Oracle® Fusion Middleware

Developer's Guide for Remote Intradoc Client (RIDC)

11*g* Release 1 (11.1.1)

E16819-01

May 2010

The *Oracle Fusion Middleware Developer's Guide for Remote Intradoc Client (RIDC)* describes Remote Intradoc Client and provides initialization and usage information. RIDC provides a thin communication API for communication with Oracle Content Server. This API removes data abstractions to the content server while still providing a wrapper to handle connection pooling, security, and protocol specifics.

This document covers the following topics:

1 What's New

Remote Intradoc Client (RIDC) 11*g* Release 1 (11.1.1) includes the following new features and enhancements.

Support for JAX_WS

RIDC supports WebServices/JAX-WS. Initialization and configuration examples for JAX-WS are included this document.

2 Initialization

RIDC supports three protocols: Intradoc, HTTP, and WebServices/JAX-WS.

Intradoc: The Intradoc protocol communicates with Oracle Content Server over the Intradoc socket port (typically 4444). This protocol does not perform password validation and thus requires a trusted connection between the client and Oracle Content Server. Clients that use this protocol are expected to perform any required authentication. Intradoc communication can also be configured to run over SSL.

HTTP: RIDC communicates with the web server for Oracle Content Server using the Apache HttpClient package. Unlike Intradoc, this protocol requires authentication credentials for each request.

Refer to Apache's Jakarta HttpClient documentation for more information: http://hc.apache.org/httpclient-3.x

JAX-WS: The JAX-WS protocol is only supported in Oracle Content Server 11*g* with the content server running in Oracle WebLogic Server. This protocol requires that additional initialization steps be performed on the server as well as the client.

- You will need the ECM Client libraries in your class path, which are shipped with the RIDC distribution.
- You will need to configure security WLS for the UCM web services. This includes:
  - Setting up the policy for the login service
  - Creating a new keystore file (or adding credentials to your existing keystore) which will be used by both the server and the client
  - Setting up wallet by adding the credentials
- The client requires the following:
  - The JPS configuration file
  - The keystore
  - The wallet from the server

Refer to the *Oracle Fusion Middleware Services Reference Guide for Universal Content Management* for more information on configuring the server and client for web services.

This table shows the URL formats that are supported:

URL	Description
http://*host*/cs/idcplg	URL to Oracle Content Server's CGI path.
https://*host*/cs/idcplg	Uses SSL over HTTP; requires extra configuration to load the SSL certificates.
idc://*host*:4444	Uses the Intradoc port, only requires hostname and port.
idcs://*host*:4444	Uses SSL over the Intradoc port; requires extra configuration to load the SSL certificates.
http://*wlsserver*:7044/idcnativews	Uses the JAX-WS protocol to connect to Oracle Content Server.

This example code initializes RIDC for an Intradoc connection:

```
// create the manager
IdcClientManager manager = new IdcClientManager();

// build a client that will communicate using the intradoc protocol
IdcClient idcClient = manager.createClient("idc://localhost:4444");
```

This example code initializes an HTTP connection (the only difference from an Intradoc connection is the URL):

```
// create the manager
IdcClientManager manager = new IdcClientManager();

// build a client that will communicate using the HTTP protocol
IdcClient idcClient = manager.createClient("http://localhost/idc/idcplg");
```

This example code initializes a JAX-WS client. These two web services are exposed by Oracle Content Server: the `login` service and the `request` service. You will need the web context root that these web services use. By default, this is `idcnativews`.

```
// create the manager
IdcClientManager manager = new IdcClientManager();

// build a client that will communicate using the JAXWS protocol
IdcClient idcClient = manager.createClient("http://wlsserver:7044/idcnativews");
```

3 Client Configuration

Configuration of the clients can be done after they are created. Configuration parameters include setting the socket timeouts, connection pool size, and so on. The configuration is specific to the protocol; if you cast the IdcClient object to the specific type, then you can retrieve the protocol configuration object for that type.

This example code sets the socket timeout and wait time for Intradoc connections:

```
// build a client as cast to specific type
IntradocClient idcClient = (IntradocClient)manager.createClient
   ("http://host/cs/idcplg");

// get the config object and set properties
idcClient.getConfig ().setSocketTimeout (30000); // 30 seconds
idcClient.getConfig ().setConnectionSize (20);   // 20 connections
```

These JAX-WS specific configurations can be set after you have created the client:

```
// build a client as a cast for jaxws type
JaxWSClient jaxwsClient = (JaxWSClient) manager.createClient(
   ("http://wlsserver:7044/idcnativews");
JaxWSClientConfig jaxwsConfig = jaxwsClient.getConfig();
```

You can set the name of the Oracle Content Server instance that you want to connect to. By default, this is `/cs/`, which is the default web context for a UCM installation. If the server web context is different than the default, then you may set it by editing the property. This example code sets your web context root:

```
// set the property
jaxwsConfig.setServerInstanceName("/mywebcontext/");
```

A JPS configuration file is required for most policies such SAML and/or Message Token. This example code sets the JPS configuration file location:

```
jaxwsConfig.setJpsConfigFile("/my/path/to/the/jps-config.xml");
```

This example code sets the security policy:

```
jaxwsConfig.setClientSecurityPolicy
```

```
   ("policy:oracle/wss11_username_token_with_message_protection_client_policy");
```

RIDC uses the default values for the installed web services. If, for some reason, the web services have been modified and do not conform to the default URI/URLs, you may need to modify the default values. This example code changes the `login` and `request` service URLs:

```
// login port wsdl url
jaxwsConfig.setLoginServiceWSDLUrl(new URL
   ("http://server:7044/webservices/loginPort?WSDL"));

//request port wsdl url
jaxwsConfig.setRequestServiceWSDLUrl(new URL
   ("http://server:7044/anotherservice/myrequestport?WSDL"));
```

The default streaming chunk size is set to 8192. This example code changes the streaming chunk size:

```
jaxwsConfig.setStreamingChunkSize(8190);
```

4 SSL Configuration

RIDC allows Secure Socket Layer (SSL) communication with Oracle Content Server using the Intradoc communication protocol.

> **Note:** You must install and enable the Security Providers component on the Oracle Content Server instance that you wish to access and configure the content server for SSL communication.

An example of using the IDC protocol over a Secure Socket (SSL):

```
// build a secure IDC client as cast to specific type
IntradocClient idcClient = (IntradocClient
   manager.createClient("idcs://localhost:54444");

// set the SSL socket options
config.setKeystoreFile("ketstore/client_keystore");  //location of keystore file
config.setKeystorePassword ("password");      // keystore password
config.setKeystoreAlias("SecureClient");  //keystore alias
config.setKeystoreAliasPassword("password");  //password for keystore alias
```

See "SSL Communication With Oracle Content Server" on page 10 for more information.

5 Authentication

All calls to RIDC require some user identity. Optionally, this identity can be accompanied by credentials as required by the protocol. The user identity is represented by the `IdcContext` object; once created, it can be reused for all subsequent calls. To create an identity, you pass in the username and optionally some credentials:

```
//create a simple identity with no password (for idc:// urls)
IdcContext userContext = new IdcContext("sysadmin");

// create an identity with a password
```

```
IdcContext userPasswordContext = new IdcContext("sysadmin", "idc");
```

For Intradoc URLs, you do not need any credentials as the request is trusted between Oracle Content Server and the client.

For HTTP and JAX-WS URLs, the context needs credentials.

For HTTP URLs, the credentials can be a simple password or anything that the HttpClient package supports.

For JAX-WS URLs, the requirement for credentials will be dependent on the service policy that the web service is configured to use by the server.

6 Usage

To invoke a service, use the `ServiceRequest` object, which can be obtained from the client. Creating a new request will also create a new binder and set the service name in the binder, along with any other default parameters. From that point, you can populate the binder as needed for the request.

This example code executes a service request and gets back a data binder of the results:

```
// get the binder
DataBinder binder = idcClient.createBinder();

// populate the binder with the parameters
binder.putLocal ("IdcService", "GET_SEARCH_RESULTS");
binder.putLocal ("QueryText", "");
binder.putLocal ("ResultCount", "20");

// execute the request
ServiceResponse response = idcClient.sendRequest (userContext, binder);

// get the binder
DataBinder serverBinder = response.getResponseAsBinder ();
```

The `ServiceResponse` contains the response from Oracle Content Server. From the response, you can access the stream from the content server directly or you can parse it into a DataBinder and query the results.

This example code takes a `ServiceResponse` and get the search results, printing out the title and author values:

```
// get the binder
DataBinder binder = response.getResponseAsBinder ();
DataResultSet resultSet = binder.getResultSet ("SearchResults");

// loop over the results
for (DataObject dataObject : resultSet.getRows ()) {
    System.out.println ("Title is: " + dataObject.get ("dDocTitle"));
    System.out.println ("Author is: " + dataObject.get ("dDocAuthor"));
}
```

7 Connection Handling

The RIDC client pools connections, meaning that the caller of the code must close resources when done with a response. This is done automatically when calling `getResponseAsBinder` or by calling `close` on the stream returned via a call to `getResponseStream`. If a user does not want to examine the results, `close` must

still be called, either by getting the stream and closing it directly or by calling `close` on the `ServiceResponse` object.

To close via the response as binder:

```
// execute the request
ServiceResponse response = idcClient.sendRequest (userContext, binder);

// get a binder closes the response automatically
response.getResponseAsBinder ();
```

To close via the stream:

```
// execute the request
ServiceResponse response = idcClient.sendRequest (userContext, binder);

// get the result stream and read it
InputStream stream = response.getResponseStream ();
int read = 0;
while ((read = stream.read ()) != -1)
{
}

//close the stream
stream.close ();
```

To close via the `close` method on the `ServiceResponse`:

```
// execute the request
ServiceResponse response = idcClient.sendRequest (userContext, binder);

// close the response (which closes the stream directly)
response.close ();
```

Pooling Options

The `IdcClientConfig#getConnectionPool` property determines how RIDC will handle pooling of connections. There are two options, `simple` and `pool`.

- `Pool` is the default, and it means to use an internal pool which allows a configurable number of active connections at a time (configurable via the `IdcClientConfig#getConnectionSize property`), with the default active size set to 20.
- `Simple` does not enforce a connection maximum and rather lets every connection proceed without blocking.

A different pool implementation can be registered via the `IdcClientManager#getConnectionPoolManager()#registerPool()` method, which maps a name to an implementation of the `ConnectionPool` interface. The name can then be used in the `IdcClientConfig` object to select that pool for a particular client.

8 Streams

Streams are sent to Oracle Content Server via the `TransferStream` interface. This interface wraps the actual stream with metadata about the stream (length, name, and so on).

This example code performs a check-in to the content server:

```
// create request
DataBinder binder = idcClient.createBinder();
binder.putLocal ("IdcService", "CHECKIN_UNIVERSAL");

// get the binder
binder.putLocal ("dDocTitle", "Test File");
binder.putLocal ("dDocName", "test-checkin-6");
binder.putLocal ("dDocType", "ADACCT");
binder.putLocal ("dSecurityGroup", "Public");

// add a file
binder.addFile ("primaryFile", new File ("test.doc"));

// checkin the file
idcClient.sendRequest (userContext, binder);
```

Receiving a stream from Oracle Content Server can be done via the `ServiceResponse` object; the response is not converted into a DataBinder unless specifically requested. If you just want the raw HDA data, you can get that directly, along with converting the response to a String or DataBinder:

```
// create request
DataBinder binder = idcClient.createBinder ();

// execute the service
ServiceResponse response = idcClient.sendRequest (userContext, binder);

// get the response stream
InputStream stream = response.getResponseStream ();

// get the response as a string
String responseString = response.getResponseAsString ();

// parse into data binder
DataBinder dataBinder = response.getResponseAsBinder ();
```

9 JSP/JSPX

The RIDC objects all follow the standard Java Collection paradigms which makes them extremely easy to consume from a JSP/JSPX page. Assume the `ServerResponse` object (used in the previous example) was available in the `HttpServletRequest` in an attribute called `idcResponse`. This example JSPX code will iterate over the response and create a small table of data:

```
<table>
  <tr>
    <th>Title</th>
    <th>Author</th>
    <th>Release Date</th>
  </tr>
<c:forEach var="row" items="${idcResponse.dataBinder.SearchResults.rows}">
  <tr>
    <td>${row.dDocTitle}</td>
    <td>${row.dDocAuthor}</td>
    <td>${row.dInDate}</td>
   </tr>
</c:forEach>
</table>
```

10 Reusable Binders

The binders can be reused among multiple requests. A binder from one request can be sent in to another request. For example, this example code pages the search results by reusing the same binder for multiple calls to Oracle Content Server:

```
// create the user context
IdcContext idcContext = new IdcContext ("sysadmin", "idc");

// build the search request binder
DataBinder binder = idcClient.createBinder();
binder.putLocal("IdcService", "GET_SEARCH_RESULTS");
binder.putLocal("QueryText", "");
binder.putLocal("ResultCount", "20");

// send the initial request
ServiceResponse response = idcClient.sendRequest (binder, idcContext);
DataBinder responseBinder = response.getResponseAsBinder();

// get the next page
binder.putLocal("StartRow", "21");
response = idcConnection.executeRequest(idcContext, binder);
responseBinder = response.getResponseAsBinder();

// get the next page
binder.putLocal("StartRow", "41");
response = idcConnection.executeRequest(binder, idcContext);
responseBinder = response.getResponseAsBinder();
```

11 User Security

Oracle Content Server has several security models that are controlled by settings on the content server. To resolve if a particular user has access to a document, three things are needed: the user's permission controls, the document's permission controls, and the content server security environment settings.

It is assumed that the application program calling the `UserSecurity` module will fetch documents and the `DOC_INFO` metadata (in the document's binder, typically the result of a search) as some superuser and cache this information. When the application needs to know if a particular user has access to the document, a call is made to Oracle Content Server as that user to fetch that user's permissions. Once the user's permission controls are known, then they can be matched to the information in the document's metadata to resolve the access level for that user. The available access levels are:

- READ
- READ/WRITE
- READ/WRITE/DELETE
- READ/WRITE/DELETE/ADMIN

It is preferable to reduce the number of calls to Oracle Content Server (using a cache) and to provide a default implementation for matching the user's permissions information with the document's permission information. One limitation is that Oracle Content Server controls which types of security are used in some server environment properties: `UseAccounts=true` and `UseCollaboration=true`.

The user security convenience is accessed through the `IUserSecurityCache` interface. There three concrete classes which implement the optional Oracle Content Server security.

- The `UserSecurityGroupsCache` class simply keeps a cache of user permissions and will match documents considering only Security Group information.
- The `UserSGAccountsCache` class adds a resolver to also consider Account information if the content server has the `UseAccounts=true` setting.
- The `UserSGAcctAclCache` class adds a resolver to also consider ACL permissions if `UseCollaboration=true`.

The IAccessResolver interface allows the addition of classes that can participate in the resolution of the access levels for a document.

Example code:

```
IdcClientManager m_clientManager = new IdcClientManager ();
IdcClient m_client = m_clientManager.createClient
   ("http://localhost/scs/idcplg");

//RIDC superuser context
IdcContext m_superuser = new IdcContext("sysadmin", "idc");

//Examples of the three concrete cache classes
IUserSecurityCache m_SGCache = new UserSecurityGroupsCache
   (m_client, 20, 1000);
IUserSecurityCache m_SGAcctCache = new UserSGAccountsCache
   (m_client, 20, 1000, 20000);
IUserSecurityCache m_SGAcctAclCache = new UserSGAcctAclCache
   (m_client, 20, 1000, 20000, m_superuser);

//Example test
testDocPermission () {
   DataBinder m_doc1 = getDataBinder ("TEST");

   //Get the document information (typically in the first row of DOC_INFO)
   DataObject docInfo = m_doc1.getResultSet ("DOC_INFO").getRows ().get (0);

   //Get the cache id for this user
      //Important: this makes a live call to content server
      //to get the user ID for "Acme1")
      //CacheId acme1 = m_SGAcctAclCache.getCacheIdForUser
      //   (new IdcContext("Acme1", "idc"));
      //You may want to include this:

   IdcContext context = new IdcContext("Acme1", "idc");
   CacheId acme1 = new CacheId (context.getUser (), context);

   //Get the access level for this document by this user
int access = m_SGAcctAclCache.getAccessLevelForDocument (acme1, docInfo);
   }

//Example code to get a Document's DOC_INFO databinder
DataBinder getDataBinder (String dDocName) throws IdcClientException {
   DataBinder dataBinder = m_client.createBinder ();
   dataBinder.putLocal ("IdcService", "DOC_INFO_BY_NAME");
   dataBinder.putLocal ("dDocName", dDocName);
```

```
   ServiceResponse response = m_client.sendRequest
      (m_superuser, dataBinder);
   return response.getResponseAsBinder ();
}

//Example code to create a DataObject
DataObject dataObject = m_client.getDataFactory ().createDataObject ();
dataObject.put ("dSecurityGroup", "public");
dataObject.put ("dDocAccount", "Eng/Acme");
```

Internally, these fields from the document are examined during `getAccessLevelForDocument()`.

- For the AccessResolverSecurityGroups class: `dSecurityGroup`
- For the AccessResolverAccounts class: `dDocAccount`
- For the AccessResolverSecurityGroups class: `xClbraUserList` and `xClbraAliasList`

The above `IAccessResolver` classes determine if they should participate based on cached information from the content server, if they do participate, the access levels are AND-ed together.

12 SSL Communication With Oracle Content Server

RIDC allows Secure Socket Layer (SSL) communication with Oracle Content Server. This section provides basic information on SSL communication including how to set up a sample implementation for testing purposes. This sample implementation, uses a JDK utility to create self-signed keypair and certificates. Oracle does not provide signed certificates. For most implementations, you will want a certificate signed by a universally recognized Certificate Authority.

This section covers the following topics:

12.1 Installing and Enabling the Security Providers Component

You must have a valid KeyStore or TrustManager with signed trusted certificates on both the client and on the content server.

You must install and enable the SecurityProviders component on the Oracle Content Server instance you wish to access. This component is installed and enabled by default on Oracle Content Server 11*g*R1. On Oracle Content Server 10*g*R3, you need to install and enable the component manually. See the UCM installation documentation for instructions on installing and enabling components.

12.2 Configuring the Content Server for SSL

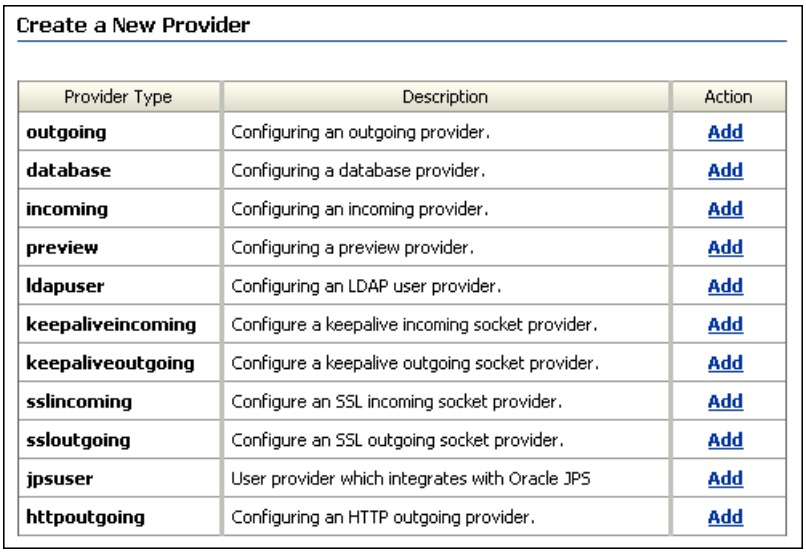
For SSL communication, the content server must be configured with a new incoming provider and the truststore/keystore information must be specified.

Setting Up a New Incoming Provider

You can set up a new keepalive incoming socket provider or a new SSL incoming socket provider. The setup steps for both are listed below. Using keepalive improves the performance of a session and is recommended for most implementations.

1. Log in to Oracle content Server as an administrator.
2. Click **Administration** and then **Providers**.

Figure 1 List of Providers in Oracle Content Server



Create a New Provider

Provider Type	Description	Action
outgoing	Configuring an outgoing provider.	Add
database	Configuring a database provider.	Add
incoming	Configuring an incoming provider.	Add
preview	Configuring a preview provider.	Add
ldapuser	Configuring an LDAP user provider.	Add
keepaliveincoming	Configure a keepalive incoming socket provider.	Add
keepaliveoutgoing	Configure a keepalive outgoing socket provider.	Add
sslincoming	Configure an SSL incoming socket provider.	Add
ssloutgoing	Configure an SSL outgoing socket provider.	Add
jpsuser	User provider which integrates with Oracle JPS	Add
httpoutgoing	Configuring an HTTP outgoing provider.	Add

3. Click **Add** for the sslincoming provider.

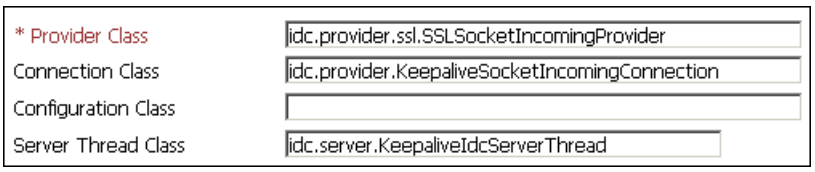
   The Add Incoming Provider page is displayed.
4. Enter a provider name and description.

   When your new provider is set up, a directory with your provider name is created as a subdirectory of the *IntradocDir*/data/providers directory.
5. Enter an open server port.
6. Enter configuration information for either a new SSL keepalive incoming socket provider or a new SSL incoming socket provider. The setup steps for both are listed below. Using keepalive improves the performance of a session and is recommended for most implementations.

   SSL keepalive incoming socket provider

   - **Provider Class:** idc.provider.ssl.SSLSocketIncomingProvider
   - **Connection Class:** idc.provider.KeepaliveSocketIncomingConnection
   - **Server Thread Class:** idc.server.KeepaliveIdcServerThread

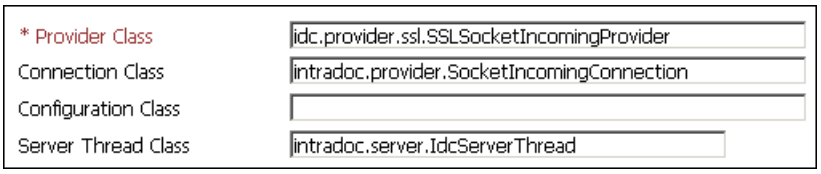
Figure 2 SSL Keepalive Provider



SSL incoming socket provider

- **Provider Class:** idc.provider.ssl.SSLSocketIncomingProvider
- **Connection Class:** intradoc.provider.SocketIncomingConnection
- **Server Thread Class:** intradoc.server.IdcServerThread

Figure 3 SSL Incoming Provider



7. Click **Add**.

After you have completed setting up a new incoming provider, you must also specify truststore and keystore information.

12.3 Certificate Signing Options

For most implementations, you will want a certificate signed by a universally recognized Certificate Authority. However, if you control both the client and server and only want to ensure that your transmissions are not intercepted, or if you are simply testing your implementation, you can create your own self-signed keypair and certificates using the JDK utility called keytool.

Sun's Key and Certificate Management Tool (keytool) is a key and certificate management utility that enables users to administer their own public/private key pairs and associated certificates for use in self-authentication. It is provided as part of Sun's JDK. Keytool is a command line utility. The executable is located in the `/bin` subdirectory.

Creating the Client and Server Keys

From a command line prompt, navigate to the *JDK-Home*/bin subdirectory and issue the `-genkey` command (this command generates a new key and takes several arguments). These arguments are used with this command:

Argument	Description
-alias	Alias of the key being created (this is the way a keystore knows which element in the file you are referring to when you perform operations on it).
-keyalg	Encryption algorithm to use for the key.
-keystore	Name of the binary output file for the keystore.
-dname	Distinguished name that will identify the key.

Argument	Description
-keypass	Password for the key that is being generated.
-storepass	Password used to control access to the keystore.

Generate a separate key pair for both the client and server. To do this, you will need to run the `-genkey` command twice, each time placing it into a separate keystore.

You will need to specify the alias, the algorithm to use, the keystore name, the distinguished name, and passwords for the keys and the keystore. This example uses RSA as the algorithm and 'idcidc' as the passwords.

Use these argument values for the client:

- -alias SecureClient
- -keyalg RSA
- -keystore client_keystore
- -dname "cn=SecureClient"
- -keypass idcidc
- -storepass idcidc

```
# keytool -genkey -alias SecureClient -keyalg RSA -keystore client_keystore -dname
"cn=SecureClient" -keypass idcidc -storepass idcidc
```

Use these argument values for the server:

- -alias SecureServer
- -keyalg RSA
- -keystore server_keystore
- -dname "cn=SecureServer"
- -keypass idcidc
- -storepass idcidc

```
# keytool -genkey -alias SecureClient -keyalg RSA -keystore client_keystore -dname
"cn=SecureClient" -keypass idcidc -storepass idcidc

# keytool -genkey -alias SecureServer -keyalg RSA -keystore server_keystore -dname
"cn=SecureServer" -keypass idcidc -storepass idcidc
```

Each of these commands will generate a key pair wrapped in a self-signed certificate and stored in a single-element certificate chain.

Self-Signing the Certificates

Keys are unusable unless they are signed. The keytool will self-sign them for you so that you can use the certificates for internal testing. However, these keys are not signed for general use.

From a command line prompt, issue the `-selfcert` command (this command self-signs your certificates and takes several arguments). Run the `-selfcert` command twice, once for the client and again for the server.

Use these argument values for the client:

- -alias SecureClient
- -keystore client_keystore
- -keypass idcidc
- -storepass idcidc

Use these argument values for the server:

- -alias SecureServer
- -keystore server_keystore
- -keypass idcidc
- -storepass idcid

Commands:

```
# keytool -selfcert -alias SecureClient -keystore client_keystore -keypass idcidc
-storepass idcidc

# keytool -selfcert -alias SecureServer -keystore server_keystore -keypass idcidc
-storepass idcidc
```

The certificate is now signed by its private and public key, resulting in a single-element certificate chain. This replaces the one that you generated previously.

Exporting the Certificates

After you have created the client and server keys, and self-signed the certificates, you now have two keypairs (public and private keys) in two certificates locked in two keystores. Since each application will need to have the public key of the other in order to encrypt and decrypt data, we need to place a copy of the public keys in the other keystore.

From a command line prompt, issue the `-export` command (this command exports your certificates and takes several arguments). Run the `-export` command twice, once for the client and again for the server. Use the `-file` argument to redirect the output to a file instead of the console.

Use these argument values for the client:

- -alias SecureClient
- -file client_cert
- -keystore client_keystore
- -storepass idcidc

Use these argument values for the server:

- -alias SecureServer
- -file server_cert
- -keystore server_keystore
- -storepass idcidc

Commands:

```
# keytool -export -alias SecureClient -file client_cert -keystore client_keystore
-storepass idcidc
```

(Certificate stored in file client_cert.)

```
# keytool -export -alias SecureServer -file server_cert -keystore server_keystore
-storepass idcidc
```
(Certificate stored in file server_cert.)

The certificate (containing the public key and signer information) has now been exported to a binary certificate file.

Importing the Certificates

The final step in setting up your self-signed certificates is to import the public certificates of each program into the keystore of the other. Keytool will present you with the details of the certificates you are requesting to be imported and provide a request confirmation.

From a command line prompt, issue the `-import` command (this command imports your certificates and takes several arguments). Run the `-import` command twice, once for the client and again for the server. Notice that the `-keystore` values are reversed.

Use these argument values for the client:

- -alias SecureClient
- -file client_cert
- -keystore server_keystore
- -storepass idcidc

Use these argument values for the server:

- -alias SecureServer
- -file server_cert
- -keystore client_keystore
- -storepass idcidc

Commands:

```
# keytool -import -alias SecureClient -file client_cert -keystore server_keystore
-storepass idcidc

Owner: CN=SecureClient
Issuer: CN=SecureClient
Serial number: 3c42e605
Valid from: Mon Jan 14 08:07:01 CST 2002 until: Sun Apr 14 09:07:01 CDT 2002
Certificate fingerprints:
  MD5: 17:51:83:84:36:D2:23:A2:8D:91:B7:14:84:93:3C:FF
  SHA1: 61:8F:00:E6:E7:4B:64:53:B4:6B:95:F3:B7:DF:56:D3:4A:09:A8:FF
Trust this certificate? [no]: y
Certificate was added to keystore

# keytool -import -alias SecureServer -file server_cert -keystore client_keystore
-storepass idcidc

Owner: CN=SecureServer
Issuer: CN=SecureServer
Serial number: 3c42e61e
Valid from: Mon Jan 14 08:07:26 CST 2002 until: Sun Apr 14 09:07:26 CDT 2002
Certificate fingerprints:
  MD5: 43:2F:7D:B6:A7:D3:AE:A7:2E:21:7C:C4:52:49:42:B1
  SHA1: ED:B3:BB:62:2E:4F:D3:78:B9:62:3B:52:08:15:8E:B3:5A:31:23:6C
```

```
Trust this certificate? [no]: y
Certificate was added to keystore
```

The certificates of each program have now been imported into the keystore of the other.

13 Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at http://www.oracle.com/accessibility/.

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/support/contact.html or visit http://www.oracle.com/accessibility/support.html if you are hearing impaired.

Oracle Fusion Middleware Developer's Guide for Remote Intradoc Client (RIDC), 11*g* Release 1 (11.1.1)
E16819-01