner application providers must trust the portal to authenticate the user in this way because the provider cannot perform the authentication itself. Authenticating the user directly requires the provider to redirect the "browser" to the Single Sign-On Server and provide success and failure URLs. This is not possible due to the Web Provider architecture. The primary reason for this is that the authentication occurs in response to an API call from the portal to the provider. The Single Sign-On Server cannot imitate that call upon successful authentication to the `initSession` method to complete its normal processing.

Note: This behavior relates to Oracle9iAS Portal release 3.0.9. In Oracle9iAS Portal release 3.0.8 and earlier, session cookies are not cleared out when an unauthenticated user accessing the portal using a public session logs into the portal as an authenticated user. This means that a session cookie may be carried over from a public session to an authenticated session. Consequently, you should always verify the username associated with a user and display content that is appropriate for the user to see.

LOGICAL AUTHENTICATION SEQUENCE

The following sequence of events assumes you are using Oracle9iAS Portal release 3.0.9 or later.

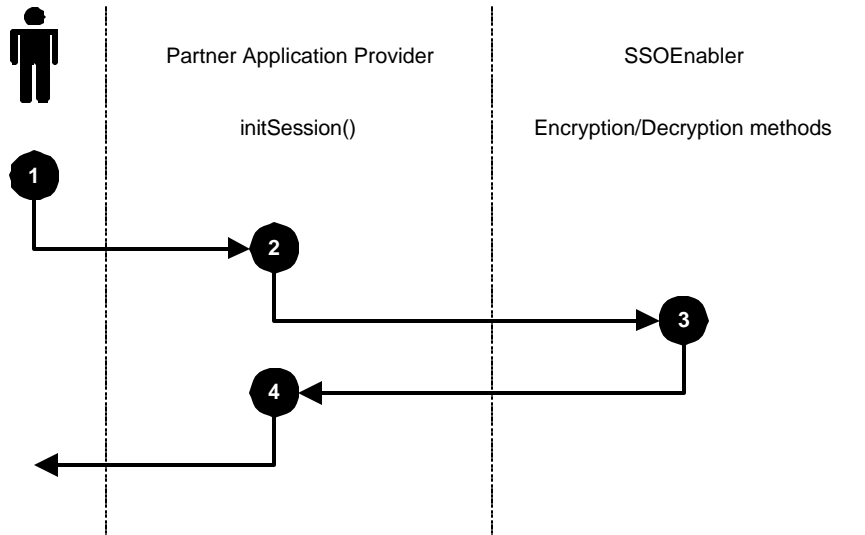


Figure 1. Partner Application Provider Logical Authentication Sequence

1. An authenticated portal user requests a portal page that contains a portlet from the partner application. Portal calls the partner provider's `initSession` method.
2. The partner application provider checks for the existence of a partner application session cookie. If a cookie does not exist, the provider creates one. If the cookie already exists, the provider does not need to do anything.
3. The provider optionally utilizes the SSO SDK to encrypt the contents of the new partner application cookie.
4. If the partner application session cookie does not exist, the `initSession` method creates a new cookie and optionally encrypts the contents using the SSO SDK utilities and establishes a provider session. The cookie is returned to the portal and is sent to the provider on subsequent requests to view portlets.

Note: If you have access to your own encryption tools, you can use them to encrypt the cookie contents. Use of the SSO SDK is entirely optional.

SAMPLE IMPLEMENTATION

The sample implementation utilizes the SSO SDK to demonstrate how you can use the SDK to encrypt the contents of your application session cookie. All interaction with the SSO SDK is encapsulated in the `PartnerSSOEnabler` class.

When a user first accesses a portal page that contains a portlet from the partner application provider, the portal calls the provider's `initSession` method. The `PartnerSSOEnabler` object is used to access the partner application session cookie. If the session cookie does not exist, but the portal has authenticated the user, a new cookie is generated.

```

public Object[] initSession(ProviderUser user, ExternalPrincipal extuser)
    throws ProviderException, AuthenticationException
{

    String bakedAppCookie = null;
    String appCookieName = null;
    String appCookieDomain = null;
    String appCookieScope = null;
    String appCookieDesc = null;
    HttpServletRequest request = null;

    //call super initSession
    super.initSession(user, extuser);

    try
    {
        // use the SSO utilities to determine if application session cookie exists
        request = (HttpServletRequest)pr.getAttribute(HttpProvider.SERVLET_REQUEST);
        if (mpSsoBean.getBakedUserInfo(request) == null)
        {
            // Set and Get cookie information.
            // "baking" the cookie is the SSO terminology for encrypting
            // the information that will be included in the cookie.
            bakedAppCookie = mpSsoBean.bakeAppCookie(user.getName());
            appCookieName = mpSsoBean.getAppCookieName();
            appCookieDomain = mpSsoBean.getAppCookieDomain();
            appCookieScope = mpSsoBean.getAppCookieScope();
            appCookieDesc = mpSsoBean.getAppCookieDesc();
        }
    }
    catch(Exception e1)
    {
        throw new ProviderException(e1.toString());
    }

    Cookie jspAppCookie = new Cookie(appCookieName,bakedAppCookie);
    Cookie[] cookies = new Cookie[1];
    jspAppCookie.setDomain(appCookieDomain);

    // In-memory cookie for better security
    jspAppCookie.setMaxAge(-1);
    jspAppCookie.setPath(appCookieScope);
    jspAppCookie.setComment(appCookieDesc);
    cookies[0] = jspAppCookie;

    return cookies;
}

```

partner applications should not throw AuthenticationExceptions. This exception class has a special use and is intended for use with external application providers only.

perform standard provider session initialization.

determine if a partner application session cookie exists.

this method cycles through the cookies looking for the partner application session cookie.

encrypt the username before adding to the cookie and initialize the cookie attributes.

create the Cookie object and set its attributes.

return the new partner application session cookie to the portal.

In the preceding code fragment, a method is called to encrypt the username before adding it to the cookie. The method that is used to encrypt the cookie is included in the following. This method uses the `SSOEnablerUtil` class to perform the actual encryption. The Java classes do not perform the encryption in Java. Instead, the Java classes call PL/SQL functions that are included in the SSO SDK to perform the encryption. The PL/SQL uses the DES encryption routines that are provided with Oracle8i.

```

public String bakeAppCookie(String unbakedAppCookie)
    throws SSOEnablerException
{
    try
    {
        //Open connection
        Connection dbCon = getDbConnection();

        // Set JSP application cookie
        SSOEnablerUtil ssoAppUtil = new SSOEnablerUtil(dbCon);

        String bakedAppCookie =
            ssoAppUtil.bakeAppCookie(mListenerToken, unbakedAppCookie);

        // Close database connection
        closeDbConnection(dbCon);

        return bakedAppCookie;
    }
    catch(Exception e)
    {
        throw new SSOEnablerException(e.toString());
    }
}

```

database connection is required by SSO SDK methods to encrypt cookie contents in the database.

encrypt the cookie string for inclusion in the cookie.

PORTLET RENDERERING OVERVIEW

After successful completion of the `initSession` method, the content of the portlet can be rendered. This section describes the logical sequence of events and a sample implementation.

LOGICAL SEQUENCE OF EVENTS

As with the authentication sequence, the partner application provider relies on the SSO SDK to decrypt the partner application session cookie.

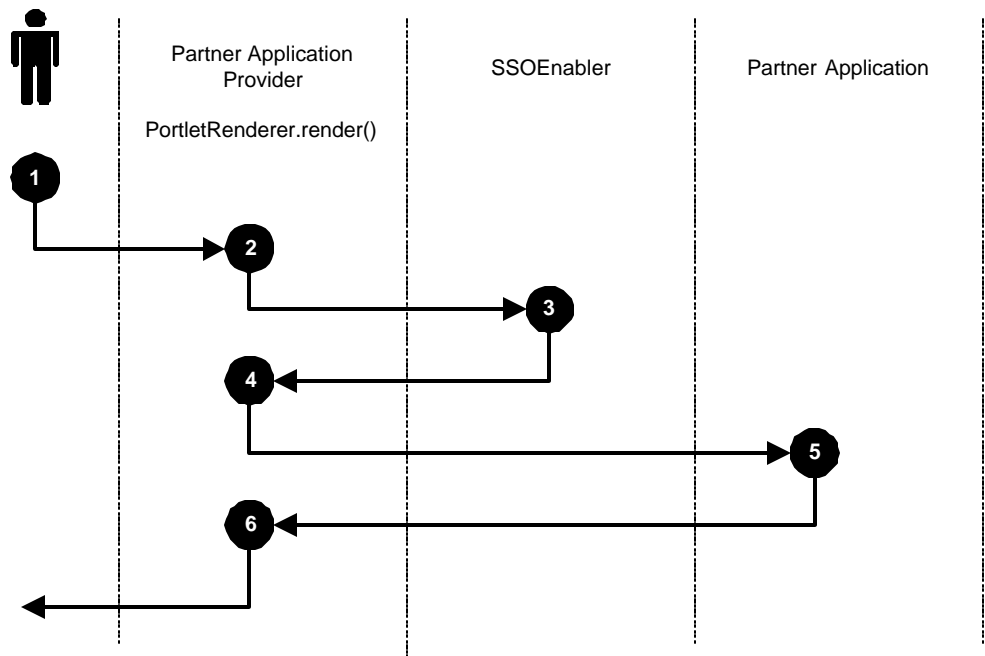


Figure 2. Partner Application Provider Logical Rendering Sequence

1. A user submits a request for a portal page containing the partner application provider's portlet.
2. Upon receiving the request to view the portlet in SHOW mode, the provider calls the `SSOEnabler` to determine if the user is authenticated.
3. The partner application uses the `PartnerSSOEnabler` class to determine if the partner application cookie exists, and if the cookie exists to decrypt the contents. The `SSOEnablerUtil` class makes a JDBC call to a database that contains the SSO SDK PL/SQL packages. These packages decrypt the cookie contents and return the decrypted values to the `PartnerSSOEnabler`.
4. The partner application provider locates the application session cookie and extracts the user's username from the cookie. If the portlet could display sensitive information, the portlet should verify the privileges of the user before displaying any sensitive content. The content in the sample application is not sensitive, but the code extracts the username from the cookie to demonstrate how the SSO SDK can be used.

Note: At this point, the partner application session cookie should always exist.

5. The partner application gathers information and either returns it to the portlet for formatting or formats it and writes the formatted output directly to the HTTP Response stream.
6. If information returned by the external application needs formatting, the portlet formats the information and writes it to the HTTP Response stream. After writing the portlet content, the portlet completes the operation by generating the portlet footer.

Note: If you use one of the `ManagedRenderers` provided with the PDK-Java, you do not need to generate the portlet header and footer (Steps 2 and 6). The framework does this for you.

SAMPLE IMPLEMENTATION

```
public void render(PortletRenderRequest pr)
    throws PortletException, AccessControlException
{
    try
    {
        PrintWriter out = pr.getWriter("text/html");

        // Writes container -- handles rendering the border if necessary
        PortletRendererUtil.renderPortletHeader(pr, out, null);
        new FlightDispatch().process(pr);

        PortletRendererUtil.renderPortletFooter(pr,out);
    }
    catch (IllegalArgumentException ie)
    {
        throw new PortletException(ie);
    }
    catch (java.io.IOException ioe)
    {
        throw new PortletException(ioe);
    }
    catch (Exception e)
    {
        throw new PortletException(e);
    }
}
```

render portlet header.

delegate generate of
portlet content to the
partner application.

render portlet header.

The `process` method that is called from `render` (in the preceding sample) is responsible for locating the partner application cookie and rendering the content. The `process` method is shown in the following:

```

public void process(PortletRenderRequest pr)
    throws Exception
{
    mPR = pr;
    mIsWebProvider = true;
    String l_userInfo = null;

    try
    {
        PartnerSSOEnabler enabler = PartnerProvider.getSSOBean();
        HttpServletRequest request;
        request = (HttpServletRequest)pr.getAttribute(HttpProvider.SERVLET_REQUEST));

        l_userInfo = enabler.getSSOUserInfo(request);

        //Begin processing by checking the login status.
        if (l_userInfo != null)
        {
            Enumeration argNames = pr.getAttributeNames();
            String name = null;
            String username = null;
            String action = null;
            String id = null;
            //Pull out the needed parameters from the argument array.
            while(argNames.hasMoreElements())
            {
                name = (String)argNames.nextElement();
                if (name.equalsIgnoreCase(ACTION))
                    action = (String)pr.getAttribute(name);
                else if (name.equalsIgnoreCase(ID))
                {
                    id = (String)pr.getAttribute(name);
                }
                else if (name.equalsIgnoreCase(USERNAME))
                {
                    username = (String)pr.getAttribute(name);
                }
            }
            String user = pr.getUser().getName();
            StringBuffer server = new StringBuffer(pr.getScheme());
            server.append("://");
            server.append(pr.getServerName());
            server.append(":");
            server.append(pr.getServerPort());

            showList(l_userInfo, server, pr.getWriter("text/html"));
        }
        //if loggedin
        else
        {
            throw new ServletException("User is not authenticated.");
        }
    }
    catch (Exception e)
    {
        throw e;
    }
}

}

```

get the PartnerSSOEnabler instance from PartnerProvider.

locate the partner application session cookie and decrypt the username. Username is decrypted at this point because it is needed to render the content later.

prepare arguments from HTTP request.

generate the portlet content using the same method that the partner application uses.

}//FlightDispatch

IMPLEMENTING AN EXTERNAL APPLICATION PROVIDER

This section explains the concepts behind implementing an external application provider and includes a sample implementation to illustrate the concepts. The code samples are taken from the external application sample that is

included with the PDK-Java. Key methods have been selected from the sample implementation to illustrate key points. To see the complete source for this application, please download the PDK from Oracle.

SAMPLE APPLICATION FUNCTIONALITY

To understand the flow of logic it helps to understand the functionality of the application. “Flights of Fancy” was originally written as a standalone Java Servlet. Once the standalone version was complete, it was modified so that it could be integrated into Oracle9iAS Portal. The sample was written this way to simulate the process of integrating a real application into Oracle9iAS Portal.

STANDALONE APPLICATION FUNCTIONALITY

The standalone application consists of three screens, a login screen, a summary screen, and a detail screen. When a user first accesses the application, the application determines if the user is authenticated. If the user is not authenticated, the login screen displays. When a user successfully logs in to the application, the application creates a cookie so that it knows the user is logged in when it processes subsequent requests.

After the user logs in to the application, a screen showing a summary of flight information displays. Clicking a flight link takes the user to the detail screen that displays the details associated with the selected flight.

EXTERNAL APPLICATION PROVIDER FUNCTIONALITY

The external application provider consists of a single portlet that displays the same content as the summary screen in the standalone application. When a user views the portlet for the first time, the provider’s `initSession()` method is called to authenticate the user. If the authentication is successful, the summary screen displays in the form of a portlet. Clicking a flight within the portlet takes the user to the application itself, which displays the detail page as a full screen, not as a portlet. Links that take the user from a portlet to an application in this manner are referred to as “deep links”.

AUTHENTICATION OVERVIEW

This section describes the steps for authenticating a portal user with your external application.

LOGICAL AUTHENTICATION SEQUENCE

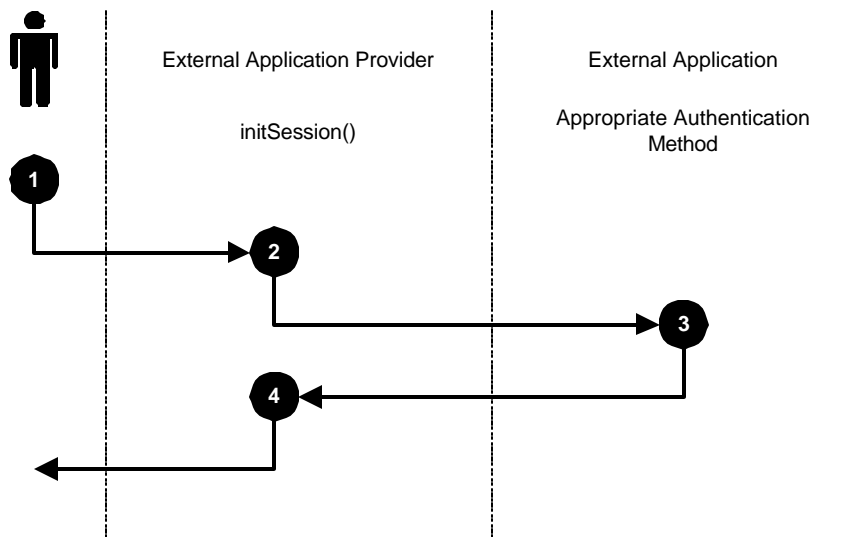


Figure 3. External Application Provider Logical Authentication Sequence

1. An authenticated portal user requests a portal page that contains a portlet from the external application.
2. The portal calls the provider's `initSession` passing an `ExternalPrincipal` object as one of the parameters.
3. The `initSession` method extracts the necessary information from the `ExternalPrincipal` object and calls an appropriate API to authenticate the user. The actual API called and the parameters required depend on the external application.
4. If authentication is successful, the `initSession` method establishes a provider session for the user. At this point, information may be stored in the servlet session indicating that the user has been successfully authenticated. Additional cookies may also be created if they are needed by the external application to identify an authenticated user. Any cookies created in `initSession` are returned to the portal and passed back to the provider when subsequent requests are made.

If authentication fails, the `initSession` method throws an `AuthenticationException`. This exception is captured by the framework and returned to the portal. Upon detecting the `AuthenticationException`, the portal generates a portlet that contains an error message and a link to the Single Sign-On Server page where the user can update mapping information (username and password) for the external application. Subsequent attempts to authenticate the user repeat the preceding steps until authentication is successful.

SAMPLE IMPLEMENTATION

The key to external application authentication is the `ExternalPrincipal` class. This class encapsulates all the information regarding the external application and the mapped user. Most external applications only require the mapped username and password to authenticate the user. The remaining information is provided for completeness, and for the few external applications whose only method of authentication is via a URL.

- Mapped user name – The username of the current portal user in the external application. This is the username the user enters when the user creates the external application user mapping in the Single Sign-On Server.
- Password – The password of the current portal user in the external application. This is the password the user enters when the user creates the external application user mapping in the Single Sign-On Server.
- Authentication URL – The URL used by the Single Sign-On Server to perform authentication.
- Authentication Method (GET/POST) – The HTTP method that should be used when submitting a request to the Authentication URL.
- Additional name/value pairs – Additional name/value pairs as specified in the external application registration.

The following code is the `initSession` method from `ExternalProvider.java`. This represents Steps 2 and 3 in the preceding description. In this implementation the `ApplicationLogin` class encapsulates the logic for authenticating a user.

```

public Object[] initSession(ProviderUser user,
                           ExternalPrincipal extuser)
    throws ProviderException, AuthenticationException
{

```

```

    ApplicationLogin login = new ApplicationLogin();
    Cookie appCookie = null;
    Cookie[] cookies = new Cookie[1];

```

```

    if ((appCookie = login.getApplicationCookie(extuser)) == null)
    {
        // external login failed so raise an
        // AuthenticationException to notify
        // the portal of the failure.
        throw new AuthenticationException("Login to Fancy Flights Failed");
    }

```

```

    // external login was successful so establish JPDK session
    super.initSession(user, extuser);
    appCookie.setMaxAge(1800);
    cookies[0] = appCookie;
    return cookies;
}

```

authenticate user
and create
application
session cookie.

throw an
AuthenticationExceptio
n if user cannot be
authenticated.

successful
authentication so
establish provider
session.

send application
session cookie back to
portal. It will be
included in
subsequent SHOW
requests.

To authenticate the portal user, the `getApplicationCookie` method (shown in the following sample) first extracts the mapped username and password from the `ExternalPrincipal` object, and then searches for that combination of username and password in its user repository, in this case a simple array.

If the username and password combination is found, the method creates a new cookie that represents the portal user's session in the external application. This is the same cookie that the external application generates and returns to the browser when running as a standalone application.

If the username/password combination is not found, a cookie does not get constructed and the method returns null. As in the `initSession` method in the preceding sample, this results in an `AuthenticationException` being thrown which is captured by the portal and processed as described in Step 4 of the authentication sequence.

```

public Cookie getApplicationCookie(ExternalPrincipal user)
{
    String username = user.toString();
    String password = user.getPassword();
    Cookie cookie = null;

    for (int i=0; i<VALIDUSERNAMES.length; i++)
    {
        if (username.equalsIgnoreCase(VALIDUSERNAMES[i]) &&
            password.equalsIgnoreCase(VALIDPASSWORDS[i]))
        {
            cookie = new Cookie(COOKIE_NAME,username);
            cookie.setPath("/servlets/flights");
            break;
        }
    }
    return cookie;
}

```

extract the username and password from the ExternalPrincipal object.

authenticate the user against valid external application user list.

create the external application cookie which is used to validate if users are authenticated in subsequent requests.

PORTLET RENDERING OVERVIEW

This section describes the sequence of events that may occur when the external application provider receives a render request.

LOGICAL RENDERING SEQUENCE

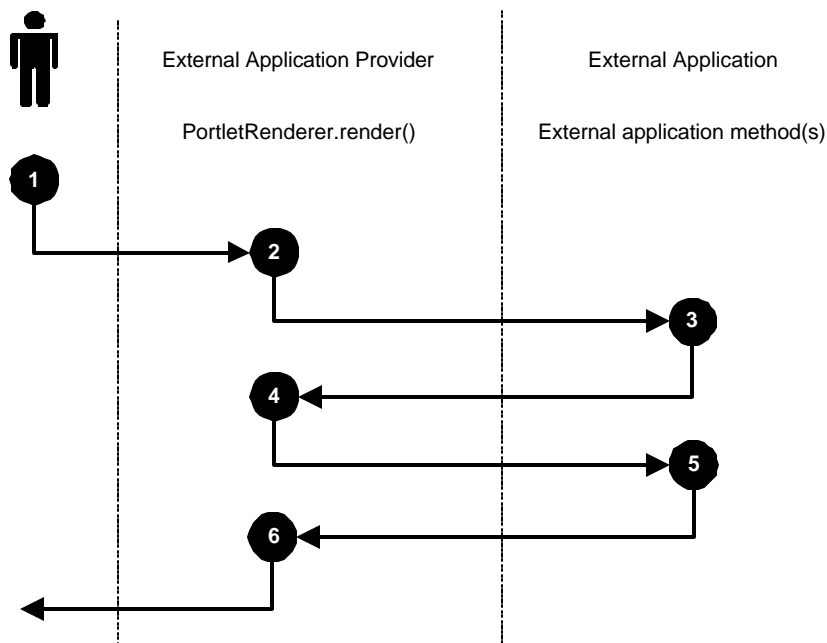


Figure 4. External Application Provider Logical Rendering Sequence.

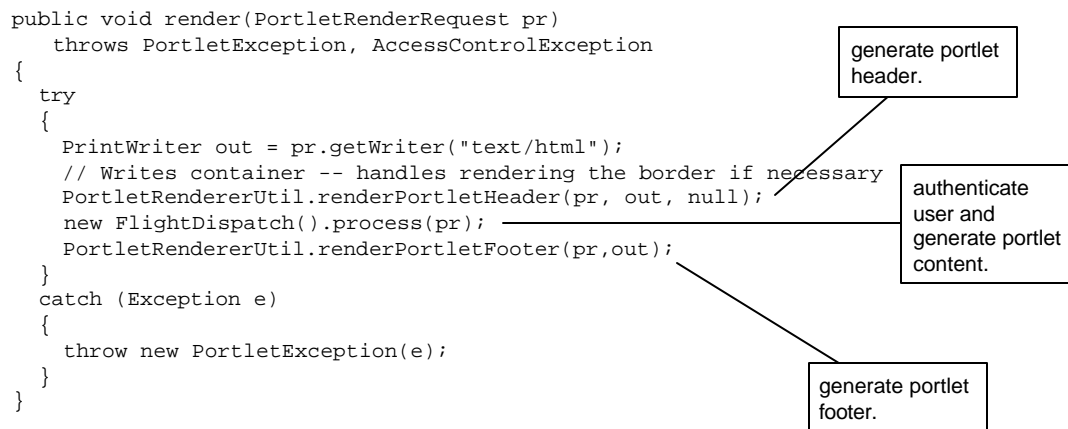
1. User submits a request for a portal page containing the external application provider's portlet.
2. Upon receiving the request to view the portlet in SHOW mode, the portlet verifies the identity of the portal user by calling to one or more external application methods.

3. The external application verifies the identity of the portal using information from one or more of the following sources. The actual source depends on the implementation of `initSession`.
 - External provider's session
 - Cookies created by the external application provider or external application
 - Information stored in a user session maintained independently by the external application
 - Parameters from the SHOW request (`_mapped_name`)
4. After authenticating the current user, the portlet renders the portlet header and makes one or more calls to the external application for the content that is to be rendered by the portlet.
5. The external application gathers information and either returns it to the portlet for formatting or formats it and writes the formatted output directly to the HTTP Response.
6. If information returned by the external application needs formatting, the portlet formats the information and writes it to the HTTP Response. After writing the portlet content, the portlet writes the portlet footer.

Note: If you use one of the `ManagedRenderers` provided with the PDK-Java, you do not need to generate the portlet header and footer (Steps 2 and 6). The framework does this for you.

SAMPLE IMPLEMENTATION

The following method is the `render` method from `FlightRenderer.java` which implements the `PortletRenderer` interface. This is the entry point for the call to render the portlet. In this implementation, the `render` method simply draws the portlet header and footer. The content for the portlet is generated by the `FlightDispatch` class which has two versions of the `process` method, one called from the external application directly and the second called from the `FlightRenderer` class.



The following method is the `process` method called from the `render` method in the preceding sample. This method verifies that the user is authenticated by calling `ApplicationLogin.isLoggedIn` before calling a common method, `showList`, to generate the portlet content.

```

public void process(PortletRenderRequest pr)
    throws Exception
{
    mPR = pr;
    mIsWebProvider = true;
    ApplicationLogin login = new ApplicationLogin(pr);
    try
    {
        ...
        ...
        ...
        if (login.isLoggedIn())
        {
            user = login.getUser();
            showList(user, server, out);
        }
    }
    catch (Exception e)
    {
        throw e;
    }
}

```

construct ApplicationLogin object to verify user authentication.

code removed for purpose of this document.

verify user is authenticated.

generate portlet content using same method that would be used by the external application.

The following method is the ApplicationLogin constructor called in the preceding method. It is this constructor that searches for the external application's session cookie, created in `initSession`, to verify that the current user is authenticated.

```

public ApplicationLogin(PortletRenderRequest pr)
    throws Exception
{
    Object[] cookieList = pr.getCookies();
    Cookie cookie = null;
    for (int i = cookieList.length - 1; i >= 0; i--)
    {
        cookie = (Cookie)cookieList[i];
        if (cookie.getName().equalsIgnoreCase(COOKIE_NAME))
        {
            mCookie = cookie;
            mUser = mCookie.getValue();
            mLoggedIn = true;
            break;
        }
    }
}

```

get array of cookies from the incoming request.

look for the external application session cookie that indicates an authenticated user.

cookie was found so set tracking variable to true.

USING URL SERVICES

Integrating an application using URL services is somewhat different than the partner application and external application approaches. Instead of tying directly into an existing application, URL Services integrates loosely with an existing application, treating the application as a black box and accessing it only using URLs. In this respect, URL Services acts as a web browser and is treated as any other web client by the remote application. The advantage of this approach is that an existing web enabled application can be quickly integrated into the portal without impacting the application in any way. This means that you can integrate both local intranet applications and remote Internet applications into your portal.

For many applications, the integration effort is confined to defining the provider declaratively using the provider definition file (provider.xml).

AUTHENTICATION OVERVIEW

URL Services performs authentication using a similar approach to the Single Sign-On Server, leveraging the external application functionality and password store to maintain user mappings. The authentication procedure involves constructing an appropriate Login URL for the external application and submitting a request to that URL using the mapped username and password from the password store. Any cookies that are returned in response to the authentication request are saved and sent back to the remote application with all subsequent requests.

In the current release of URL Services, the authentication information must be specified both in the provider definition file and in the Single Sign-On Server by registering the remote application as an external application. The provider definition entry is required because the PDK-Java did not expose all the information that was needed from the Single Sign-On Server at the time URL Services was developed. However, you must still register your external application with the Single Sign-On Server because the registration is needed to save username and password mappings in the password store.

Future versions of URL Services will derive all authentication information from the Single Sign-On Server. The external application support from the Single Sign-On Server is required because it provides the mapping of portal users to external application users. This allows portal users to access their own accounts on the remote system instead of relying on a single account shared by all users.

LOGICAL AUTHENTICATION SEQUENCE

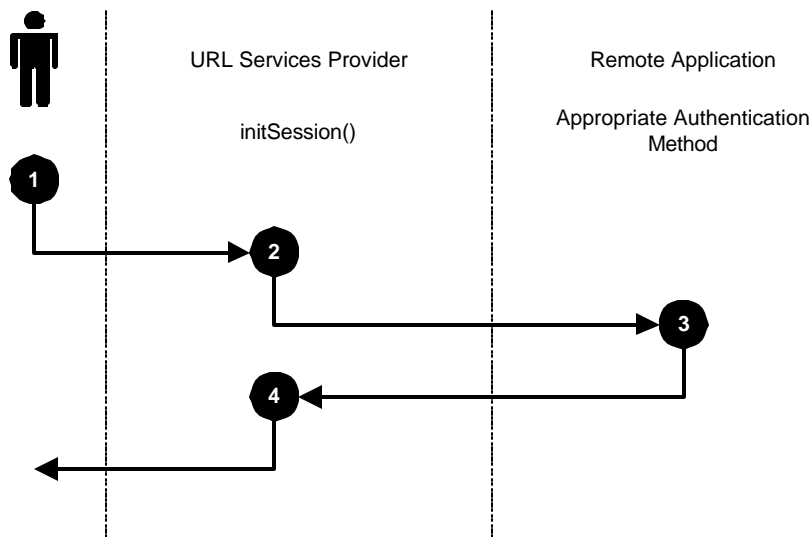


Figure 5. URL Services Logical Authentication Sequence

1. User views a portal page containing a portlet from the URL Services provider. Since this is the first time the user views a portlet from the URL Services Provider in the current portal session, the provider's `initSession` method is called to establish a provider session.
2. The URL Services provider establishes a provider session between the portal and the provider. It then uses the authentication information specified in the provider definition file and constructs a login URL for the remote application. The login URL is constructed using the base login URL for the application, the portal user's mapped username/password (which is obtained from the Single Sign-On Server) and any additional application specific parameters. Having constructed the URL, the URL Services Provider submits an HTTP request using the URL.
3. The remote application receives an HTTP request and processes the request. The processing that occurs at this

point is entirely dependent on the remote application which can be thought of as a “black box” when using URL Services.

4. If any cookies are created in response to the request they are returned to the URL provider and stored in the provider session. They can then be used in subsequent requests to the remote application.

SAMPLE IMPLEMENTATION

Unlike partner applications and external applications, the behavior described in the preceding section is encapsulated within the URL Services framework. No provider specific code needs to be written. However, the provider does require a certain amount of information to be declared in the provider definition file. This section describes the <authentication> and <proxyInfo> elements of the provider definition file. These elements are specific to URL Services providers.

The <authentication> element is an optional element that encapsulates information about the type of authentication a URL Services provider must perform. The sub-elements of <authentication> are:

- <authType> is a required tag under authentication and specifies what kind of authentication is supported. This tag has two possible values: none indicates no authentication, and ExternalApp indicates that authentication is performed by retrieving the authentication URL, Login user ID, Login password, authentication type, and other login parameters from the Single Sign-On Server.
- <loginURL> is a required tag under authentication and specifies the value of the login URL. This value should be the same as the login URL specified when creating an external application.
- <requestType> is a required tag under authentication and specifies the request type for submission of the Login URL. The tag has two possible values: GET and POST.
- <userFieldName> is a required tag under authentication and specifies the name of the User ID field as specified in the Login form when creating an external application. This is not the value of the User ID field, but the name. For example: p_username.
- <userPwdName> is a required tag under authentication and specifies the name of the Password field as specified in the login form when creating an external application. This is not the value of the Password field, but the name. For example: p_password.
- <fieldName> is an optional tag under authentication and specifies the name of an additional field as specified in the login form. This is the same as the “field1” name as specified at the time of creation of the External application. You can use this same tag name for each additional field needed.

A sample <authentication> element might look like this:

```
<provider class="oracle.portal.provider.v1.http.DefaultURLProvider">
  <session>true</session>
  <authentication>
    <authType>ExternalApp</authType>
    <loginURL>http://my.yahoo.com/login</loginURL>
    <requestType>POST</requestType>
    <userFieldName>p_username</userFieldName>
    <userPwdName>p_password</userPwdName>
    <fieldName>p_ssn</fieldName>
    <fieldName>Submit</fieldName>
  </authentication>
  . . .
  . . .
</provider>
```

The `<proxyInfo>` element is an optional element that allows the URL Services provider to access internet applications that are outside a firewall. The sub-elements of `<proxyInfo>` are:

- `<proxyHost>` is an optional tag and specifies the name of the proxy server.
- `<proxyPort>` is an optional tag and specifies the port for the proxy server.

A sample `<proxyInfo>` element might look like this:

```
<proxyInfo>
  <httpProxyHost>proxyhost.domain.com</proxyHost>
  <httpProxyPort>80</proxyPort>
</proxyInfo>
```

You can also specify a proxy for HTTPS communication using the `<httpsProxyInfo>` element, which has the following sub-elements:

```
<httpsProxyInfo>
  <httpsProxyHost>proxyhost.domain.com</proxyHost>
  <httpsProxyPort>80</proxyPort>
</httpsProxyInfo>
```

PORTLET RENDERING OVERVIEW

Portlet rendering with URL Services is similar to authentication in that the remote application is treated as a black box. The authentication overview describes how a URL is constructed and an HTTP request is submitted to authenticate a user with the remote application. . Rendering content is very similar. A URL is submitted to the remote application and the content generated by the remote application is captured for display in the form of a portlet.

However, content from a web application cannot be included in a portlet because a portlet represents a fragment of a page whereas the content returned by the remote application is formatted as an entire page. Also, you may not want to display all the content returned by the remote application. You may be interested in only a fraction of the page. To resolve this problem, the content is filtered before it is included in the portlet.

URL Services currently provides two filters you can use, one for HTML content and one for XML. The URL Services framework also includes a simple interface you can implement to create your own filters if the basic ones do not meet your needs.

LOGICAL RENDERING SEQUENCE

The following figure represents the logical sequence of events when rendering a portlet using URL Services:

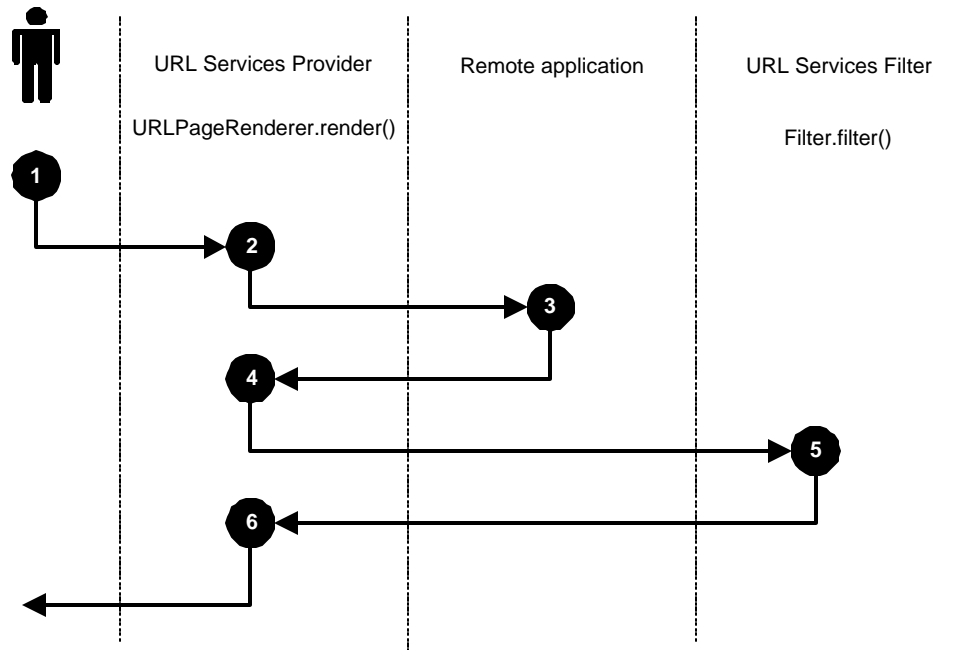


Figure 6. URL Services Logical Rendering Sequence

1. A user requests a portal page containing a portlet from a URL Services provider. If this is the first time the user accesses a portlet from this provider in the current portlet session, the portal calls the `initSession` method first to establish a provider session and authenticate the user with the remote application.
2. The portlet renders the portlet header and then constructs a URL to retrieve the content from the remote application. Any cookies returned during the authentication sequence are sent with the request so the remote application can re-attach to the user's session in the remote application.
3. The remote application re-attaches to the remote application session that is established during the authentication sequence and proceeds to generate the content. The content is returned to the URL Services provider as an HTTP response in the same format as returned to a regular web browser.
4. The `URLPageRenderer` receives the content from the remote application and converts any relative hyperlinks to absolute links. This is necessary because the user's browser reads the completed page from the portal, not from the remote application. This means any links in the content must be absolute links back to the remote application. Once the links are converted, the content is streamed through the specified filter.
5. The filter processes the content according to the rules of the filter. The minimum requirement for filtering HTML content is to remove all the HTML headers along with the opening and closing `<html>` and `<body>` tags so the content can be included in a table cell. This is a requirement for all portlet content regardless of the source.
6. The `URLPageRenderer` streams the content back to the portal and renders the portlet footer.

SAMPLE IMPLEMENTATION

As with authentication, there is no code to review for URL Services integration. All you need to do is declare the URL and a few other pieces of information in the provider definition file. A sample renderer specification is included in the following:

```

<renderer class="oracle.portal.provider.v1.RenderManager">
  <showPage class="oracle.portal.provider.v1.http.URLPageRenderer ">
    <accessControl>registered</accessControl>
  </showPage>
</renderer>
  
```

```

<pageUrl>http://www.microsoft.com/windows2000/upgrade/default.asp</pageUrl>
<filterType>text/html</filterType>
<filter class="oracle.portal.provider.v1.http.HtmlFilter">
  <headerTrimTag>BODY</headerTrimTag>
  <footerTrimTag>/BODY</footerTrimTag>
  <baseHref>http://www.microsoft.com/</baseHref>
  <secBaseHref>http://www.microsoft.com/windows2000/upgrade/</secBaseHref>
  <convertTarget>true</convertTarget>
  <useAuthLinks>false</useAuthLinks>
</filter>
</showPage>
</render>

```

This sample accesses an Active Server Page located on Microsoft's web site to demonstrate how you can use URL Services to integrate content from any source, independent of the technology used to create it. The meaning of the tags is described in the following:

- `<accessControl>` is an optional tag and indicates whether this portlet is publicly available or available only to registered users. There are two possible values: `public` and `registered`.
- `<pageUrl>` is an optional tag that specifies the URL to be used for getting the contents of a portlet. This can be any URL that can be accessed from a browser. The URL must contain the HTTP prefix. Also, if the URL contains "&", it must be replaced by the escape characters "&". This prevents provider.xml from becoming ill-formed.
- `<filterType>` is an optional tag that indicates the kind of filter that needs to be used for filtration. There are three possible values:
 - `none` - this is the default
 - `text/html`
 - `text/xml`
- `<filter>` is an optional tag which specifies the filter to be applied on rendered contents. It has one attribute and multiple tags. The tags used depend on the filter used. In this case, the HTML filter is used. This is declared using the class attribute of the filter element. You can specify any Java class that implements the `oracle.portal.provider.v1.ContentFilter` interface.
 - `<headerTrimTag>` is an optional tag. All content up to and including the tag or tag sequence specified as a value for this element are removed from the beginning of the HTML content. This tag takes values without "<" or ">" signs to eliminate the need to use escape characters. If you need to trim the HTML headers and opening `<BODY>` tag, specify the value "BODY".
 - `<footerTrimTag>` is an optional tag. All content including and following the tag or tag sequence specified as a value for this element is removed from the end of the HTML content. This tag takes values without "<" or ">" signs to eliminate the need to use escape characters. If you need to trim the closing `</BODY>` and `</HTML>` tags from the end of the content, specify the value "BODY".
 - `<baseHref>` is an optional tag which specifies the base HREF for the HTML page. This is used for conversion of relative URLs to absolute URLs.
 - `<secBaseHref>` is an optional tag which specifies a secondary base HREF for the HTML page.
 - `<convertTarget>` is an optional tag which specifies whether hyperlinks on the page need to be modified so they open a new browser window rather than overwriting an existing portal page. This is a Boolean value.
 - `<useAuthLinks>` is an optional tag that indicates whether the hyperlinks within the portlet content need

authenticated access. This is a Boolean value. The default for this tag is false.

- `<redirectUrlFieldName>` is an optional tag and indicates the parameter name recognized by the external application's Login URL as a URL that should be redirected to after a successful authentication. This parameter is used only if the Portlet indicates it has authenticated hyperlinks by setting the `<useAuthLinks>` tag to true. If the portlet does not need authenticated hyperlinks it is not used.

IMPLEMENTING SECURE DEEP LINKS

Deep links are hyperlinks in portlets that when clicked take you from Oracle9iAS Portal to another web application. This allows you to perform a task in the web application. For example, if you have a portlet that displays summarized information about quarterly sales from your accounting applications, clicking a link in the portlet may take you to your sales application where you can review more detailed information about the sales. Generally, applications that are accessed using deep links include a link that allows you to return to the portal page you started from. This may replace the “home” link that the application displays if you had accessed the application directly, or it may be an additional link that is only displayed when the application is accessed using your portal.

DEEP LINKS WITH A SHARED SESSION

When you integrate an application with Oracle9iAS Portal by creating a partner application provider or an external application provider, the application and the provider normally maintain distinct user sessions. This means that if you log in to the application you establish one user session and when you access a provider to view a portlet, the provider establishes another session with the application. Although both sessions are for the same user, they are maintained separately. Information from one cannot be accessed by the other. In certain circumstances, you may want your application and provider to share the same session as the actual application. This is referred to as session sharing.

WHY SHARE SESSION INFORMATION?

The importance of session sharing depends on the architecture of the application you are integrating into Oracle9iAS Portal. If your application relies on an in-memory session store (such as an HTTP Session) and your portlets include deep links, session sharing is probably important to you.

For example, you have an application that includes a shopping cart and a portlet that displays the current contents of the cart. The application maintains the contents of the cart in memory. You do the following:

1. Display a portal page containing the cart portlet. Click a deep link to add an item to the cart.
2. This takes you into your application where you search for an item to add to your cart.
3. You search for another item and add it to the cart.
4. You return to the portal by clicking a link that you added to your application. The link returns you to the portal page you started from.
5. When the portlet is displayed, the items you added to your cart are displayed in the portlet.

If the provider and the application could not share sessions, you would have to modify your application so that it saved the state of the cart in a persistent store such as a database. While this seems relatively trivial, locating all the locations where the cart could possibly be updated and finding all the other pieces of information that need to be shared could be a huge task that results in significant modifications to the original application.

Session sharing allows you to integrate the application and get the behavior you need without making significant changes to the logic of your application. You should consider including a way for your users to navigate back to the portal page they started from (possibly replacing “home” links when the page is called from a portal), but this is not usually a significant change.

HOW DOES SESSION SHARING WORK?

Web enabled applications usually track sessions using “cookies”. Cookies are small containers of information that can be created by a web application and stored for varying lengths of time by your web browser. Web applications

create session cookies to store the keys that are used to identify entries in a session store. The session store may be maintained in memory or it may be persistent. These cookies are usually temporary, which means they are only stored while your browser is running. When you exit from your browser, the cookies are destroyed.

The key to session sharing is cookie sharing. If you can create a single session cookie and make sure that session cookie is sent with all requests to both the application and the provider then they will share the same session. The application and provider share the same application session because it is the cookie that is used to identify and re-attach to the session before processing the request.

This appears straightforward, but due to the security features of cookies it only works in limited circumstances. The requirements for session sharing are described in the following section.

REQUIREMENTS FOR SESSION SHARING

Session sharing is only possible under a very specific set of circumstances. The requirements for session sharing include:

- application is web enabled and uses cookies to track sessions
- cookies created to track the sessions are identical
- application session cookies can be scoped to a common domain
- provider and application reside on the same listener (or do not rely on an in-memory session)
- application is not load balanced (or does not rely on an in-memory session store)
- application is a partner application or an external application
- provider cookies are forwarded to the user's browser
- each application is associated with a uniquely named servlet zone

WEB ENABLED APPLICATION USING COOKIES TO TRACK SESSIONS

Web enabled applications usually track sessions using “cookies”. Cookies are small containers of information that can be created by a web application and stored for varying lengths of time by your web browser. Web applications create session cookies to store the keys that are used to identify entries in a session store. The session store may be maintained in memory or it may be persistent. These cookies are usually temporary, which means they are only stored while your browser is running. When you exit from your browser, the cookies are destroyed.

Cookies must be used to track the sessions. The current release of Oracle9iAS Portal does not allow you to track application sessions using URL parameters, the only option available is cookies.

COOKIES CREATED TO TRACK SESSIONS ARE IDENTICAL

The cookies that your provider creates to track the provider's application session are indistinguishable from those the application normally creates when accessed directly. This is important because session sharing requires the application and provider to share the same set of cookies.

APPLICATION SESSION COOKIES SCOPED TO COMMON DOMAIN

When cookies are created they are usually scoped so that the web browser only sends them to the specific HTTP listener that is running the application that created them. This is a security feature intended to prevent cookies from being sent to applications that should not “see” them.

However, this is also the factor that makes session sharing complex because the application's session is established between the web browser and the application's listener, and the provider's session is established between the portal and the application's listener. Since these are likely to be different machines, the session cookies for each machine are not visible to each other. To make them visible you must set the cookie domain to a value that is common to both.

For example, if the application is hosted at `app.mydomain.com` and the portal is hosted at `portal.mydomain.com`, the session cookie's domain must be set to `.mydomain.com`. This is the most specific

domain that is common to both hosts. When defining a cookie domain, the specification requires that the domain contain at least two dots. This ensures that a cookie cannot be created with the domain “.com”

PROVIDER AND APPLICATION RESIDE ON SAME LISTENER

If the application and provider need to share an in-memory session, such as a Servlet session, they must be hosted on the same machine and in the same servlet zone so that they are running in a common memory space. In this case, within the same JVM.

APPLICATIONS USING IN-MEMORY SESSION STORAGE CANNOT BE LOAD-BALANCED

If the application must share an in-memory session, it cannot be load balanced. This is because load balancing hardware either routes requests to the server with the lowest load, or if it is configured to be “sticky”, uses the IP address of HTTP requests. This is to make sure that all requests from the same IP address are forwarded to the same server. Since requests originate from three possible sources (your web browser, the portal middle-tier, or portal database) that most likely have different IP addresses, load balancing in this way results in the requests going to different servers.

PARTNER APPLICATIONS OR EXTERNAL APPLICATIONS ONLY

The application must be integrated with the Oracle9iAS Portal as a partner application or an external application. Applications integrated using URL Services cannot share sessions because the session information is maintained by the URL Services framework. It is never sent to the portal. Consequently, the portal can never send the information to the web browser. Even if the session information could be forwarded to the portal, the remote application must obey the other rules, such as scoping the cookie domain to a common domain.

PROVIDER COOKIES ARE FORWARDED TO THE USER'S WEB BROWSER

The final requirement to enable session sharing lies in the registration of the Provider with Oracle9iAS Portal. The requirements for registering a provider to support session sharing differ depending on the release of Oracle9iAS Portal.

RELEASE 3.0.6, 3.0.7 AND 3.0.7 PATCH 1

In releases 3.0.6, 3.0.7, and 3.0.7 patch 1 of Oracle9iAS Portal, all browser cookies are automatically forwarded to the provider and cookies created by a provider are forwarded back to the browser

This means an existing application session cookie, which is held by the web browser, is forwarded to the provider whenever the portal contacts the provider. The provider can determine the existence of a valid application session cookie in calls to `initSession` and use that session instead of establishing a new session. The same is also true if the provider is the first to establish an application session. The session cookie established during `initSession` is forwarded to the web browser and subsequently sent to the application when you first access the application. The application should detect the existence of the session cookie and use that cookie instead of establishing a new session.

RELEASE 3.0.8 AND LATER

In Oracle9iAS Portal release 3.0.8 and later, the handling of cookies was modified. To enable session sharing there is a provider registration setting that indicates to the portal that the provider is in the same cookie domain as the portal and as such should receive all the cookies that the portal receives from the browser. The check-box for this setting is located under the URL for a web provider. The prompt is “Web provider in same cookie domain as the portal”. Checking this box during provider registration causes provider cookies to be forwarded to the web browser and cookies from the web browser to be forwarded to the provider

Note: The `initSession` method should always be coded to check for the existence of a current session. This is necessary because the user may have logged into the application already in which case the application session cookie created by the application is forwarded to the provider. `initSession` may also be called multiple times, even if the login frequency is set to “Once per user session”. This can occur if the provider session is close to expiration or has expired when a portlet is requested.

EACH APPLICATION IS ASSOCIATED WITH A UNIQUELY NAMED SERVLET ZONE

Each application and provider that uses HTTP Sessions in any way must be allocated to a uniquely named JServ

servlet zone. This is necessary because the HTTP Session is tracked using a cookie and changing the scope of this cookie to allow the HTTP Session to be shared makes the cookie visible to multiple HTTP listeners. This means that a single request received by a listener may have more than one JServ session cookie associated with it. The cookie that originated from that listener and cookies from other listeners that fall in the same scoped domain. If multiple JServ session cookies are received by a JServ instance, JServ attempts to use the first cookie it finds, which is probably not the one it needs. When this happens, the application or provider behaves as if the session has been terminated.

This can be avoided by naming all servlet zones differently, even if they are on different physical hosts. Naming the servlet zones uniquely prevents problems between JServ session cookies because the cookie name includes the servlet zone name.

DEEP LINKS WITHOUT SESSION SHARING

Although it may be desirable to share a single session when accessing an application using deep links, it is not necessary, and in some cases, not possible. For example, it is not possible to share sessions if a cookie cannot be scoped to a common domain. This does not mean that you cannot implement deep links, you can. You just cannot share session information between the provider and the application.

DEEP LINKS IN EXTERNAL APPLICATIONS OR URL SERVICES

If you have a remote application (for example, on the Internet) you can create portlets that contain links to specific application functionality and defer authentication until the deep link is accessed. This is done by wrapping the link in a call to the Single Sign-On Server. When a user clicks the link, the Single Sign-On Server is contacted and authenticates the user with the remote application using information from the password store and external application registry. After authenticating the user with the remote application and establishing a remote application session cookie with the browser, the Single Sign-On Server redirects the user's browser to the remote application page using the wrapped link.

This approach works with most web applications because they already have built-in facilities that support authentication of a user followed by a redirect to a specific application page. For example, if a user creates a bookmark for a link that is within a password protected web application, the application detects that the user does not have a session and displays the login screen instead of the final destination. As soon as the user enters a valid username and password, the user is taken to the page identified by the bookmark. Applications that behave in this way usually have several hidden HTML form fields in their login page. Typically they include:

- username
- password
- redirect URL
- additional application specific fields

The redirect URL field is used by the login page to redirect the user's web browser to the page the user originally requested if authentication is successful. This same infrastructure is leveraged by the Single Sign-On Server to automatically authenticate you when you click an authenticated link.

WRAPPING AUTHENTICATED LINKS

The following sample code below is taken from the external application sample. This code fragment demonstrates how to construct or wrap a URL. The approach is straightforward. All you need to do is append two parameters to the Single Sign-On Server's URL. The first is the name of the redirect URL field described in the preceding section, and the second is the URL of the page you want to access.

```
String loginURL=mPR.getLoginServerURL();
String str1 = PortletRendererUtil.parameterizeLink(loginURL,"p_url_name=FlightDest");

server.append("/servlets/flights?Flightaction=details&FlightId=" + id);
hrefValue = PortletRendererUtil.parameterizeLink(str1, "p_url_value=" +
URLLEncoder.encode(server.toString()));
```

When the Single Sign-On Server is contacted, it takes the preceding parameters and constructs a new URL to the remote application's login page, including the username, password and redirect URL. The remote application's login page includes code that looks something like the following. This is taken from the external application sample included with the PDK-Java.

```
public void performLogin(String username, String password, String dest)
    throws Exception
{
    StringBuffer server = new StringBuffer();

    Cookie cookie = null;
    for (int i=0; i<VALIDUSERNAMES.length; i++)
    {
        if (username.equalsIgnoreCase(VALIDUSERNAMES[i]) &&
            password.equalsIgnoreCase(VALIDPASSWORDS[i]))
        {
            cookie = new Cookie(COOKIENAME,username);
            cookie.setPath("/servlets/flights");
            break;
        }
    }
    if (cookie != null)
    {
        mResponse.addCookie(cookie);
    }
    mResponse.sendRedirect(dest);
}
```

authenticate the user.

create application session cookie.

add the new cookie to the HTTP response.

redirect the browser to the requested URL - in this case, the wrapped URL.

AUTHENTICATED LINKS IN URL SERVICES

URL Services uses the same approach to wrap authenticated links. The main difference is that the parameters (redirect field name and URL) are specified declaratively in the provider registry file and the filters provided with URL Services automatically wrap the links for you. When the source page is filtered by URL Services, the links the filter finds are wrapped using code similar to the preceding sample. You only need to know the name of the parameter that is used for the redirect link in the application's remote login page. You can usually determine this by opening the login page for the application and reviewing the source. There is usually one or more hidden fields in a form, one for the username, one for password, one for the redirection URL, and possibly others specific to the application.

SUMMARY

You should now be able to integrate your applications with Oracle9iAS Portal using the most appropriate integration approach. The preferred method of integration is to create a partner application provider that tightly integrates your application with Oracle9iAS Portal and Oracle9iAS Single Sign-On Server. External application providers are intended for applications that support their own authentication mechanisms and user repositories, and that cannot be easily converted to partner applications. URL Services provides a fast way of integrating a wide variety of web based applications, regardless of their location on the Internet or the technology used in their implementation. Finally, the key to seamless integration between your portal and application for all approaches is deep links. Deep links in your portlets allow users to access the applications they need from a central location without the concerns of multiple usernames and passwords.

Chris Broadbent, Oracle Corporation