EMC® Documentum®
Documentum Foundation Services

Version 6.7

Development Guide

EMC Corporation
Corporate Headquarters:
Hopkinton, MA 01748-9103
1-508-435-1000
www.EMC.com

Documentation Feedback

Your opinion matters. We want to hear from you regarding our product documentation. If you have feedback about how we can make our documentation better or easier to use, please send us your feedback directly at IIGDocumentationFeedback@emc.com.

Table of Contents

List of Figures

List of Tables

Preface

This document is a guide to using EMC Documentum Foundation Services (DFS) for the development of DFS service consumers, and of custom DFS services.

Intended readership

This document is intended for developers and architects building consumers of DFS services, and for service developers seeking to extend DFS services with custom services. This document will also be of interest to managers and decision makers seeking to determine whether DFS would offer value to their organization.

Revision history

The following changes have been made to this document.

Revision history

Revision Date	Description
February 2013	Updated for 6.7 SP2: Added multi-domain Kerberos support.
April 2011	Initial Publication

For more information

This document is part of a set that also includes the following documents:

- The *EMC Documentum Enterprise Content Services Reference*. This volume provides reference material and usage examples (in Java and C#.NET) for all of the services delivered with the DFS product, as well as services delivered with related products.
- The *EMC Documentum Foundation Services Deployment Guide*, which provides instructions for deploying DFS on all supported application servers.
- The *EMC Documentum Foundation Services Release Notes*. The release notes for the current release contain important information about new features in the release, known and fixed bugs, system requirements, and technical notes.

For additional information, refer to the Javadocs or .NET HTML help files, to the sample code delivered with the DFS SDK, and to resources on EMC Developer Network (https://community.emc.com/community/edn/documentum).

Conventions for referring to methods and properties

This development guide makes reference to the DFS Java and .NET API, with occasional references to the web services SOAP API. All of the APIs use the same underlying data model, but have slightly different naming conventions.

For public method names C# conventionally uses Pascal case (for example GetStatus), while Java uses "camel case" (getStatus). The corresponding WSDL message uses the same naming convention as the Java method. This document will use the convention followed by Java and the SOAP API.

Java uses getter and setter methods for data encapsulation (properties are an abstraction) and C# uses properties; these correspond to typed message parts in the SOAP API. This document will refer to such an entity as a *property*, using the name from the SOAP API. For example:

C# Property	Java getters/setters	Refer to as property
SourceLocation	getSourceLocation, setSourceLocation	sourceLocation

Chapter 1

Overview

This chapter is intended to provide a brief overview of DFS products and technologies. This chapter covers the following topics:

What are Documentum Foundation Services?

EMC Documentum Foundation Services (DFS) are a set of technologies that enable service-oriented programmatic access to the EMC Documentum Content Server platform and related products. DFS is a service layer over a Documentum Foundation Classes (DFC) client, which connects to one or more Documentum repositories managed by a Content Server.

The following table lists some of the technologies that are included in DFS:

Table 1. DFS technologies

DFS technology	Description
Web services	The DFS web services are SOAP/WSDL-based services deployed in a Java EE application server.
Java services	DFS can also be deployed as a local Java API using class libraries provided in the SDK. In a local application, the DFS services run in the same JVM as the service consumer.
Data model	The DFS WSDL interface and corresponding class libraries define a service-oriented data model for representing objects and properties, and for profiling service operation options.

DFS technology	Description
Java client productivity layer	Optional client-side libraries for Java consumers of DFS.
.NET client productivity layer	Optional client-side libraries for .NET consumers of DFS.
Tools for generating services	Service-generation tools based on JAX-WS (Java API for XML-based Web Services), and Ant, which generate deployable DFS services from annotated source code, or from WSDL. These tools also generate client-side runtime support for Java clients. Client-side .NET classes are generated using the DFS Proxy Generator utility.

Other EMC products provide services that are compatible with the DFS framework. The overarching term for the services as a whole is *Enterprise Content Services.* For a comprehensive reference to the available services, refer to the *Enterprise Content Services Reference.*

Web services

The DFS web services are SOAP/WSDL-based services that can be accessed using a standard WSDL client such as Axis 2 or the Sun JAX-WS reference implementation, or by using the DFS Java or .NET productivity layer. The DFS web services are installed as part of Content Server, where they run in the Java Method Server on the same machine as the Content Server. You can also deploy DFS on a separate tier or cluster on any supported JEE application server. For details on supported application servers, refer to the *Documentum Foundation Services Release Notes.* For deployment instructions, refer to the *Documentum Foundation Services Deployment Guide.*

Java services

The DFS Java services are delivered on the DFS SDK as a class library. Like the web services, the Java services are a service layer over a DFC client, which connects to Documentum Content Server. The DFS Java services are exposed as Java interfaces and run in the same Java Virtual Machine as the service consumer. The Java services are optimal for building an application that integrates a UI server or custom web service layer with DFS on a single tier, rather than consuming DFS services remotely from multiple client locations. The Java services are in almost all areas functionally identical to the web services, to the extent that it is possible to build a test consumer that can be switched between local and remote modes (as in fact is done in the SDK Java consumer samples). This can be useful in for debugging custom services locally before testing them in a remote deployment. However, there are significant differences between the Java services and web services in specific areas, such as content transfer modes and content return types, service context registration, and Kerberos integration.

Productivity layer

The DFS SDK includes Java and .NET client class libraries known as the *productivity layer*. We want to emphasize that the productivity layer is not required to consume DFS services, but it does provide features that may simplify development, such as:

- Convenience methods and constructors that simplify working with collections, instantiation of service proxies, registration of service contexts, and so forth.
- Transparent integration with ACS and BOCS for distributed content.
- Transparent invocation of UCF and handling of UCF content transfer.
- Support classes for Kerberos and other SSO authentication.
- The ability to execute the service either remotely via web services or locally within the same JVM (Java only).

The Java productivity layer is based on the Sun reference implementation of JAX-WS. The Java productivity layer can be used either as a consumer of DFS web services (remote mode), or a local consumer, running in the same JVM as the DFS Java services.

The .NET productivity layer is based on Microsoft Windows Communication Framework (WCF) and has functional parity with the Java productivity layer.

DFS and DFC

DFS is a service-oriented API and an abstraction layer over the Documentum Foundation Classes (DFC). DFS is generally simpler to use from the perspective of consumer development than DFC and is intended to allow development of client applications in less time and with less code. It also greatly increases the interoperability of the Documentum platform and related technologies by providing WSDL interface to SOAP clients (and client libraries for Java and .NET). However, because it exposes a data model and service API that are significantly different from DFC, it does require some reorientation for developers who are used to DFC.

When programming in DFS, some of the central and familiar concepts from DFC are no longer a part of the model. Session managers and sessions are not part of the DFS abstraction for DFS consumers. However, DFC sessions are used by DFS services that interact with the DFC layer. The DFS consumer sets up identities (repository names and user credentials) in a *service context*, which is use to instantiate service proxies, and with that information DFS services take care of all the details of getting and disposing of sessions. For more details on how sessions are used in DFS, see DFC sessions in DFS services, page 226. DFS does not have (at the exposed level of the API) an object type corresponding to a SysObject. Instead it provides a generic DataObject class that can represent any persistent object, and which is associated with a repository object type using a property that holds the repository type name (for example “dm_document”). Unlike DFC, DFS does not generally model the repository type system (that is, provide classes that map to and represent repository types). Any repository type can be represented by a DataObject, although some more specialized classes can also represent repository types (for example Acl or a Lifecycle).

In this documentation, we’ve chosen to call the methods exposed by DFS services *operations*, in part because this is what they are called in the WSDLs that represent the web service APIs. Don’t confuse

the term with DFC operations—in DFS the term is used generically for any method exposed by the service.

DFS services generally speaking expose a just a few service operations. The operations generally have simple signatures. For example the Object service update operation has this signature:

```
DataPackage update(DataPackage dataPackage, OperationOptions options)
```

However, this "simple" operation provides a tremendous amount of power and flexibility. It's just that the complexity has moved from the "verbs" (the number of methods and the complexity of the method signature) to the "nouns" (the objects passed in the operation). The operation makes a lot of decisions based on the composition of the objects in the DataPackage and relationships among those objects, and on profiles and properties provided in the operationOptions parameter or set in the service context—these settings are used to modify the default assumptions made by the service operation. This approach helps to minimize chattiness in a distributed application. The DFS client spends most of its effort working with local objects, rather than in conversation with the service API.

XML Support

Documentum's XML support has many features. The XML support provided by DFS is similar to the way in which DFC supports XML. For more information on XML processing options (import/export) and XML support, refer to *DFC Development Guide*.

Using XML support requires you to provide a controlling XML application. When you import an XML document, DFC examines the controlling application's configuration file and applies any chunking rules that you specify there. If the application's configuration file specifies chunking rules, DFC creates a virtual document from the chunks it creates. It imports other documents that the XML document refers to as entity references or links and makes them components of the virtual document. It uses attributes of the containment object associated with a component to remember whether it came from an entity or a link and to maintain other necessary information. Assembly objects have the same XML-related attributes as containment objects do. The processed XML files are imported in Content Server as virtual documents and therefore, in order to retrieve the XML files, you must use methods that are applicable for processing virtual documents.

DFC provides substantial support for the Documentum XML capabilities. XML processing by DFC is largely controlled by configuration files that define XML applications. Refer to the *XML Application Development Guide* for information about working with content in XML format.

The following declaration sets an application name:

```
ContentTransferProfile.setXMLApplicationName(String xmlApplicationName);
```

If no XML application is provided, DFC will use the default XML application for processing. To disable XML processing, set the application name to **Ignore**.

Use the UCF mode for importing XML files with external links and uploading external files. If you use other content transfer modes, only the XML file will be imported and the links will not be processed.

DFS tools

The DFS tools provide functionality for creating services based on Java source code (“code first”), services based on WSDL (“WSDL first”), or client runtime support for existing services based on WSDL. These tools can be used through a Composer interface, or scripted using Ant.

DFS services can be implemented as POJOs (Plain Old Java Objects), or as BOF (Business Object Framework) service-based business objects (SBOs). The service-generation tools build service artifacts that are archived into a deployable EAR file for remote execution and into JAR files for local execution using the optional client runtime. C# client-side proxies are generated using the DFS Proxy Generator utility.

For information on using the tools through the Composer interface, refer to the *Composer User Guide*.

The following table describes the supported Ant tasks that can be used for tools scripting:

Table 2. DFS tools tasks

Ant task name	Class name	Description
generateModel	com.emc.documentum.fs.tools.GenerateModelTask	Generates the service model that describes how to generate service artifacts
generateArtifacts	com.emc.documentum.fs.tools.GenerateArtifactsTask	Generates the service artifacts that are required to build and package the service
buildService	com.emc.documentum.fs.tools.BuildServiceTask	Builds jar files with the output generated by generateArtifacts
packageService	com.emc.documentum.fs.tools.PackageServiceTask	Packages all of the service artifacts and jar files into a deployable EAR or WAR file
generateService	com.emc.documentum.fs.tools.GenerateServiceTask	Generates service proxies from a given WSDL that you can use to create a custom service
generateRemoteClient	com.emc.documentum.fs.tools.GenerateRemoteClientTask	Generates client proxies from a given WSDL that you can use to create a service consumer
generatePublishManfiest	com.emc.documentum.fs.tools.GeneratePublishManifestTask	Generates a manifest that contains information on how to publish a service

To avoid out of memory errors when running DFS build tools in Ant, set the ANT_OPTS environment variable. For running in a UNIX shell, DOS, or Cygwin, set ANT_OPTS to “-Xmx512m -XX:MaxPermSize=128m” to solve an OutOfMem and PermGen space error.

For running in Eclipse, set the Window -> Preferences -> Java -> Installed JREs -> Edit > Default VM Parameter to “Xmx512m -XX:MaxPermSize=128m” .

For more information on building custom services and the build tools, see Chapter 8, Custom Service Development with DFS.

Enterprise Content Services

Enterprise Content Services (ECS), which includes all services that operate within the DFS framework, share a common set of technologies built around JAX-WS, including a service context, use of the ContextRegistry and Agent DFS runtime services, and common runtime classes. DFS delivers a set of core Enterprise Content Services, which are deployed with Content Server, where they are hosted by the Java Method Server. These services can also be deployed in a standalone or clustered configuration using a separate installation (For details, see the *Documentum Foundation Services Deployment Guide.)* The services that are provided with DFS can be extended with additional Enterprise Content Services provided by EMC, partners, and customers. For more information on the Enterprise Content Services that are not packaged with DFS, see the *Enterprise Content Services Reference.*

Chapter 2

Consuming DFS Using Web Service Frameworks

This chapter provides information about how to consume DFS services remotely using web services frameworks and the DFS WSDL interface (or for that matter any Enterprise Content Services WSDL), without the support of the client productivity layer. The chapter will present some concrete examples using Axis2 to illustrate a general approach to DFS service consumption that is applicable to other frameworks. Documentation of an additional Java sample that uses JAX-WS RI and demonstrates content transfer using the Object service, is available on the EMC Developer Network (EMC) at https://community.emc.com/docs/DOC-3038. A .NET consumer sample is available at https://community.emc.com/docs/DOC-2604.

Principal issues

The following issues apply generally to DFS consumers that do not use the client productivity layer.

Proxy generation

The sample consumers presented here use frameworks that generate static proxies that are bound to data types that defined in the WSDL and associated XML schemas. Using these classes, the framework takes care of marshalling the data that they contain into XML included in SOAP requests and unmarshalling objects in SOAP responses into instances of the proxy classes. This frees the programmer to work with the proxy classes in an object-oriented manner and exchange instances of the types with remote operation as if it were a local method. The samples presented here use static proxies, so generating the proxies is one of the first steps in creating a consumer. Some frameworks, such as Apache CXF, support dynamic proxies that are created at runtime.

Service context and authentication

DFS requires a correctly formatted service context to be included in the header of SOAP requests. The service context includes the user credentials as well as data about service state.

There are two supported approaches to the management of service context:

- consuming services in a stateless manner, in which all contextual data is passed in the SOAP request
- registered service contexts, in which service state is registered on the server and referenced with a token in service invocations

In the latter case any state information passed in the SOAP request header is merged into the service context on the server.

In general we *promote* the stateless option, which has the virtual of being simpler and avoids some limitations that are imposed by maintaining state on the server; however both options are fully supported.

Whichever of these options you choose, if you are not using the client productivity layer your consumer code will need to modify the SOAP header. In the case of stateless consumption, a serviceContext header is constructed based on user credentials and other data regarding service state and placed in the SOAP envelope. In the case of a registered service context, the consumer invokes the DFS ContextRegistryService to obtain a token in exchange for the service context data. The token must then be included within the SOAP request header within a wsse:Security element. Both of these techniques are illustrated in the provided Axis2 samples.

Note: Use of a BinarySecurityToken is supported only if the client registers the service context. The client must register the service context and obtain the security token as described in the Axis 2 sample, then inject the BinarySecurityToken into the SOAP header prior to calling the service.

For more information about service context registration see Service context registration, page 76.

Exception handling

If you are creating a Java consumer, you have the option of having the full stack trace of service exceptions serialized and included in the SOAP response. You can turn this on by setting the dfs.exception.include_stack_trace property in the service context to true. By default, the server returns the top level exception and stack trace, but not the entire exception chain. Non-Java clients should use the default setting.

Content transfer

DFS supports a number of different options for transferring content to and from the repository. The sample in this chapter does not illustrate content transfer. The topic is covered in Chapter 10, Content Transfer and Chapter 11, Content Transfer with Unified Client Facilities.

Consuming DFS with Axis2

This section describes how to generate client-side proxies with Axis2 and then use them to consume DFS. The basic guidelines for consuming DFS without the productivity layer remains the same with other tools and languages. The AxisDfsNoRegistrationConsumer.java demonstrates how to

consume services in a stateless manner by passing service context information in the SOAP header. The AxisDfsConsumer.java sample demonstrates how to write a consumer that registers the service context and passes a security token in the SOAP header. The following sections walk you through the steps of re-creating these samples:

As a convenience, an Ant build.xml file is provided for the Axis samples. You can use this file to compile and run the samples instead of carrying out the tasks in the following sections manually. For more information see, Running the Axis samples, page 30

Configuring the classpath

You will need all of the necessary jars that your consumer application requires before building your consumer. This will vary from consumer to consumer, so for the sake of simplicity, include all of the jar files that are located in the "lib" folder of your Axis2 installation. The samples that are contained in the DFS SDK only require these jar files to be present along with the generated client proxies that you will generate later. For your consumer, you will also need any other third party jar files that it depends on.

Generating the client proxies

These client proxies are classes that provide an API to the remote services and to the data types required by the service operations. You will need Axis2 1.4 before you begin this procedure. The provided build.xml files for the Axis samples have targets to generate the proxies for you if you wish to use them. The following procedure shows you how to generate the proxies on the command line, so you know how they are built.

To generate the client proxies:

Generating client proxies with Apache Axis2 1.4

1. Generate the proxies for the ContextRegistryService with the following command:

   ```
   wsdl2java -o javaOutputDirectory -d jaxbri
   -uri http://host:port/services/core/runtime/
   ContextRegistryService?wsdl
   ```

   where the variables represent the following values:

   - *javaOutputDirectory* — the directory where you want the java client proxies to be output to
   - *host:port* — the host and port where DFS is located

   The classes that are generated from this WSDL are recommended for all DFS consumers regardless of whether or not you register the service context. The ContextRegistryService provides convenience classes such as the ServiceContext class, which makes developing consumers easier.

2. Generate the proxies for each service that you want to consume with the following command. For the samples to work correctly, generate proxies for the SchemaService:

   ```
   wsdl2java -o javaOutputDirectory -d jaxbri
   -uri http://host:port/services/
   module/ServiceName?wsdl
   ```

   where the variables represent the following values:

   - *javaOutputDirectory* — the directory where you want the java client proxies to be output to
   - *host:port* — the host and port where DFS is located
   - *module* — the name of the module that the service is in, such as "core" or "search"
   - *ServiceName* — the name of the service that you want to consume, such as "SchemaService" or "SearchService"

3. Add the directory that you specified for *javaOutputDirectory* as a source folder in your project. They need to be present for your consumer to compile correctly.

Once you have the proxies generated, you can begin writing your consumer.

Writing a consumer that registers the service context

When a consumer registers a service context, the context is stored on the server and can be referenced with a security token. You can pass the security token on subsequent service calls and do not have to send the service context with each service request. There are four basic steps to writing a consumer that registers service contexts:

- Creating the service context and registering it, page 24
- Injecting the security token into the SOAP header, page 25
- Preparing input to the service operation (registered), page 26
- Calling the service operation (registered), page 26

The AxisDfsConsumer classes demonstrate how to consume DFS with registered service contexts. The code samples that are shown in the following samples are taken from these classes. You can obtain the full sample from the %DFS_SDK%/samples/JavaConsumers/DesktopWsdlBased directory.

Creating the service context and registering it

Before calling a service, you need to create a ServiceContext object that contains information such as user credentials and profiles. The following code sample shows a simple method that calls the Context Registry Service given a ServiceContext object. The return value is the security token that you need to inject into the SOAP header.

Example 2-1. Registering the service context in Axis2

```
private String registerContext(ServiceContext s) throws RemoteException
{
    Register register = new Register();
    register.setContext(s);
    ContextRegistryServiceStub stub =
```

```
      new ContextRegistryServiceStub(this.contextRegistryURL);
    RegisterResponse response = stub.register(register);
    return response.getReturn();
}
```

The following SOAP message is the SOAP message request when a context gets registered:

```
<?xml version="1.0" ?>
 <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
   <ns7:register
       xmlns:ns2="http://properties.core.datamodel.fs.documentum.emc.com/"
   xmlns:ns3="http://profiles.core.datamodel.fs.documentum.emc.com/"
   xmlns:ns4="http://context.core.datamodel.fs.documentum.emc.com/"
   xmlns:ns5="http://content.core.datamodel.fs.documentum.emc.com/"
   xmlns:ns6="http://core.datamodel.fs.documentum.emc.com/"
   xmlns:ns7="http://services.rt.fs.documentum.emc.com/">
    <context>
     <ns4:Identities xmlns:xsi="http://www.w3.org/2001/
                                    XMLSchema-instance"
     xsi:type="ns4:RepositoryIdentity" repositoryName="repository"
     password="password" userName="username"/>
    </context>
    <host>http://host:port/services/core
    </host>
   </ns7:register>
  </S:Body>
 </S:Envelope>
```

Injecting the security token into the SOAP header

When registering a service context with the Context Registry service, a security token is returned that you can use to refer to the service context that is stored on the server. You can then use the security token in subsequent requests to the service, so the consumer does not have to re-send service context information over the wire. To utilize this feature, you must inject the security token into the SOAP header before calling the service. To do this, create a WS-Security compliant header that contains the security token. Once the header is created, you can add it to the SOAP header. The following code sample shows a method that creates the security header given the security token that was returned by the Context Registry Service:

Example 2-2. Creating the security header in Axis2

```
private OMElement getSecurityHeader(String token)
{
  OMFactory omFactory = OMAbstractFactory.getOMFactory();
  OMNamespace wsse = omFactory.createOMNamespace(
    "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
                                            secext-1.0.xsd","wsse");
  OMElement securityElement = omFactory.createOMElement("Security", wsse);
  OMElement tokenElement = omFactory.createOMElement("BinarySecurityToken",
                                                     wsse);
  OMNamespace wsu = tokenElement.declareNamespace(
    "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
                                                utility-1.0.xsd","wsu");
  tokenElement.addAttribute("QualificationValueType",
    "http://schemas.emc.com/documentum#ResourceAccessToken",
                                                     wsse);
  tokenElement.addAttribute("Id", "RAD", wsu);
```

```
  tokenElement.setText(token);
  securityElement.addChild(tokenElement);
  return securityElement;
}
```

The following snippet of XML is what the security header should look like. The value for the BinarySecurityToken element is the token that was returned by the Context Registry Service.

```
<wsse:Security xmlns:wsse=
"http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
                                                    secext-1.0.xsd">
 <wsse:BinarySecurityToken
 QualificationValueType="http://schemas.emc.com/documentum#ResourceAccessToken"
 xmlns:wsu=
    "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
                                          utility-1.0.xsd" wsu:Id="RAD">
  hostname/123.4.56.789-123456789123-45678901234567890-1
 </wsse:BinarySecurityToken>
</wsse:Security>
```

Preparing input to the service operation (registered)

After obtaining the security token and creating the security header, you can then begin to code your consumer logic and setup any data model objects that the service operation requires. The JaxWsDfsConsumer sample is intentionally simple and does not require much preparation for the call to the getRepositoryInfo operation of the Schema service.

Calling the service operation (registered)

Once you have the input that is required by the service operation, you can set up the call to the service. The following code sample shows how to instantiate a service, get its port, and call an operation on the port. The code also shows how to set the outbound headers for the request to the service, which is important because it adds the security header to the request.

Example 2-3. Calling the Schema Service in Axis

```
public void callSchemaService()
{
  try
  {
    schemaService = new SchemaService(
      new URL(schemaServiceURL),
      new QName("http://core.services.fs.documentum.emc.com/",
                                          "SchemaService"));
    System.out.println("Retrieving the port from the Schema Service");
    SchemaServicePort port = schemaService.getSchemaServicePort();
    System.out.println("Invoking the getRepositoryInfo operation on the port.");

    //Set the security header on the port so the security
    //token is placed in the SOAP request
    WSBindingProvider wsbp = ((WSBindingProvider) port);
    Header h = Headers.create(header);
    wsbp.setOutboundHeaders(h);
```

```
    //Call the service
    RepositoryInfo r = port.getRepositoryInfo(((RepositoryIdentity)
      (serviceContext.getIdentities().get(0))).getRepositoryName(),
                                                          null);
    System.out.println("Repository Default Schema Name:" +
      r.getDefaultSchemaName() + "\n" +
      "Repository Description: " + r.getDescription() +
      "\n" + "Repository Label: " + r.getLabel() +
      "\n" + "Repository Schema Names: " + r.getSchemaNames());
  }
  catch (Exception e)
  {
    e.printStackTrace();
  }
}
```

The following SOAP message gets sent as the request when calling the Schema Service's getRepositoryInfo operation:

```
<?xml version="1.0" ?>
  <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Header>
      <wsse:Security xmlns:wsse=
      "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
                                                        secext-1.0.xsd">
        <wsse:BinarySecurityToken
        QualificationValueType="http://schemas.emc.com/
                                documentum#ResourceAccessToken"
        xmlns:wsu=
        "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
                                              utility-1.0.xsd"
        wsu:Id="RAD">
           hostname/123.4.56.789-123456789123-45678901234567890-1

        </wsse:BinarySecurityToken>
      </wsse:Security>
    </S:Header>
 <S:Body>
    <ns7:getRepositoryInfo
    xmlns:ns2="http://properties.core.datamodel.fs.documentum.emc.com/"
    xmlns:ns3="http://core.datamodel.fs.documentum.emc.com/"
    xmlns:ns4="http://content.core.datamodel.fs.documentum.emc.com/"
    xmlns:ns5="http://profiles.core.datamodel.fs.documentum.emc.com/"
    xmlns:ns6="http://schema.core.datamodel.fs.documentum.emc.com/"
    xmlns:ns7="http://core.services.fs.documentum.emc.com/">
      <repositoryName>repository</repositoryName>
    </ns7:getRepositoryInfo>
  </S:Body>
</S:Envelope>
```

Writing a consumer that does not register the service context

When a consumer does not register a service context, the state of the service is not maintained on the server. Consequently, service contexts must be passed to every call to a service and maintained on the consumer. There are three basic steps to writing a consumer that registers the service context:

1. Creating a service context with login credentials, page 28
2. Preparing input to the service operation (unregistered), page 28
3. Calling the service operation (unregistered), page 28

The AxisDfsNoRegistrationConsumer classes demonstrate how to consume DFS with unregistered service contexts. The code samples that are shown in the following samples are taken from these classes. You can obtain the full samples from the %DFS_SDK%/samples/JavaConsumers/DesktopWsdlBased directory.

Creating a service context with login credentials

If you do not register the service context, you have to pass in the credentials along with any other service context information with every call to a service operation. The state of the service is not kept on the server so you must maintain the service context on the client, and merging of service contexts on the server is not possible. The following code snippet shows how to create a simple service context object.

```
RepositoryIdentity identity = new RepositoryIdentity();
identity.setUserName(user);
identity.setPassword(password);
identity.setRepositoryName(repository);
ServiceContext context = new ServiceContext();
context.getIdentities().add(identity);
```

Preparing input to the service operation (unregistered)

After setting the credentials along with any other desired information in the service context, you can begin to code your consumer logic and setup any data model objects that the service operation requires. The JaxWsDfsConsumer sample is intentionally simple and does not require much preparation for the call to the getRepositoryInfo operation of the Schema service.

Calling the service operation (unregistered)

Once you have the input that is required by the service operation, you can setup the call to the service. The following code sample shows how to instantiate a service, get its port, and call an operation on the port. The code also shows the service context being set in the outbound header of the request to the service. This places the service context information, most notably the credentials, in the SOAP header so the service can authenticate the consumer. All other desired service context information must be present in every call to the service as it is not cached on the server.

Example 2-4. Calling the Schema Service in Axis2

```
public void callSchemaService()
{
   try{
   SchemaServiceStub stub = new SchemaServiceStub(this.objectServiceURL);

   // add service context to the header for subsequent calls
   ServiceClient client = stub._getServiceClient();
   SAXOMBuilder builder = new SAXOMBuilder();
  JAXBContext jaxbContext =
     JAXBContext.newInstance("com.emc.documentum.fs.datamodel.core.context");
  Marshaller marshaller = jaxbContext.createMarshaller();
  marshaller.marshal( new JAXBElement(
     new QName("http://context.core.datamodel.fs.documentum.emc.com/",
      "ServiceContext"), ServiceContext.class, serviceContext ), builder);
  OMElement header= builder.getRootElement();
   header.declareDefaultNamespace("http://context.core.datamodel.fs.
                                  documentum.emc.com/");
   client.addHeader(header);

   // invoke the service
   GetRepositoryInfo get = new GetRepositoryInfo();
   get.setRepositoryName(((RepositoryIdentity)
   serviceContext.getIdentities().get(0)).getRepositoryName());
   GetRepositoryInfoResponse response = stub.getRepositoryInfo(get);
   RepositoryInfo r = response.getReturn();
   System.out.println("Repository Default Schema Name:" +
   r.getDefaultSchemaName() + "\n" +
  "Repository Description: " + r.getDescription() + "\n" + "Repository Label: "
   + r.getLabel() + "\n" + "Repository Schema Names: " + r.getSchemaNames());
   }
   catch (Exception e)
   {
      e.printStackTrace();
   }
}
```

The following SOAP message gets sent as the request when calling the Schema Service's getRepositoryInfo operation:

```
<?xml version="1.0" ?>
 <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header>
   <ns4:ServiceContext
   xmlns:ns2="http://properties.core.datamodel.fs.documentum.emc.com/"
   xmlns:ns3="http://profiles.core.datamodel.fs.documentum.emc.com/"
   xmlns:ns4="http://context.core.datamodel.fs.documentum.emc.com/"
   xmlns:ns5="http://content.core.datamodel.fs.documentum.emc.com/"
   xmlns:ns6="http://core.datamodel.fs.documentum.emc.com/"
   token="temporary/USXXLYR1L1C-1210202690054">
   <ns4:Identities xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:type="ns4:RepositoryIdentity" repositoryName="repository"
   password="password" userName="username"/>
   </ns4:ServiceContext>
  </S:Header>
  <S:Body>
   <ns7:getRepositoryInfo
   xmlns:ns2="http://properties.core.datamodel.fs.documentum.emc.com/"
   xmlns:ns3="http://core.datamodel.fs.documentum.emc.com/"
   xmlns:ns4="http://content.core.datamodel.fs.documentum.emc.com/"
   xmlns:ns5="http://profiles.core.datamodel.fs.documentum.emc.com/"
   xmlns:ns6="http://schema.core.datamodel.fs.documentum.emc.com/"
   xmlns:ns7="http://core.services.fs.documentum.emc.com/">
    <repositoryName>repository</repositoryName>
```

```
   </ns7:getRepositoryInfo>
  </S:Body>
 </S:Envelope>
<?xml version="1.0" ?>
 <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header>
   <ns4:ServiceContext
   xmlns:ns2="http://properties.core.datamodel.fs.documentum.emc.com/"
   xmlns:ns3="http://profiles.core.datamodel.fs.documentum.emc.com/"
   xmlns:ns4="http://context.core.datamodel.fs.documentum.emc.com/"
   xmlns:ns5="http://content.core.datamodel.fs.documentum.emc.com/"
   xmlns:ns6="http://core.datamodel.fs.documentum.emc.com/"
   token="temporary/USXXLYR1L1C-1210201103234">
    <ns4:Identities xmlns:xsi="http://www.w3.org/2001/
                                  XMLSchema-instance"
    xsi:type="ns4:RepositoryIdentity" repositoryName="techpubs"/>
   </ns4:ServiceContext>
  </S:Header>
  <S:Body>
   <ns7:getSchemaInfo
   xmlns:ns2="http://properties.core.datamodel.fs.documentum.emc.com/"
   xmlns:ns3="http://core.datamodel.fs.documentum.emc.com/"
   xmlns:ns4="http://content.core.datamodel.fs.documentum.emc.com/"
   xmlns:ns5="http://profiles.core.datamodel.fs.documentum.emc.com/"
   xmlns:ns6="http://schema.core.datamodel.fs.documentum.emc.com/"
   xmlns:ns7="http://core.services.fs.documentum.emc.com/">
    <repositoryName>techpubs</repositoryName>
    <schemaName>DEFAULT</schemaName>
   </ns7:getSchemaInfo>
  </S:Body>
 </S:Envelope>
```

Running the Axis samples

The provided build.xml files for both the Axis samples compile and run the samples for you. The following table describes the files that you need to edit or run:

Table 3. Axis sample files

File	Location	Description
AxisDfsConsumer.java	%DFS_SDK%/samples/WSDLBasedConsumers/Axis/src	The Axis sample consumer that demonstrates how to consume DFS with registered service contexts.
AxisDfsNoRegistrationConsumer.java	%DFS_SDK%/samples/WSDLBasedConsumers/Axis/src	The Axis sample consumer that demonstrates how to consume DFS with unregistered service contexts.
build.xml	%DFS_SDK%/samples/WSDLBasedConsumers/Axis	The Ant script that builds and runs the Axis samples.
build.properties	%DFS_SDK%/samples/WSDLBasedConsumers/Axis	The Ant properties files that specify user specific options for the Ant build.

To run the Axis samples with the Ant script:

1. Edit the AxisDfsConsumer.java and AxisDfsNoRegistrationConsumer.java files and specify values for user, password, repository, and the location for the Schema service in the main method of each class. You also need to specify the location for the Context Registry service for AxisDfsConsumer.java.
2. Edit the build.properties file for the appropriate sample and specify values for axis.home, schema.service.wsdl, and context.registry.service.wsdl.
3. Enter “ant all” on the command line from the location of the build.xml file to compile and run the samples. This target calls the clean, artifacts, compile, run.registered.client, and run.unregistered.client targets. You can choose to run these targets individually to examine the output of each step.

Chapter 3

Getting Started with the Java Productivity Layer

This chapter describes how to run the Java consumers provided with the DFS SDK that utilize the Java productivity layer. The Java productivity layer is an optional client library that provides convenience classes to make it easier to consume DFS services using Java. For more general information about Java productivity layer consumers see Chapter 4, Consuming DFS with the Java DFS Productivity Layer.

Verifying prerequisites for the samples

Before running the consumer samples, verify that you have all these required prerequisites.

- A running DFS server. This can be a standalone instance of DFS running on your local machine or on a remote host, or it can be a DFS server installed with Content Server. For more information see Verify the DFS server, page 34.
- The DFS server that you are using needs to be pointed to a Connection Broker through which it can access a test repository. Your consumer application will need to know the name of the test repository and the login and password of a repository user who has Create Cabinet privileges. For more information see Verify repository and login information, page 35.
- Optionally, a second repository can be available for copying objects across repositories. This repository should be accessible using the same login information as the primary repository.
- You must have a Java 5 or 6 JDK installed on your system, and your JAVA_HOME environment variable should be set to the JDK location.
- You must have Apache Ant 1.7 or higher installed and on your path.
- The DFS SDK must be available on the local file system. Its location will be referred to as %DFS_SDK%. Make sure that there are no spaces in the folder names on the path to the SDK.
- The sample consumer source files are located in %DFS_SDK%\samples\DfsJavaSamples. This folder will be referred to as %SAMPLES_LOC%.

Note: The LifecycleService samples require installing sample data on your test repository. Before running these samples, install the Documentum Composer project contained in ./csdata/LifecycleProject.zip to your test repository using Documentum Composer, or install the DAR file contained in the zip archive using the darinstaller utility that comes with Composer. *Documentum Composer User Guide* provides more information on how to use a composer to install a DAR file on the repository.

A few samples, such as the VersionControlService, depend on the availability of ACS server. If you have problems with VersionControlService samples or ObjectService samples, ensure ACS is available. (For more information, refer to Distributed Configurations section in the *Documentum Administrator User Guide*.)

Verify the DFS server

You should verify that the DFS server application is running and that you have the correct address and port for the service endpoints. There are two probable scenarios.

The first is that DFS services are running on a Content Server installation, in which case DFS application will be listening on port 9080. This is the simplest scenario, because it does not require anything to be installed other than Content Server.

The second possibility is that a standalone DFS is deployed on a separate application server, perhaps one that you have installed on your local machine. In this case the port number will have been determined during deployment. (For instructions on how to deploy DFS, refer to the *Documentum Foundation Services Deployment Guide*).

In either case a DFS service endpoint address will take the following form:

protocol://*hostName*:
port/services/*moduleName*/
serviceName

For example, if your services are running on localhost at port 8888 and using the default http protocol (rather than https), the address of the Object service would be

```
http://localhost:8888/services/core/ObjectService
```

DFS services are organized into several different service modules. The central module, named “core”, contains the Object service and other fundamental Documentum platform services. Unless you specifically change the shipped configuration settings, “core” will be the default module.

To test whether you have the correct endpoint address and whether the DFS server is running and available, you can open a service WSDL in a browser. For example, if you want to test a DFS instance deployed with Content Server on a host with the DNS name MyContentServer, you could open the QueryService WSDL in your browser using the following URL (DFS on a Content Server installation is always at port 9080):

```
http://MyContentServer:9080/services/core/QueryService?wsdl
```

If the WSDL does not come up in your browser, make sure that the Java Method Server is running on the Content Server host.

Verify repository and login information

To access a repository, a DFS service will need three pieces of data to identify the repository and to authenticate a user on the repository.

- repository name
- user name of a user with Create Cabinet privileges
- user password

The repository name must be the name of a repository accessible to the DFS server. The list of available repositories is maintained by a *connection broker*.

Note: DFS knows where the connection broker is, because the IP address or DNS name of the machine hosting the connection broker is specified in the dfc.properties file stored in the DFS EAR file. If you are using the DFS server deployed with Content Server and running under the Java Method Server, then DFS will have been automatically configured to point to the connection broker running on that Content Server installation. If DFS was deployed to a separate application server using the installer, the connection broker host and port will have been set during DFS installation. If the EAR file was manually deployed to an application server, the connection broker host and port should have been set manually as part of the deployment procedure. For more details, see the *Documentum Foundation Services Deployment Guide*.

Local and remote consumers

The Java productivity layer supports two varieties of consumers: local and remote. In remote consumers, the client invokes services running on a remote application server using http (or https) and XML (SOAP). In local consumers, the services are part of the same local process. For production purposes most users are primarily interested in remote consumers; however, the local consumer is very valuable in a test environment, and may be an option in some production scenarios. The procedures in this chapter provide instructions for both local and remote consumers. For more details about this see Local and remote Java consumers, page 43.

Running your first sample consumer

The TQueryServiceTest class is a standalone sample consumer that demonstrates how to begin programming with DFS. The TQueryServiceTest also verifies that your DFS and Content Server installations are correctly communicating with each other.

This procedure walks you through running the sample using the provided Ant build script. This script and its properties file are located in %SAMPLES_LOC%. These require no modification if they are used from their original location in the SDK.

1. Edit the TQueryServiceTest.java file that is located in the %SAMPLES_LOC%\src\com\emc\documentum\fs\doc\samples\client directory and specify

the following hard coded information. (Of course in a real application you would never hard code these settings—we do it this way in the samples for the sake of simplicity.)

- repository
- userName
- password
- host

```
/*************************************************************
  * You must supply valid values for the following fields: */

 /* The repository that you want to run the query on */
 private String repository =  "YOUR_REPOSITORY_NAME";

 /* The username to login to the repository */
 private String userName =  "YOUR_USER_NAME";

 /* The password for the username */
 private String password =  "YOUR_PASSWORD";

 /* The address where the DFS services are located */
 private String host = "http://YOUR_DFS_HOST:PORT/services";
 /*************************************************************/
 /* The module name for the DFS core services */
 private static String moduleName = "core";
```

2. Run the following command to delete any previously compiled classes and compile the Java samples project:

   ```
   ant clean compile
   ```

3. Run the following command to run the TQueryServiceTest.class file:

   ```
   ant run -Dtest.class=TQueryServiceTest
   ```

   The TQueryServiceTest program queries the repository and outputs the names and IDs of the cabinets in the repository.

Exploring the DFS service consumer samples

The DFS SDK includes a suite of consumer samples that demonstrate the DFS core services and their functionality. This section describes how the samples are structured and how to run them using Ant or through the Eclipse IDE. Before beginning, you should ensure that you have met all the prerequisites outlines in Verifying prerequisites for the samples, page 33 before running the samples.

If you are running the samples in the Eclipse IDE, read the following sections:

If you are running the samples using Ant, read the following sections:

Overview of the consumer samples

The samples are organized into two packages. The com.emc.documentum.fs.doc.samples.client package is located in the %SAMPLES_LOC%\src folder, and contains the service consumer implementations. The consumer implementations show you how to set up calls to DFS. The com.emc.documentum.fs.doc.test.client package is located in the %SAMPLES_LOC%\test folder and contains the drivers for the consumer implementations. The package also contains convenience classes and methods for the drivers, which are described in the following list:

- com.emc.documentum.fs.doc.test.client.SampleContentManager—A class that creates and deletes sample objects in the repository. Most tests utilize the SampleContentManager to create objects in the repository.
- com.emc.documentum.fs.doc.test.client.DFSTestCase—The parent class of the test driver classes that must be edited for the tests to work.
- com.emc.documentum.fs.doc.test.client.ObjectExaminer—A convenience class that facilitates the examining and printing of object properties.

The rest of the classes in the com.emc.documentum.fs.doc.test.client package begin with the letter “T” and contain main methods to drive the consumer implementations in the com.emc.documentum.fs.doc.samples.client package. These classes set up sample data, run a particular service by creating an instance of a service implementation class, then clean up the sample data from the repository.

Creating an Eclipse project

You can set up the samples as a project in your IDE as an alternative to running the samples from Ant. If you are using Documentum Composer (an Eclipse-based tool for creating Documentum applications), you may want to set up a DFS project in Composer. This procedure describes setting up the samples in the Eclipse IDE version 3.2. To create an Eclipse project:

1. In Eclipse, click on File>New>Project....
2. In the New Project dialog box, choose Java Project from Existing Ant Build File, then click **Next**.
3. Browse to %SAMPLES_LOC%\build.xml, then click **Open.** Specify any name for the project, or accept the name provided from the Ant build file, then click **Finish.**
4. Now add the client configuration file to the classpath.

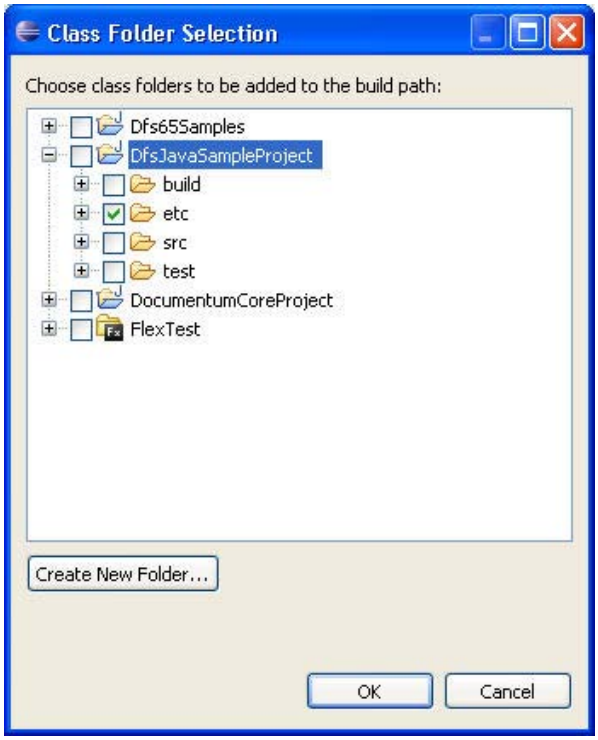
a. In the Package Explorer, right-click on the project and choose **Properties**.

b. In the project properties window, select on **Java Build Path**.

c. Click the **Add Class Folder** button.

d. In the Class Folder Selection window, find the heading for the new project, select the Etc folder check box, then click **OK**.

Figure 1. Adding the class folder



5. Under Project>Properties>Java Compiler, make sure that JDK compliance is set to 5.0, then click **OK**.

You should now be able to compile and run the samples in Eclipse after you have configured the samples correctly, which will be discussed in Setting hard coded values in the sample code, page 38 and Configuring DFS client properties (remote mode only), page 39.

Setting hard coded values in the sample code

The sample code depends on certain hard coded values that are contained in the com.emc.documentum.fs.doc.test.client.SampleContentManager and com.emc.documentum.fs.doc.test.client.DFSTestCase classes. You must verify that the values are correct before running the samples. To set the hard coded values:

1. Edit %SAMPLES_LOC%\test\com\emc\documentum\fs\doc\test\client\SampleContentManager.java and specify the values for the gifImageFilePath and gifImage1FilePath variables. The consumer

samples use these files to create test objects in the repository. Two gif images are provided in the %SAMPLES_LOC%\content folder, so you can set the variables to point to these files.

2. Edit the %SAMPLES_LOC%\test\com\emc\documentum\fs\doc\test\client\DFSTestCase.java file and specify the values for repository, userName, password, and remoteMode. You can optionally specify a value for the **toRepository** variable if you want to run a test such as copying an object from one repository to another. If you do not want to run the test, you must set the value of **toRepository** variable to null.

Configuring DFS client properties (remote mode only)

The configuration file %SAMPLES_LOC%\etc\dfs-client.xml provides some settings used by the DFS client runtime to call remote services. Edit the file as shown below, providing valid values for host (either the DNS name or IP address) and port number where the DFS server is deployed.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<DfsClientConfig defaultModuleName="core" registryProviderModuleName=
                                          "core">
    <ModuleInfo name="core"
                protocol="http"
                host="YOUR_HOST_NAME"
                port="YOUR_PORT"
                contextRoot="services">
    </ModuleInfo>
    <ModuleInfo name="bpm"
                protocol="http"
                host="YOUR_HOST_NAME"
                port="YOUR_PORT"
                contextRoot="services">
    </ModuleInfo>
    <ModuleInfo name="collaboration"
                protocol="http"
                host="YOUR_HOST_NAME"
                port="YOUR_PORT"
                contextRoot="services">
    </ModuleInfo>
    <ModuleInfo name="ci"
                protocol="http"
                host="YOUR_HOST_NAME"
                port="YOUR_PORT"
                contextRoot="services">
    </ModuleInfo>
</DfsClientConfig>
```

Note: There is more than one copy of this file on the SDK, so make sure you edit the one in %SAMPLES_LOC%\etc.

For more information see Configuring service addressing (remote consumers only), page 48.

Configuring dfc.properties (local mode only)

If you are running the samples in local mode (specified false for the remoteMode variable in DFSTestCase.java), you must edit the %DFS_SDK%\etc\dfc.properties file and specify correct values for dfc.docbroker.host[0] and dfc.dockbroker.port[0].

Also, to run the workflow service tests, you must specify correct values for dfc.globalregistry.username, dfc.globalregistry.password, and dfc.globalregistry.repository. If you are running the samples in remote mode, you do not have to edit your local copy of dfc.properties. The dfc.properties file that is on the server is used, which was configured during installation of DFS.

Note: The dfc.globalregistry.password setting stored in dfc.properties is encrypted by the DFS installer, so the easiest way to get this value is from the dfc,properties created by the installer. You can find it within the emc-dfs.ear file deployed on the application server in the APP-INF\classes directory.

Compiling and running the samples using Ant

To compile and run the productivity-layer consumer samples in Ant, follow these steps.

1. If necessary, modify the path to the DFS SDK root directory in the Ant %SAMPLES_LOC%/build.properties file. In the provided file, this value is set as follows—modify it as required:

   ```
   dfs.sdk.home=c:/emc-dfs-sdk-6.7/
   ```

2. Open a command prompt, change to the %SAMPLES_LOC% directory and execute the following command:

   ```
   ant clean compile
   ```

3. Execute the info target for information about running the samples.

   ```
   ant info
   ```

   This will print information to the console about available Ant targets and available samples to run.

   ```
   Buildfile: build.xml
        [echo] EMC DFS SDK home is 'c:/emc-dfs-sdk-6.7/'
        [echo] This project home is
        'C:\emc-dfs-sdk-65\samples\JavaConsumers\DesktopProductivityLayer'
        --beware spaces in this path (JDK issues).

   info:
        [echo] Available tasks for the project
        [echo] ant clean -  to clean the project
        [echo] ant compile - to compile the project
        [echo] ant run -Dtest.class=<class name>  - to run a test class
        [echo] Available test classes for run target:
        [echo]     TAccessControlService
        [echo]     TDriver
        [echo]     TLifecycleService
        [echo]     TObjServiceCopy
        [echo]     TObjServiceCreate
        [echo]     TObjServiceGet
        [echo]     TObjServiceDelete
        [echo]     TObjServiceMove
        [echo]     TObjServiceUpdate
        [echo]     TQueryServicePassthrough
        [echo]     TSchemaServiceDemo
        [echo]     TSearchService
        [echo]     TVersionControlServiceDemo
        [echo]     TVirtualDocumentService
        [echo]     TWorkflowService
   ```

   The classes listed are all in the com.emc.documentum.fs.doc.test.client package; these classes all contain the main methods that drive the consumer sample implementations.

The TDriver class runs all of the tests by calling the main method of each of the test classes. You can edit the TDriver class and comment out any tests that you do not want to run.

4. Run any of the classes, listed above, individually using the ant run target as follows:

```
ant run -Dtest.class=<className>
```

Chapter 4

Consuming DFS with the Java DFS Productivity Layer

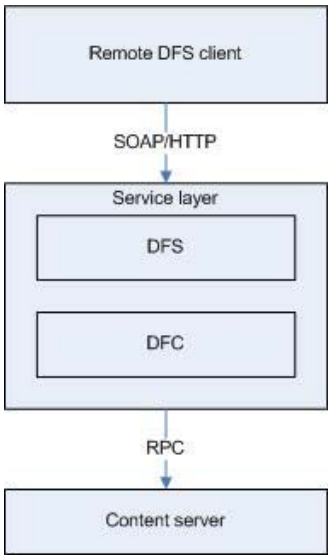
The DFS productivity layer contains a set of Java libraries that assist in writing DFS consumers. Using the DFS productivity layer is the easiest way to begin consuming DFS. This chapter covers the following topics:

Local and remote Java consumers

With the Java productivity layer, you have a choice between consuming remote web services, or running the services locally, within the same process as your consumer code. The Java sample code is set up so that it can be switched between remote and local modes.

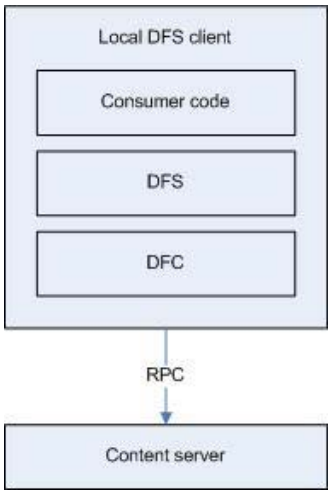
A *remote* DFS client is a web service consumer, using SOAP over HTTP to communicate with a remote DFS service provider. The service runs in the JEE container JVM and handles all of the implementation details of invoking DFC and interacting with Content Server.

Figure 2. Remote DFS consumer



In a *local* DFS client, both the consumer and service run on the same Java virtual machine. DFS uses a local DFC client to interact with Content Server. Consumer code invokes DFS services using the productivity layer, and does not invoke classes on the DFC layer.

Figure 3. Local Java DFS consumer



Local DFS deployment is a mainstream application topology, and is particularly prevalent in Java web applications, because it integrates the DFS client directly into the web application, which is simpler and more efficient that consuming web services over a remote connection.

Necessarily, a local DFS consumer differs in some important respects from a remote consumer. In particular note the following:

- Service context registration (which sets state in the remote DFS service) has no meaning in a local context, so registering the service context does nothing in a local consumer.
- Content transfer in a local application is completely different from content transfer in a remote application. Remote content transfer protocols (MTOM, Base64, and UCF) are not used by a local

consumer. Instead, content is transferred by the underlying DFC client. For more information see Content types returned by DFS, page 164.

Configuring Java dependencies for DFS productivity-layer consumers

To utilize the DFS Java productivity layer, you must include JAR files form the DFS SDK on your classpath or among your IDE project dependencies. Many of the jar files included in the SDK are not necessary for developing consumers (they may be required for service development or by the SDK tools). The dependencies will also vary depending on what services you need to invoke, and whether you want to invoke them remotely or locally.

To develop (or deploy) a DFS consumer that can only invoke services remotely, include the JARs listed in the following table.

Table 4. Remote consumer dependencies

Path on SDK	JARs
%DFS_SDK%/lib/java/	*-remote.jar
%DFS_SDK%/lib/java/jaxws/	All except jaxb-xjc.jar and jaxws-tools.jar
%DFS_SDK%/lib/java/commons/	commons-lang-2.4.jar, commons-io-1.2.jar
%DFS_SDK%/lib/java/utils/	aspectjrt.jar, log4j.jar, servlet.jar
%DFS_SDK%/lib/java/bof/	collaboration.jar

To develop (or deploy) a DFS consumer that can invoke services locally, include the JARs listed in the following table. A local consumer can also invoke services remotely, so these are the dependencies you will need to develop a consumer that can be switched between local and remote modes.

Table 5. Local consumer dependencies

Path on SDK	JARs
%DFS_SDK%/lib/java/	*-service.jar, emc-dfs-rt.jar
%DFS_SDK%/lib/java/jaxws/	All except jaxb-xjc.jar and jaxws-tools.jar
%DFS_SDK%/lib/java/commons/	commons-lang-2.4.jar, commons-io-1.2.jar
%DFS_SDK%/lib/java/utils/	aspectjrt.jar, log4j.jar, servlet.jar
%DFS_SDK%/lib/java/bof/	collaboration.jar, workflow.jar
%DFS_SDK%/lib/java/dfc/	dfc.jar, xtrim-api.jar

Framework-only consumer

If you are writing a consumer of a custom service that does not need to invoke any standard DFS services, and does not expose the DFS data model, your project does not need to include JARs from

%DFS_SDK%/lib/java/ other than emc-dfs-rt-remote.jar (for remote consumers) or emc-dfs-rt.jar (for local consumers). The following tables show these dependencies. (Of course, if there are additional dependencies that are specific to your consumer, you will need to include those as well.)

Table 6. Remote framework-only consumer dependencies

Path on SDK	JARs
%DFS_SDK%/lib/java/	emc-dfs-rt-remote.jar
custom	<your-custom>-services-remote.jar
%DFS_SDK%/lib/java/jaxws/	All except jaxb-xjc.jar and jaxws-tools.jar
%DFS_SDK%/lib/java/commons/	commons-lang-2.4.jar, commons-io-1.2.jar
%DFS_SDK%/lib/java/utils/	aspectjrt.jar, log4j.jar, servlet.jar
%DFS_SDK%/lib/java/bof/	collaboration.jar

Table 7. Local framework-only consumer dependencies

Path on SDK	JARs
%DFS_SDK%/lib/java/	emc-dfs-rt.jar
custom	<your-custom>-services.jar
%DFS_SDK%/lib/java/jaxws/	All except jaxb-xjc.jar and jaxws-tools.jar
%DFS_SDK%/lib/java/commons/	commons-lang-2.4.jar, commons-io-1.2.jar
%DFS_SDK%/lib/java/utils/	aspectjrt.jar, log4j.jar, servlet.jar
%DFS_SDK%/lib/java/bof/	collaboration.jar
%DFS_SDK%/lib/java/dfc/	dfc.jar, xtrim-api.jar

Data model classes in service JARs

Generally speaking, if a module exposes the data model of another module, then the service jars for the other module need to be on the classpath. For example, if a custom service uses the LifecycleInfo class, the class path would needs to include:

- <your-custom>-services.jar
- emc-dfs-services.jar

Similarly, a remote-only Java client would need to include:

- <your-custom>-services-remote.jar
- emc-dfs-services-remote.jar

Applications that use core services and the core data model should also include on their classpath, in addition to the core services and runtime jars:

- emc-search-services.jar (or emc-search-services-remote.jar), which includes classes required by the Query and QueryStore services.
- emc-collaboration-services.jar (or emc-collaboration-services-remote.jar), which includes classes related to the RichText object.

Adding dfs-client.xml to the classpath

Add the folder that contains dfs-client.xml to your classpath. For example, if the path of the file is %DFS_SDK%/etc/config/dfs-client.xml, add %DFS_SDK%/etc (dfs-client.xml can reside in a subfolder) or %DFS_SDK%/etc/config to your classpath.

Avoid having dctm.jar on classpath when executing services

Avoid having dctm.jar on the client classpath when executing services. Otherwise, it causes the DFS client runtime to use the wrong version of JAXB. Documentum installers that install DFC can add dctm.jar to the classpath.

Configuring dfc.properties

When you call DFS locally with the Java productivity layer, DFS uses the DFC client that is bundled in the DFS SDK. This DFC client is configured in a dfc.properties file that must be located on the project classpath (you can start with the copy that is provided in the %DFS_SDK/etc directory).

For remote execution of DFS services, you do not have to configure a local copy of dfc.properties.. DFS uses the DFC client that is bundled in the dfs.ear file that is deployed with Content Server, or on a standalone application server. In these cases, the minimum dfc.properties settings for the connection broker and global registry are set during the DFS installation. If you do not use the DFS installation program you will need to configure dfc.properties in the EAR file—see the *Documentum Foundation Services Deployment Guide* for details

To configure dfc.properties:

1. At a minimum, provide values for the dfc.docbroker.host[0] and dfc.docbroker.port[0] for the connection broker.
2. To run services that require an SBO, you will need to provide values for dfc.globalregistry. username, dfc.globalregistry.repository, and dfc.globalregistry.password.
3. Add the folder where dfc.properties is located to your classpath. For example, if you are using the %DFS_SDK%/etc/dfc.properties file, add %DFS_SDK%/etc to your classpath.

Note: For more information on SBOs, see the *Documentum Foundation Classes Development Guide.*

Configuring service addressing (remote consumers only)

The dfs-client.xml config file is used for runtime lookup of service address endpoints. This allows a client to instantiate services using implicit addressing, in which the address is not provided or partially provided—the DFS client support will look up the complete service address at runtime using the data provided in dfs-client.xml. For more information on instantiation methods and implicit addressing see Instantiating a service in Java, page 54.

If you are using explicit addressing all of the time, you do not have to configure the dfs-client.xml file, because you will be specifying the host information with each service invocation.

The following example illustrates a typical dfs-client.xml file:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<DfsClientConfig defaultModuleName="core" registryProviderModuleName="
                                                                    core">
    <ModuleInfo name="core"
                protocol="http"
                host="dfsHostName"
                port="8888"
                contextRoot="services">
    </ModuleInfo>
    <ModuleInfo name="bpm"
                protocol="http"
                host="dfsHostName"
                port="8888"
                contextRoot="services">
    </ModuleInfo>
    <ModuleInfo name="collaboration"
                protocol="http"
                host="dfsHostName"
                port="8888"
                contextRoot="services">
    </ModuleInfo>
    <ModuleInfo name="ci"
                protocol="http"
                host="dfsHostName"
                port="8888"
                contextRoot="services">
    </ModuleInfo>
   <ModuleInfo name="my_module"
               protocol="http"
               host="dfsHostName"
               port="8888"
               contextRoot="my_services">
    </ModuleInfo>
</DfsClientConfig>
```

A complete service address is composed of the following components:

- protocol—either http or https, depending on whether the application server is configured to use SSL.
- host—the DNS name or IP address of the service host.
- port—the port number at which the DFS application server listens. When DFS is installed with Content Server, the port defaults to 9080. The default port for a standalone DFS installation is 8888.

- contextRoot—the root address under which service modules are organized; the contextRoot for DFS-provided services is “services”
- name—the name of the service module, under which a set of related services are organized

The fully-qualified service address is constructed as runtime as follows:

```
<protocol>://<host>:<port>/<contextRoot>/<module>/<serviceName>
```

For example:

```
http://dfsHostName:8888/services/core/ObjectService
```

The defaultModuleName value is used when no module name is provided by the ServiceFactory getRemoteService method. The registryProviderModuleName value is used to specify the location of the ContextRegistryService where the service context will be registered if the module name is not explicitly provided by the ContextFactory register method.)

Creating a service context in Java

Service invocation in DFS takes place within a service context, which is an object that maintains identity information for service authentication, profiles for setting options and filters, a locale, and properties. Service contexts can be shared among multiple services.

Please also note that the service context is not thread safe and should not be accessed by separate threads in a multi-threaded application. If you require multiple threads your application must provide explicit synchronization.

Service context is represented in the client object model by the IServiceContext interface, instances of which encapsulate information that is passed in the SOAP header to the service endpoint during service registration and/or service invocation.

If a service context is registered, it is stored on the DFS server and represented by a token that is passed in the SOAP header during service invocation, along with optional service context data that is treated as a delta and merged into the existing service context. If a service is unregistered, the complete service context is passed in the SOAP header with each service invocation. There are advantages and disadvantages to both approaches (see Service context registration, page 76).

Properties and profiles can often be passed to an operation during service operation invocation through an OperationOptions argument, as an alternative to storing properties and profiles in the service context, or as a way of overriding settings stored in the service context.

Setting up service context (Java)

To be used by a service that requires authentication, the service context should be populated with at least one identity. The following sample creates and returns a minimal service context that contains a ContentTransferProfile:

Example 4-1. Minimal service context example

```
public IServiceContext getSimpleServiceContext(String repositoryName,
                                               String userName,
                                               String password)
```

```
{
    ContextFactory contextFactory = ContextFactory.getInstance();
    IServiceContext context = contextFactory.newContext();
    RepositoryIdentity repoId = new RepositoryIdentity();
    repoId.setRepositoryName(repositoryName);
    repoId.setUserName(userName);
    repoId.setPassword(password);
    context.addIdentity(repoId);
    return context;
}
```

Identities

A service context contains a collection of identities, which are mappings of repository names onto sets of user credentials used in service authentication. A service context is expected to contain only one identity per repository name. Identities are set in a service context using one of the concrete Identity subclasses:

- BasicIdentity directly extends the Identity parent class, and includes accessors for user name and password, but not for repository name. This class can be used in cases where the service is known to access only a single repository, or in cases where the user credentials in all repositories are known to be identical. BasicIdentity can also be used to supply fallback credentials in the case where the user has differing credentials on some repositories, for which RepositoryIdentity instances will be set, and identical credentials on all other repositories. Because BasicIdentity does not contain repository information, the username and password is authenticated against the global registry. If there is no global registry defined, authentication fails.
- RepositoryIdentity extends BasicIdentity, and specifies a mapping of repository name to a set of user credentials, which include a user name, password, and optionally a domain name if required by your network environment. In a RepositoryIdentity, you can use the "*" wildcard (represented by the constant RepositoryIdentity.DEFAULT_REPOSITORY_NAME) in place of the repository name. In this case DFS will authorize the credentials against the global registry. If no global registry is available, or if the credentials are not valid on the global registry, the authentication fails. Using this wildcard in a RepositoryIdentity is essentially the same as using a BasicIdentity.
- SsoIdentity allows an SSO solution to be used to authenticate the user. You can use SsoIdentity class when the service that you are requesting is accessing only one repository, or if the user credentials in all repositories are identical. Because SsoIdentity does not contain repository information, the username and password is authenticated against the designated global registry. If there is no global registry defined, authentication fails.
- BinaryIdentity is used only in a local Java client and is not serializable. It is used to encapsulate Kerberos credentials (see Kerberos authentication in a local DFS web application, page 206).
- PrincipalIdentity is used to indicate that DFC principal mode login should be used with the user name provided in the identity instance. PrincipalIdentity is not XML serializable, so it will not be sent over the wire. For security reasons, it will work only when the DFS service is invoked in local mode.

Locale

The locale property of an IServiceContext object specifies the language and optionally country setting to use for locale-sensitive features. The locale is used, for example, to control which NLS-specific Data Dictionary strings will be provided by Content Server to the DFS layer. The format of the locale string value is based on Java locale strings, which in turn are based on ISO 639-1 two-character, lowercase language codes and ISO 3166 country codes. The format of a Java locale string is <languagecode>[_<countrycode>]; for example, the Java locale string for British English is "en_GB". Further information on Java locale strings can be found at http://java.sun.com/developer/technicalArticles/J2SE/locale/.

To find out the locale codes currently supported by Content Server at installation, refer to the documentation for the locale_name property of the dm_server_config object in the *EMC Documentum System Object Reference Manual.*

If the locale is not set in the service context, the DFS server runtime will use the value set in the DFS server application. Typically this means that a DFS client (particularly a remote client) should set the locale to the locale expected by the user, rather than relying on the value set on the server. The locale setting used by the DFS server can be specified in the dfc.locale property of dfc.properties. If the value is not set in the service context by the client and not set on the server, the DFS server will use the locale of the JVM in which it is running.

Service context runtime properties

A service context contains a RuntimeProperties collection, in which properties can be set for all services sharing the service context. These properties settings can be used to store configuration settings that are scoped to the service context, and therefore are not suitable as operation parameters or inclusion in the OperationOptions PropertySet. Properties included in RuntimeProperties would generally be standalone properties. DFS services generally use profiles in preference to RuntimeProperties. The following table lists the properties that you can set in the service context.

Table 8. Service context properties

Property Name	Description
“dfs.exception.include_stack_trace”	A value of true indicates that a Java stack trace needs to be included in the serialized DFS exception.
IServiceContext.OVERRIDE_WSDL_ENDPOINT_ADDRESS	If set to FALSE the SOAP client uses the URL returned by the web service in its WSDL as the service address and not the one provided by the service developer. Has to be set to true if the service’s address is not publicly available.

IServiceContext.PAYLOAD_PROCESSING_POLICY	Sets whether or not to continue execution on an exception. Values can be “PAYLOAD_FAIL_ON_EXCEPTION” or “PAYLOAD_CONTINUE_ON_EXCEPTION” .
IServiceContext.USER_TRANSACTION_HINT	If set to IServiceContext.TRANSACTION_REQUIRED, attempts to run all "nested" calls in a single transaction, subject to the support provided by DFC. Valid values are IServiceContext.TRANSACTION_REQUIRED and IServiceContext.TRANSACTION_NOT_REQUIRED.

The PAYLOAD_PROCESSING_POLICY property is used when transferring multiple content objects to control whether the transfer will continue processing if the upload or download of a content object fails.

Transaction support

DFS provides basic support for transactions. If transactions are enabled operations will use implicit transactional behavior; that is, they will begin the transaction at the start of the operation and commit the transaction at the end of the operation if it completes successfully. If the any part of the operation fails, the entire operation will be rolled back.

To enable transactions, set the USER_TRANSACTION_HINT runtime property in the service context to IServiceContext.TRANSACTION_REQUIRED.

Combining USER_TRANSACTION_HINT and PAYLOAD_PROCESSING_POLICY

Note: Please note that PAYLOAD_PROCESSING_POLICY may be deprecated in a future release.

Transactional behavior for a service operation is enabled by setting the USER_TRANSACTION_HINT runtime property in the service context. It is possible to combine this setting with PAYLOAD_CONTINUE_ON_EXCEPTION, as shown here:

```
context.SetRuntimeProperty(IServiceContext.USER_TRANSACTION_HINT,
                           IServiceContext.TRANSACTION_REQUIRED);
context.SetRuntimeProperty(IServiceContext.PAYLOAD_PROCESSING_POLICY,
                           IServiceContext.PAYLOAD_CONTINUE_ON_EXCEPTION);
```

The expected behavior is that the payload policy must be honored first, then the transaction policy. For example, suppose that we use the Object service to create objects based on a DataPackage that has two DataObject trees. We use PAYLOAD_CONTINUE_ON_EXCEPTION with transaction support to create the objects. At runtime, a leaf in the first DataObject tree fails and all others succeed. In this case only the objects in the second DataObject tree would be created; the creation of the first DataObject tree would be rolled back. If no transaction support were used, some leaves from the first DataObject tree would be created, as well as the entire second DataObject tree.

Service context registration

Context registration is an optional technique for optimizing how much data is sent over the wire by remote DFS consumers. It is available for remote web services consumers, but does not apply to local Java consumers because the consumer and service share the same JVM. When you register a service context within a consumer, the DFS server-side runtime creates and maintains the service context on the server.

There are two benefits to registering the service context. The first benefit is that services can share a registered context. This minimizes over the wire traffic since the consumer does not have to send service context information to every service it calls.

The second benefit occurs when a consumer calls a service and passes in a *delta modification* to the service context. The DFS client runtime figures out the minimal amount of data to send over the wire (the modifications) and the server runtime merges the delta modifications into the service context that is stored on the server. If your application is maintaining a lot of data (such as profiles, properties, and identities) in the service context, this can significantly reduce how much data is sent with each service call, because most of the data can be sent just once when the service context is registered. On the other hand, if your application is storing only a small amount of data in the service context, there is really not much to be gained by registering the service context.

You should be aware that there are limitations that result from registration of service context.

- The service context can be shared only by services that share the same classloader. Typically this means that the services are deployed in the same EAR file on the application server. This limitation means that the client must be aware of the physical location of the services that it is invoking and manage service context sharing based on shared physical locations.
- Registration of service contexts prevents use of failover in clustered DFS installations.
- Registration of the service context is not supported with identities that store Kerberos credentials.

If you are using the DFS client productivity layer, registering a service context is mostly handled by the runtime, with little work on your part. You start by creating a service context object, then you call one of the overloaded register methods.

Table 9. Methods for registering services (Java)

ContextFactory method	Description
register(IServiceContext serviceContext)	Registers service context at default ContextRegistryService which is set in the registryProviderModuleName attribute in dfs-client.xml.
register(IServiceContext serviceContext, serviceModule)	Registers service context at the ContextRegistryService located in the specified serviceModule. The full address of the service module is looked up in dfs-client.xml by module name. In this case the module name in dfs-client.xml must be unique.
register(IServiceContext serviceContext, String serviceModule, String contextRoot)	Registers service context at the ContextRegistryService located in the specified serviceModule and contextRoot.

Table 10. Methods for registering services (.NET)

ContextFactory method	Description
Register(IServiceContext serviceContext)	Registers service context at default ContextRegistryService which is set in the registryProviderModuleName attribute in the configuration file.
Register(IServiceContext serviceContext, serviceModule)	Registers service context at the ContextRegistryService located in the specified serviceModule. The full address of the service module is looked up in the configuration file by module name. In this case the module name in the configuration file must be unique.
Register(IServiceContext serviceContext, String serviceModule, String contextRoot)	Registers service context at the ContextRegistryService located in the specified serviceModule and contextRoot.

If you wish to register the service context and are not using the productivity layer, you can register the context by invoking the ContextRegistry service directly (see Writing a consumer that registers the service context, page 24).

The register method can only be executed remotely and is meaningless in a local Java service client. If you are running your client in local mode, the register method will still result in an attempt at remote invocation of ContextRegistryService. If the remote invocation fails, an exception will be thrown. If the invocation succeeds (because there is a remote connection configured and available), there will be a harmless invocation of the remote service.

Instantiating a service in Java

A Java client (or a service) can create an instance of a service using one of several methods of ServiceFactory, shown in Table 11, page 54. These factory methods return service objects that allow the service to be executed either locally or remotely, and allow the service address to be explicitly provided, or obtained by lookup from dfs-client.xml. For more information on dfs-client.xml, see Configuring service addressing (remote consumers only), page 48.

Table 11. Methods for instantiating services

ServiceFactory method	Description
getRemoteService(Class<T> wsInterface, IServiceContext serviceContext)	Get service to be called using remote (SOAP) invocation. In this method neither the module name nor the context root is specified, so the service address is looked up in dfs-client.xml based on the defaultModuleName.

ServiceFactory method	Description
getRemoteService(Class<T> wsInterface, IServiceContext serviceContext, String serviceModule, String contextRoot)	Get service to be called using remote (SOAP) invocation, with contextRoot and module name explicitly provided. If null is passed in contextRoot, then the service address will be looked up based on the provided module name. In this case the module name must be unique in dfs-client.xml. If both serviceModule and serviceContext are null, then the lookup is based on the defaultModuleName in dfs-client.xml. If the serviceModule is null and serviceContext is not null, then the service is assumed to have no module name (that is, its address is contextRoot). Note that contextRoot is fully-qualified, and includes the protocol, host, and port: for example “http://localhost:8888/services”.
getLocalService(Class<T> wsInterface, IServiceContext serviceContext)	Get service to be called and executed in local JVM.
getService(Class<T> wsInterface, IServiceContext serviceContext)	Attempts to instantiate the service locally; if this fails, then attempts to instantiate the service remotely. If the invocation is remote, then the service address is looked up in dfs-client.xml based on the defaultModuleName.
getService(Class<T> wsInterface, IServiceContext serviceContext, String serviceModule, String contextRoot)	Attempts to instantiate the service locally; if this fails, then attempts to instantiate the service remotely. If the invocation is remote, then the service address is looked up in dfs-client.xml, with contextRoot and module name explicitly provided. If null is passed in contextRoot, then the service address will be looked up based on the provided module name. In this case the module name must be unique in dfs-client.xml. If both serviceModule and serviceContext are null , then the lookup is based on the defaultModuleName in dfs-client.xml. If the serviceModule is null and serviceContext is not null, then the service is assumed to have no module name (that is, its address is contextRoot). Note that the value passed to contextRoot is fully-qualified, and includes the protocol, host, and port: for example “http://localhost:8888/services”.

OperationOptions

DFS services generally take an OperationOptions object as the final argument when calling a service operation. OperationOptions contains profiles and properties that specify behaviors for the operation. The properties have no overlap with properties set in the service context’s RuntimeProperties. The profiles can potentially overlap with properties stored in the service context. In the case that they do overlap, the profiles in OperationOptions always take precedence over profiles stored in the service context. The profiles stored in the service context take effect when no matching profile is stored in the OperationOptions for a specific operation. The override of profiles in the service context takes place on a profile-by-profile basis: there is no merge of specific settings stored within the profiles.

As a recommended practice, a service client should avoid storing profiling information or properties in the service operation that are likely to be modified by specific service instances. This avoids possible side-effects caused by modifications to a service context shared by multiple services. It is likely that ContentTransferProfile will not change and so should be included in the service context. Other profiles are better passed within OperationOptions.

OperationOptions are discussed in more detail under the documentation for specific service operations. For more information on core profiles, see PropertyProfile, page 93, ContentProfile, page 97, PermissionProfile, page 101, and RelationshipProfile, page 110. Other profiles are covered under specific services in the *Enterprise Content Services Reference.*

WSDL-first consumption of services

The DFS SDK offers an Ant task that generates client proxies (from now on referenced to as the "light" API) from a service's WSDL, so you can consume third party services or even DFS and custom DFS services with a WSDL-first approach. The generateRemoteClient task generates and packages a light API for the service, which contains data model classes and a SOAP based client to consume the service. The light API differs from the DFS productivity layer API in the following ways:

- The productivity layer API features Java beans with additional convenience functionality and logic for the data model classes, while the light API only contains generated beans.
- The productivity layer API supports both local and remote service invocation, while the light API supports remote service invocation only.

The light API for the services is intended to be used in conjunction with the DFS productivity layer, so you can still utilize conveniences such as the ContextFactory and ServiceFactory. The generateRemoteClient task also generates a service model XML file and a dfs-client.xml file that you can use for implicit addressing of the services that you want to consume. For more information on the generateRemoteClient task, see the generateRemoteClient task, page 150 section. WSDL first consumption of DFS is also available through the Composer IDE. See the *Documentum Composer User Guide* for more information on generating the light API through Composer.

Chapter 5

Getting Started with the .NET Productivity Layer

This chapter has two goals. The first is to show you a basic DFS consumer, invoke a service, and get some results. This will let you know whether your environment is set up correctly, and show you the basic steps required to code a DFS consumer.

The second goal is to show you how to set up and run the DFS documentation samples. You may want to debug the samples in Visual Studio to see exactly how they work, and you may want to modify the samples or add samples of your own to the sample project.

Verifying prerequisites for the samples

Before running the consumer samples, verify that you have all these required prerequisites.

- A running DFS server. This can be a standalone instance of DFS running on your local machine or on a remote host, or it can be a DFS server installed with Content Server. For more information see Verify the DFS server, page 34.
- The DFS server that you are using needs to be pointed to a Connection Broker through which it can access a test repository. Your consumer application will need to know the name of the test repository and the login and password of a repository user who has Create Cabinet privileges. For more information see Verify repository and login information, page 35.
- Optionally, a second repository can be available for copying objects across repositories. This repository should be accessible using the same login information as the primary repository.
- For UCF content transfer, you must have a Java 5 or 6 Java Runtime Engine (JRE) installed on your system, and the JAVA_HOME environment variable should be set to the Java location.
- The DFS SDK must be available on the local file system.
- The sample consumer source files are located in %DFS_SDK%\samples\DfsJavaSamples. This folder will be referred to as %SAMPLES_LOC%.

Verify the DFS server

You should verify that the DFS server application is running and that you have the correct address and port for the service endpoints. There are two probable scenarios.

The first is that DFS services are running on a Documentum 6 or higher Content Server installation, in which case DFS application will be listening on port 9080. This is the simplest scenario, because it does not require anything to be installed other than Content Server. We recommend that if possible you use the latest version of Content Server.

The second possibility is that a standalone DFS is deployed on a separate application server, perhaps one that you have installed on your local machine. In this case the port number will have been determined during deployment. (For instructions on how to deploy DFS, refer to the *Documentum Foundation Services Deployment Guide*).

In either case a DFS service endpoint address will take the following form:

```
protocol://hostName:
port/services/moduleName/
serviceName
```

For example, if your services are running on localhost at port 8080 and using the default http protocol (rather than https), the address of the Object service would be

```
http://localhost:8080/services/core/ObjectService
```

DFS services are organized into several different service modules. The central module, named “core”, contains the Object service and other fundamental Documentum platform services. Unless you specifically change the shipped configuration settings, “core” will be the default module.

To test whether you have the correct endpoint address and whether the DFS server is running and available, you can open a service WSDL in a browser. For example, if you want to test a DFS instance deployed with Content Server on a host named MyContentServer, you could open the QueryService WSDL in your browser using the following URL:

```
http://MyContentServer:9080/services/core/QueryService?wsdl
```

If the WSDL does not come up in your browser, make sure that the Java Method Server is running on the Content Server host.

Verify repository and login information

To access a repository, a DFS service will need three pieces of data to identify the repository and to authenticate a user on the repository.

- repository name
- user name of a user with Create Cabinet privileges
- user password

The repository name must be the name of a repository accessible to the DFS server. The list of available repositories is maintained by a *connection broker* (often still referred to as docbroker).

Note: DFS knows where the connection broker is, because the IP address or host name of the machine hosting the connection broker is specified in the dfc.properties file stored in the DFS EAR file. If you are using the DFS server deployed with Content Server and running under the Java Method Server,

then DFS will have been configured to point to the connection broker running on that Content Server installation. If the EAR or WAR file was manually deployed to an application server, the connection broker host and port should have been set manually as part of the deployment procedure. For more details, see the *Documentum Foundation Services Deployment Guide*.

Verify your .NET requirements

This guide assumes that you are using Visual Studio 2008 Professional. Make sure to check the current *Documentum Foundation Services Release Notes* to verify .NET requirements for the DFS SDK. The sample .NET projects on the SDK are targeted to .NET 3.0.

Setting up the .NET solution

1. Open the DotNetDocSamples.sln file in Visual Studio

   The solution contains three projects:

   - DotNetDocSamples. This project contains the sample objects and methods included in the Documentum Foundation Services Development Guide.
   - DotNetSampleRunner. This project provides a couple of easy ways to run the samples (see Set up and run the documentation samples, page 63).
   - HelloDFS. This project is a freestanding, self-contained DFS consumer that demonstrates what you need to do (at minimum) to invoke a DFS service.

2. In all three projects in the solution, replace the following references with references to the corresponding assemblies on the DFS SDK (in emc-dfs-sdk-6.7\lib\dotnet).
   - Emc.Documentum.FS.DataModel.Bpm
   - Emc.Documentum.FS.DataModel.CI
   - Emc.Documentum.FS.DataModel.Collaboration
   - Emc.Documentum.FS.DataModel.Core
   - Emc.Documentum.FS.DataModel.Shared
   - Emc.Documentum.FS.Runtime
   - Emc.Documentum.FS.Services.Bpm
   - Emc.Documentum.FS.Services.CI
   - Emc.Documentum.FS.Services.Collaboration
   - Emc.Documentum.FS.Services.Core
   - Emc.Documentum.FS.Services.Search

Examine and run the HelloDFS project

The HelloDFS project is a minimal self-contained test application (all settings are hard-coded in the test client source) that you can use to verify your environment. Examining the source code in this project will give you some basic information about consuming DFS services using the .NET productivity layer.

Examine QueryServiceTest.cs

In Visual Studio, open and examine the source file QueryServiceTest.cs, which is the sole class in the HelloDFS project. The QueryServiceTest class creates a service context, invokes the DFS QueryService, and prints the results of the query to the console.

Building a service context

The private method getSimpleContext returns a ServiceContext, which the DFS .NET client runtime uses to encapsulate and process data that is passed in the SOAP header to the remote service. At minimum the ServiceContext needs to contain an identity consisting of a repository name, user name, and user password, that will enable the DFS server-side runtime to connect to and authenticate on a repository.

```
/*
* This routine returns up a service context
* which includes the repository name and user credentials
*/
private IServiceContext getSimpleContext()
{
   /*
    * Get the service context and set the user
    * credentials and repository information
    */
    ContextFactory contextFactory = ContextFactory.Instance;
    IServiceContext serviceContext = contextFactory.NewContext();
    RepositoryIdentity repositoryIdentity =
        new RepositoryIdentity(repository, userName, password, "");
    serviceContext.AddIdentity(repositoryIdentity);
    return serviceContext;
}
```

When the QueryService is invoked, the DFS client-side runtime will serialize data from the local ServiceContext object and pass it over the wire as a SOAP header like the one shown here:

```
<s:Header>
    <ServiceContext token="temporary/127.0.0.1-1205168560578-476512254"
                    xmlns="http://context.core.datamodel.fs.documentum.emc.com/">
      <Identities xsi:type="RepositoryIdentity"
                  userName="MyUserName"
                  password="MyPassword"
                  repositoryName="MyRepositoryName"
                  domain=""
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/>
      <RuntimeProperties/>
   </ServiceContext>
</s:Header>
```

Examine the CallQueryService method

The CallQueryService method does most of the work in the sample class, including the essential piece of invoking the remote service through the service proxy object.

It begins by instantiating an IQueryService object representing the remote service (the service proxy). This object encapsulates the service context described in the preceding section.

```
/*
 * Get an instance of the QueryService by passing
 * in the service context to the service factory.
 */
 ServiceFactory serviceFactory = ServiceFactory.Instance;
 IServiceContext serviceContext = getSimpleContext();
 IQueryService querySvc
     = serviceFactory.GetRemoteService<IQueryService>(serviceContext,
       moduleName, address);
```

Next, CallQueryService constructs two objects that will be passed to the Execute method: a PassthroughQuery object that encapsulates a DQL statement string, and a QueryExecution object, which contains service option settings. Both objects will be serialized and passed to the remote service in the SOAP body.

```
/*
 * Construct the query and the QueryExecution options
 */
 PassthroughQuery query = new PassthroughQuery();
 query.QueryString = "select r_object_id, object_name from dm_cabinet";
 query.AddRepository(repository);
 QueryExecution queryEx = new QueryExecution();
 queryEx.CacheStrategyType = CacheStrategyType.DEFAULT_CACHE_STRATEGY;
```

CallQueryService then calls the Execute method of the service proxy, which causes the runtime to serialize the data passed to the proxy Execute method, invoke the remote service, and receive a response via HTTP.

```
/*
  * Execute the query passing in operation options
  * This sends the SOAP message across the wire
  * Receives the SOAP response and wraps the response in the
  * QueryResult object
  */
  OperationOptions operationOptions = null;
  QueryResult queryResult = querySvc.Execute(query, queryEx, operationOptions);
```

The complete SOAP message passed to the service endpoint is shown here:

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Header>
    <ServiceContext token="temporary/127.0.0.1-1205239338115-25203285"
      xmlns="http://context.core.datamodel.fs.documentum.emc.com/">
      <Identities xsi:type="RepositoryIdentity"
                  userName="MyUserName"
                  password="MyPassword"
                  repositoryName="MyRepositoryName"
                  domain=""
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/>
      <RuntimeProperties/>
    </ServiceContext>
```

```
  </s:Header>
  <s:Body xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <execute xmlns="http://core.services.fs.documentum.emc.com/">
      <query xsi:type="q1:PassthroughQuery"
             queryString="select r_object_id, object_name from dm_cabinet"
             xmlns=""
             xmlns:q1="http://query.core.datamodel.fs.documentum.emc.com/">
        <q1:repositories>techpubs</q1:repositories>
      </query>
      <execution startingIndex="0"
                 maxResultCount="100"
                 maxResultPerSource="50"
                 cacheStrategyType="DEFAULT_CACHE_STRATEGY"
                 xmlns=""/>
    </execute>
  </s:Body>
</s:Envelope>
```

The remainder of the CallQueryService method examines the QueryResult and prints information about its contents to the console.

Configure and run QueryServiceTest in Visual Studio

To run QueryServiceTest:

1. Open QueryServiceTest.cs source file and specify valid hard-coded values for the following fields.
   - repository
   - userName
   - password
   - address

   ```
   /*************************************************************
    * You must supply valid values for the following fields: */

   /* The repository that you want to run the query on */
   private String repository = "MyRepositoryName";

   /* The username to login to the repository */
   private String userName = "MyUserName";

   /* The password for the username */
   private String password = "MyUserPassword";

   /* The address where the DFS service endpoints are located */
   private String address = "http://HostName:PortNumber/services";

   /*************************************************************/
   ```

   To specify the service endpoint address, replace HostName with the IP address or host name of the machine where DFS is deployed, and replace PortNumber with the port number where the DFS application is deployed. For example, if DFS is deployed with Content Server on a machine called MyContentServer, the value of *address* will be

   ```
   http://MyContentServer:9080/services
   ```

If DFS was deployed on its own tier, the port name will depend on the deployment environment, typically port 8080:

```
http://localhost:8080/services
```

2. Display the Visual Studio Output window (**View**, **Output**).
3. In the Solution Explorer window, right-click on the HelloDFS project and choose **Debug**, **Start New Instance**.
4. If the sample executes successfully, the output window should show, among other things, a list of the names and object identities of the cabinets in the test repository. The first time run of the sample will be significantly slower than subsequent runs.

Set up and run the documentation samples

This section will tell you how to set up and run the DFS documentation samples that are packaged in the DFS SDK. You may find it useful to walk through some of these samples in the debugger and examine objects to understand how the samples work at a detailed level.

The documentation samples proper (that is, the ones you will see in the this Development Guide) are all in the DfsDotNetSamples project. The DotNetSampleRunner project provides a way of running the samples, including some support for creating and deleting sample data on a test repository. Methods in the DotNetSampleRunner project set up expected repository data, call the sample methods, passing them appropriate values, then remove the sample data that was initially set up.

For some samples (specifically the LifeCycleService samples) you will need to install additional objects on the repository using Composer. The objects that you need are provided on the SDK as a Composer project file in emc-dfs-sdk-6.7\samples\DfsDotNetSamples\Csdata\LifecycleProject.zip.

To set up and run the samples, follow these steps, which are detailed in the sections that follow.

1. Install sample lifecycle data in the repository, page 63
2. Configure the DFS client runtime, page 64.
3. Set hard-coded values in TestBase.cs, page 65
4. Optionally set sample data options, page 65
5. Run the samples in the Visual Studio debugger, page 66

Install sample lifecycle data in the repository

To run the LifecycleService samples, you will need to install the Composer project (or the DAR file contained in the project) provided in the DfsDotNetSamples\Csdata folder. This step is not required to run the other samples.

Configure the DFS client runtime

The documentation samples depend on default configuration settings read at program startup from the App.config file in the DotNetSampleRunner project. You need to change only the host IP address or DNS name and port number in the <ContextRoot> element to point to your DFS deployment. The following is a fragment of the config file, showing the elements that you will need to edit.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <sectionGroup name="Emc.Documentum">
      <sectionGroup name="FS">
        <section name="ConfigObject"
type="Emc.Documentum.FS.Runtime.Impl.Configuration.XmlSerializerSectionHandler,
Emc.Documentum.FS.Runtime"/>
      </sectionGroup>
    </sectionGroup>
  </configSections>
  <Emc.Documentum>
    <FS>
      <ConfigObject
type="Emc.Documentum.FS.Runtime.Impl.Configuration.ConfigObject,
Emc.Documentum.FS.Runtime"
          defaultModuleName="core"
          registryProviderModuleName="core">
        <ModuleInfo name="core"
                    protocol="http"
                    host="MY_DFS_HOST"
                    port="MY_PORT"
                    contextRoot="services"/>
        <ModuleInfo name="search"
                    protocol="http"
                    host="MY_DFS_HOST"
                    port="MY_PORT"
                    contextRoot="services"/>
        <ModuleInfo name="bpm"
                    protocol="http"
                    host="MY_DFS_HOST"
                    port="MY_PORT"
                    contextRoot="services"/>
        <ModuleInfo name="collaboration"
                    protocol="http"
                    host="MY_DFS_HOST"
                    port="MY_PORT"
                    contextRoot="services" />
      </ConfigObject>
.
.
.
```

Set hard-coded values in TestBase.cs

For simplicity TestBase.cs hard-codes data that is needed for accessing the test repository. You need to set valid values for the following three variables:

- defaultRepository
- userName
- password

If you have a second repository available, you can also set a valid value for secondaryRepository, otherwise leave it set to null. The secondary repository is used in only one sample, which demonstrates how to copy and object from one repository to another. The secondary repository must be accessible by using the same login credentials that are used for the default repository.

```
// TODO: You must supply valid values for the following variables
private string defaultRepository = "YOUR_REPOSITORY";
private string userName = "YOUR_USER_NAME";
private string password = "YOUR_USER_PASSWORD";

// set this to null if you are not using a second repository
// a second repository is required only to demonstrate
// moving an object across repositories
// the samples expect login to both repositories using the
// same credentials
private string secondaryRepository = null;
```

For more information see Verify repository and login information, page 35.

Optionally set sample data options

There are a couple of settings you may want to change in SampleContentManager.cs before running the samples.

The sample runner removes any sample data that it created from the repository after each sample is run. If you want to leave the data there so you can see what happened on the repository, set isDataCleanedUp to false.

```
// set the following to false if you want to preserve sample data
// bear in mind this may lead to duplicate file errors if you run multiple samples
private bool isDataCleanedUp = false;
```

If you do this, you should delete the created sample data yourself after running each sample to avoid errors related to duplicate object names when running successive samples.

If more than one client is going to test the samples against the same repository, you should create a unique name for the test cabinet that gets created on the repository by the sample runner. Do this by changing the testCabinetPath constant (for example, by replacing XX with your initials).

```
// if multiple developers are testing the samples on the same repository,
// create a unique name testCabinetPath to avoid conflicts
public const String testCabinetPath = "/DFSTestCabinetXX";
```

Run the samples in the Visual Studio debugger

To run the samples:

1. Open Program.cs in the SampleRunner project.
2. In the main method, optionally comment out any sets of samples that you do not want to run. In particular, comment out RunLifecycleServiceDemos if you have not installed the lifecycle sample data in the repository. You can also comment out individual sample methods in the subroutines called by the Main method for more granular tests. The sample methods do not depend on previous sample methods and can be run in any order.

```
static void Main()
{
    RunQueryServiceDemos();
    RunObjectServiceDemos();
    RunVersionControlServiceDemos();
    RunSchemaServiceDemos();
    RunSearchServiceDemos();
    RunWorkflowServiceDemos();
    RunAccessControlServiceDemos();
    RunLifecycleServiceDemos();
    RunVirtualDocumentServiceDemos();
}
```

3. If you are going to run multiple samples, open SampleContentManager.cs and make sure that isDataCleanedUp is initialized to true. This will prevent duplicate filename errors.
4. Display the Output window.
5. Build and run the DotNetSampleRunner project in debug mode.

Chapter 6

Consuming DFS with the .NET Productivity Layer

The .NET productivity layer is functionally identical to the Java productivity layer, except that the .NET productivity layer supports only remote service invocation. Note that while DFS samples are in C#, the .NET library is CLS compliant and can be used by any CLS-supported language.

This chapter covers the following topics:

Configuring .NET consumer project dependencies

The DFS .NET client library requires .NET 3.0, which includes the Windows Communication Foundation (WCF), Microsoft's unified framework for creating service-oriented applications. For more information see http://msdn2.microsoft.com/en-us/library/ms735119.aspx.

DFS consumer projects will require references to the following assemblies from the DFS SDK:

- Emc.Documentum.FS.DataModel.Core
- Emc.Documentum.FS.DataModel.Shared
- Emc.Documentum.FS.Runtime

In addition, the application may need to reference some of the following assemblies, depending on the DFS functionality that the application utilizes:

- Emc.Documentum.FS.DataModel.Bpm
- Emc.Documentum.FS.DataModel.CI

- Emc.Documentum.FS.DataModel.Collaboration
- Emc.Documentum.FS.Services.Bpm
- Emc.Documentum.FS.Services.CI
- Emc.Documentum.FS.Services.Collaboration
- Emc.Documentum.FS.Services.Core
- Emc.Documentum.FS.Services.Search

Configuring a .NET client

.NET client configuration settings are specified in the consumer application's configuration file (which for Windows Forms clients is app.config), an example of which is shown below.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <sectionGroup name="Emc.Documentum">
      <sectionGroup name="FS">
        <section name="ConfigObject"
                 type="Emc.Documentum.FS.Runtime.Impl.Configuration.
                 XmlSerializerSectionHandler,
                 Emc.Documentum.FS.Runtime"/>
      </sectionGroup>
    </sectionGroup>
  </configSections>
  <Emc.Documentum>
    <FS>
      <ConfigObject type="Emc.Documentum.FS.Runtime.Impl.Configuration.
                                                          ConfigObject,
         Emc.Documentum.FS.Runtime"
                     defaultModuleName="core"
                     registryProviderModuleName="core"
                     requireSignedUcfJars="true">
        <ModuleInfo name="core"
                    protocol="http"
                    host="MY_DFS_HOST"
                    port="MY_PORT"
                    contextRoot="services"/>
        <ModuleInfo name="search"
                    protocol="http"
                    host="MY_DFS_HOST"
                    port="MY_PORT"
                    contextRoot="services"/>
        <ModuleInfo name="bpm"
                    protocol="http"
                    host="MY_DFS_HOST"
                    port="MY_PORT"
                    contextRoot="services"/>
        <ModuleInfo name="collaboration"
                    protocol="http"
                    host="MY_DFS_HOST"
                    port="MY_PORT"
                    contextRoot="services" />
      </ConfigObject>
    </FS>
  </Emc.Documentum>
  <system.serviceModel>
```

```
<bindings>
  <basicHttpBinding>
    <binding name="DfsAgentService"
             closeTimeout="00:01:00"
             openTimeout="00:01:00"
             receiveTimeout="00:10:00"
             sendTimeout="00:01:00"
             allowCookies="false"
             bypassProxyOnLocal="false"
             hostNameComparisonMode="StrongWildcard"
             maxBufferSize="1000000"
             maxBufferPoolSize="10000000"
             maxReceivedMessageSize="1000000"
             messageEncoding="Text"
             textEncoding="utf-8"
             transferMode="Buffered"
             useDefaultWebProxy="true">
      <readerQuotas maxDepth="32"
                    maxStringContentLength="8192"
                    maxArrayLength="16384"
                    maxBytesPerRead="4096"
                    maxNameTableCharCount="16384" />
      <security mode="None">
        <transport clientCredentialType="None"
                      proxyCredentialType="None"
                      realm="" />
        <message clientCredentialType="UserName"
                     algorithmSuite="Default" />
      </security>
    </binding>
    <binding name="DfsContextRegistryService"
             closeTimeout="00:01:00"
             openTimeout="00:01:00"
             receiveTimeout="00:10:00"
             sendTimeout="00:01:00"
             allowCookies="false"
             bypassProxyOnLocal="false"
             hostNameComparisonMode="StrongWildcard"
             maxBufferSize="1000000"
             maxBufferPoolSize="10000000"
             maxReceivedMessageSize="1000000"
             messageEncoding="Text"
             textEncoding="utf-8"
             transferMode="Buffered"
             useDefaultWebProxy="true">
      <readerQuotas maxDepth="32"
                    maxStringContentLength="8192"
                    maxArrayLength="16384"
                    maxBytesPerRead="4096"
                    maxNameTableCharCount="16384" />
      <security mode="None">
        <transport clientCredentialType="None"
                   proxyCredentialType="None"
                   realm="" />
        <message clientCredentialType="UserName"
                  algorithmSuite="Default" />
      </security>
    </binding>
    <binding name="DfsDefaultService"
             closeTimeout="00:01:00"
             openTimeout="00:01:00"
             receiveTimeout="00:10:00"
             sendTimeout="00:01:00"
             allowCookies="false"
             bypassProxyOnLocal="false"
```

```
                hostNameComparisonMode="StrongWildcard"
                maxBufferSize="1000000"
                maxBufferPoolSize="10000000"
                maxReceivedMessageSize="1000000"
                messageEncoding="Text"
                textEncoding="utf-8"
                transferMode="Buffered"
                useDefaultWebProxy="true">
          <readerQuotas maxDepth="32"
                        maxStringContentLength="8192"
                        maxArrayLength="16384"
                        maxBytesPerRead="4096"
                        maxNameTableCharCount="16384" />
          <security mode="None">
            <transport clientCredentialType="None"
                       proxyCredentialType="None"
                       realm="" />
            <message clientCredentialType="UserName"
                     algorithmSuite="Default" />
          </security>
        </binding>
      </basicHttpBinding>
    </bindings>
  </system.serviceModel>
</configuration>
```

The configuration file contains settings that are DFS-specific, as well as settings that are WCF-specific, but which impact DFS behavior. The DFS-specific settings are those within the <Emc.Documentum> <FS> tags. The remaining settings (within <basicHttpBinding>) are specific to Microsoft WCF.

For documentation of the Microsoft settings refer to http://msdn.microsoft.com/en-us/library/ms731361.aspx.

The ConfigObject section includes the following DFS-specific attributes:

- defaultModuleName—the module name to use if no module is specified in the service instantiation method
- registryProviderModuleName—the module that includes the ContextRegistryService (under normal circumstances leave this set to “core”)
- requireSignedUcfJars—sets whether the .NET runtime requires that UCF-related JAR files downloaded from the DFS server be signed; default is “true”; normally this is not needed, but it must be set to false if the client runtime is version 6.5 or higher and the service runtime is version 6 (which does not have signed UCF JARs).

The ModuleInfo elements have properties that together describe the address of a module (and of the services at that address), using the following attributes:

- protocol—either http or https, depending on whether the application server is configured to use SSL.
- host—the DNS name or IP address of the service host.
- port—the port number at which the DFS application server listens. When DFS is installed with Content Server, the port defaults to 9080.
- contextRoot—the root address under which service modules are organized; the contextRoot for DFS-provided services is “services”
- name—the name of the service module, under which a set of related services are organized

The fully-qualified service address is constructed as runtime as follows:

```
<protocol>://<host>:<port>/<contextRoot>/<module>/<serviceName>
```

For example:

```
http://dfsHostName:8080/services/core/ObjectService
```

Setting MaxReceivedMessageSize for .NET clients

DFS exceptions can sometimes result in SOAP messages that exceed size limits defined for the .NET consumer. The DFS SDK provides an app.config in which the default values are increased across all declared bindings as follows:

- maxBufferSize is increased to 1000000 instead of default 65536
- maxBufferPoolSize is increased to 1000000 instead of default 524288
- maxReceivedMessageSize is increased to 1000000 instead of default 65536

Use these values or similar values in your .NET consumer.

Beware that the app.config provided with the SDK is oriented toward productivity-layer consumers. In productivity-layer-oriented app.config, the DfsDefaultService binding acts as the configuration for all DFS services, except for DFS runtime services (the AgentService and ContextRegistryService), which have separate, named bindings declared. The following sample shows the DfsDefaultService binding as delivered with the SDK.

```
<binding name="DfsDefaultService"
         closeTimeout="00:01:00"
         openTimeout="00:01:00"
         receiveTimeout="00:10:00"
         sendTimeout="00:01:00"
         allowCookies="false"
         bypassProxyOnLocal="false"
         hostNameComparisonMode="StrongWildcard"

         maxBufferSize="1000000"
         maxBufferPoolSize="10000000"
         maxReceivedMessageSize="1000000"

         messageEncoding="Text"
         textEncoding="utf-8"
         transferMode="Buffered"
         useDefaultWebProxy="true">

  <readerQuotas maxDepth="32"
                maxStringContentLength="8192"
                maxArrayLength="16384"
                maxBytesPerRead="4096"
                maxNameTableCharCount="16384" />
  <security mode="None">
    <transport clientCredentialType="None"
               proxyCredentialType="None"
               realm="" />
    <message clientCredentialType="UserName" algorithmSuite=
                                            "Default" />
  </security>
</binding>
```

A WSDL-based consumer by default introduces a per-service binding application configuration file into the overall solution, in this case you have to configure the binding for every service. The following example shows the declaration generated for the ObjectServicePortBinding with default values:

```
<binding name="ObjectServicePortBinding"
         closeTimeout="00:01:00"
         openTimeout="00:01:00"
         receiveTimeout="00:10:00"
         sendTimeout="00:01:00"
         allowCookies="false"
         bypassProxyOnLocal="false"
         hostNameComparisonMode="StrongWildcard"
         maxBufferSize="65536"
         maxBufferPoolSize="524288"
         maxReceivedMessageSize="65536"
         messageEncoding="Text"
         textEncoding="utf-8"
         transferMode="Buffered"
         useDefaultWebProxy="true">
  <readerQuotas maxDepth="32"
                maxStringContentLength="8192"
                maxArrayLength="16384"
                maxBytesPerRead="4096"
                maxNameTableCharCount="16384" />
  <security mode="None">
    <transport clientCredentialType="None"
               proxyCredentialType="None"
               realm="" />
    <message clientCredentialType="UserName"
             algorithmSuite="Default" />
  </security>
</binding>
```

If you are concerned about preventing users from declaring too small a value for such attributes, programmatically check and override the declared values, as follows:

```
System.Reflection.FieldInfo appConfigInfo = typeof(ContextFactory)
  .GetField("appConfig",
  System.Reflection.BindingFlags.Instance | System.Reflection.
                                        BindingFlags.NonPublic);
System.Reflection.FieldInfo agentServiceBindingInfo = typeof(AppConfig)
                              .GetField("m_agentServiceBinding",
  System.Reflection.BindingFlags.Instance | System.Reflection.
                                     BindingFlags.NonPublic);
System.Reflection.FieldInfo contextRegistryServiceBindingInfo =
   typeof(AppConfig).GetField
   ("m_contextRegistryServiceBinding",
  System.Reflection.BindingFlags.Instance | System.Reflection.
                                  BindingFlags.NonPublic);
System.Reflection.FieldInfo defaultServiceBindingInfo =
typeof(AppConfig).GetField("m_defaultServiceBinding",
System.Reflection.BindingFlags.Instance | System.Reflection.
       BindingFlags.NonPublic);
BasicHttpBinding binding = new BasicHttpBinding();
binding.MaxReceivedMessageSize = 0x7fffffffL;
binding.MaxBufferSize = 0x7fffffffL;
agentServiceBindingInfo.SetValue(appConfigInfo.
GetValue(contextFactory), binding);
contextRegistryServiceBindingInfo.SetValue(appConfigInfo.
      GetValue(contextFactory),binding);
defaultServiceBindingInfo.SetValue(appConfigInfo.
GetValue(contextFactory), binding);
```

Creating a service context in .NET

Service invocation in DFS takes place within a service context, which is an object that maintains identity information for service authentication, profiles for setting options and filters, a locale, and properties. Service contexts can be shared among multiple services.

Please also note that the service context is not thread safe and should not be accessed by separate threads in a multi-threaded application. If you require multiple threads your application must provide explicit synchronization.

Service context is represented in the client object model by the IServiceContext interface, instances of which encapsulate information that is passed in the SOAP header to the service endpoint during service registration and/or service invocation.

If a service context is registered, it is stored on the DFS server and represented by a token that is passed in the SOAP header during service invocation, along with optional service context data that is treated as a delta and merged into the existing service context. If a service is unregistered, the context is stored in a client object and the complete service context is passed in the SOAP header with each service invocation. There are advantages and disadvantages to both approaches; for more information see Service context registration, page 76.

Properties and profiles can often be passed to an operation during service operation invocation through an OperationOptions argument, as an alternative to storing properties and profiles in the service context, or as a way of overriding settings stored in the service context. OperationOptions settings are passed in the SOAP body, rather than the SOAP header.

Setting up service context (.NET)

To be used by a service that requires authentication, the service context should be populated with at least one identity. The following sample creates and returns a minimal service context that contains a ContentTransferProfile:

Example 6-1. C#: Initializing service context

```
private void initializeContext()
{
    ContextFactory contextFactory = ContextFactory.Instance;
    serviceContext = contextFactory.NewContext();

    RepositoryIdentity repoId = new RepositoryIdentity();
    RepositoryIdentity repositoryIdentity =
    new RepositoryIdentity(DefaultRepository, UserName, Password, "");
    serviceContext.AddIdentity(repositoryIdentity);

    ContentTransferProfile contentTransferProfile =
                                    new ContentTransferProfile();
    contentTransferProfile.TransferMode = ContentTransferMode.MTOM;
    serviceContext.SetProfile(contentTransferProfile);
}
```

Identities

A service context contains a collection of identities, which are mappings of repository names onto sets of user credentials used in service authentication. A service context is expected to contain only one identity per repository name. Identities are set in a service context using one of the concrete Identity subclasses:

- BasicIdentity directly extends the Identity parent class, and includes accessors for user name and password, but not for repository name. This class can be used in cases where the service is known to access only a single repository, or in cases where the user credentials in all repositories are known to be identical. BasicIdentity can also be used to supply fallback credentials in the case where the user has differing credentials on some repositories, for which RepositoryIdentity instances will be set, and identical credentials on all other repositories. Because BasicIdentity does not contain repository information, the username and password is authenticated against the global registry. If there is no global registry defined, authentication fails.
- RepositoryIdentity extends BasicIdentity, and specifies a mapping of repository name to a set of user credentials, which include a user name, password, and optionally a domain name if required by your network environment. In a RepositoryIdentity, you can use the "*" wildcard (represented by the constant RepositoryIdentity.DEFAULT_REPOSITORY_NAME) in place of the repository name. In this case DFS will authorize the credentials against the global registry. If no global registry is available, or if the credentials are not valid on the global registry, the authentication fails. Using this wildcard in a RepositoryIdentity is essentially the same as using a BasicIdentity.
- SsoIdentity allows an SSO solution to be used to authenticate the user. You can use SsoIdentity class when the service that you are requesting is accessing only one repository, or if the user credentials in all repositories are identical. Because SsoIdentity does not contain repository information, the username and password is authenticated against the designated global registry. If there is no global registry defined, authentication fails.
- BinaryIdentity is used only in a local Java client and is not serializable. It is used to encapsulate Kerberos credentials (see Kerberos authentication in a local DFS web application, page 206).
- PrincipalIdentity is used to indicate that DFC principal mode login should be used with the user name provided in the identity instance. PrincipalIdentity is not XML serializable, so it will not be sent over the wire. For security reasons, it will work only when the DFS service is invoked in local mode.

Locale

The locale property of an IServiceContext object specifies the language and optionally country setting to use for locale-sensitive features. The locale is used, for example, to control which NLS-specific Data Dictionary strings will be provided by Content Server to the DFS layer. The format of the locale string value is based on Java locale strings, which in turn are based on ISO 639-1 two-character, lowercase language codes and ISO 3166 country codes. The format of a Java locale string is <languagecode>[_<countrycode>]; for example, the Java locale string for British English is "en_GB". Further information on Java locale strings can be found at http://java.sun.com/developer/technicalArticles/J2SE/locale/.

To find out the locale codes currently supported by Content Server at installation, refer to the documentation for the locale_name property of the dm_server_config object in the *EMC Documentum System Object Reference Manual.*

If the locale is not set in the service context, the DFS server runtime will use the value set in the DFS server application. Typically this means that a DFS client (particularly a remote client) should set the locale to the locale expected by the user, rather than relying on the value set on the server. The locale setting used by the DFS server can be specified in the dfc.locale property of dfc.properties. If the value is not set in the service context by the client and not set on the server, the DFS server will use the locale of the JVM in which it is running.

Service context runtime properties

A service context contains a RuntimeProperties collection, in which properties can be set for all services sharing the service context. These properties settings can be used to store configuration settings that are scoped to the service context, and therefore are not suitable as operation parameters or inclusion in the OperationOptions PropertySet. Properties included in RuntimeProperties would generally be standalone properties. DFS services generally use profiles in preference to RuntimeProperties. The following table lists the properties that you can set in the service context.

Table 12. Service context properties

Property Name	Description
“dfs.exception.include_stack_trace”	A value of true indicates that a Java stack trace needs to be included in the serialized DFS exception.
IServiceContext.OVERRIDE_WSDL_ ENDPOINT_ADDRESS	If set to FALSE the SOAP client uses the URL returned by the web service in its WSDL as the service address and not the one provided by the service developer. Has to be set to true if the service’s address is not publicly available.
IServiceContext.PAYLOAD_PROCESSING_ POLICY	Sets whether or not to continue execution on an exception. Values can be “PAYLOAD_ FAIL_ON_EXCEPTION” or “PAYLOAD_ CONTINUE_ON_EXCEPTION” .
IServiceContext.USER_TRANSACTION_HINT	If set to IServiceContext.TRANSACTION_ REQUIRED, attempts to run all "nested" calls in a single transaction, subject to the support provided by DFC. Valid values are IServiceContext.TRANSACTION_REQUIRED and IServiceContext.TRANSACTION_NOT_ REQUIRED.

The PAYLOAD_PROCESSING_POLICY property is used when transferring multiple content objects to control whether the transfer will continue processing if the upload or download of a content object fails.

Service context registration

Context registration is an optional technique for optimizing how much data is sent over the wire by remote DFS consumers. It is available for remote web services consumers, but does not apply to local Java consumers because the consumer and service share the same JVM. When you register a service context within a consumer, the DFS server-side runtime creates and maintains the service context on the server.

There are two benefits to registering the service context. The first benefit is that services can share a registered context. This minimizes over the wire traffic since the consumer does not have to send service context information to every service it calls.

The second benefit occurs when a consumer calls a service and passes in a *delta modification* to the service context. The DFS client runtime figures out the minimal amount of data to send over the wire (the modifications) and the server runtime merges the delta modifications into the service context that is stored on the server. If your application is maintaining a lot of data (such as profiles, properties, and identities) in the service context, this can significantly reduce how much data is sent with each service call, because most of the data can be sent just once when the service context is registered. On the other hand, if your application is storing only a small amount of data in the service context, there is really not much to be gained by registering the service context.

You should be aware that there are limitations that result from registration of service context.

- The service context can be shared only by services that share the same classloader. Typically this means that the services are deployed in the same EAR file on the application server. This limitation means that the client must be aware of the physical location of the services that it is invoking and manage service context sharing based on shared physical locations.
- Registration of service contexts prevents use of failover in clustered DFS installations.
- Registration of the service context is not supported with identities that store Kerberos credentials.

If you are using the DFS client productivity layer, registering a service context is mostly handled by the runtime, with little work on your part. You start by creating a service context object, then you call one of the overloaded register methods.

Table 13. Methods for registering services (Java)

ContextFactory method	Description
register(IServiceContext serviceContext)	Registers service context at default ContextRegistryService which is set in the registryProviderModuleName attribute in dfs-client.xml.
register(IServiceContext serviceContext, serviceModule)	Registers service context at the ContextRegistryService located in the specified serviceModule. The full address of the service module is looked up in dfs-client.xml by module name. In this case the module name in dfs-client.xml must be unique.
register(IServiceContext serviceContext, String serviceModule, String contextRoot)	Registers service context at the ContextRegistryService located in the specified serviceModule and contextRoot.

Table 14. Methods for registering services (.NET)

ContextFactory method	Description
Register(IServiceContext serviceContext)	Registers service context at default ContextRegistryService which is set in the registryProviderModuleName attribute in the configuration file.
Register(IServiceContext serviceContext, serviceModule)	Registers service context at the ContextRegistryService located in the specified serviceModule. The full address of the service module is looked up in the configuration file by module name. In this case the module name in the configuration file must be unique.
Register(IServiceContext serviceContext, String serviceModule, String contextRoot)	Registers service context at the ContextRegistryService located in the specified serviceModule and contextRoot.

If you wish to register the service context and are not using the productivity layer, you can register the context by invoking the ContextRegistry service directly (see Writing a consumer that registers the service context, page 24).

The register method can only be executed remotely and is meaningless in a local Java service client. If you are running your client in local mode, the register method will still result in an attempt at remote invocation of ContextRegistryService. If the remote invocation fails, an exception will be thrown. If the invocation succeeds (because there is a remote connection configured and available), there will be a harmless invocation of the remote service.

Instantiating a service in .NET

A .NET client (or a service) can create an instance of a service using one of several methods of ServiceFactory. These generic factory methods return service objects that allow the service address to be explicitly provided, or obtained by lookup from the application configuration file.

Table 15. Methods for instantiating services

ServiceFactory method	Description
GetRemoteService<T>(IServiceContext serviceContext)	Instantiates service proxy. In this method neither the module name nor the context root is specified, so the service address is looked up in the configuration file based on the defaultModuleName.

ServiceFactory method	Description
GetRemoteService<T>(IServiceContext serviceContext, String serviceModule)	Instantiates service proxy, with module name explicitly provided. The service address will be looked up in the configuration file based on the provided module name, therefore the module name must be unique in the configuration file.
GetRemoteService<T>(IServiceContext serviceContext, String serviceModule, String contextRoot)	Instantiates service proxy using explicit service addressing with contextRoot and module name explicitly provided. If null is passed in contextRoot, then the service address will be looked up based on the provided module name. In this case the module name must be unique in the configuration file. If both serviceModule and serviceContext are null, then the lookup is based on the defaultModuleName in the configuration file. If the serviceModule is null and serviceContext is not null, then the service is assumed to have no module name (that is, its address is contextRoot). Note that contextRoot is fully-qualified, and includes the protocol, host, and port: for example “http://localhost:8080/services”.

Transaction support

DFS provides basic support for transactions. If transactions are enabled operations will use implicit transactional behavior; that is, they will begin the transaction at the start of the operation and commit the transaction at the end of the operation if it completes successfully. If the any part of the operation fails, the entire operation will be rolled back.

To enable transactions, set the USER_TRANSACTION_HINT runtime property in the service context to IServiceContext.TRANSACTION_REQUIRED.

Combining USER_TRANSACTION_HINT and PAYLOAD_PROCESSING_POLICY

Note: Please note that PAYLOAD_PROCESSING_POLICY may be deprecated in a future release.

Transactional behavior for a service operation is enabled by setting the USER_TRANSACTION_HINT runtime property in the service context. It is possible to combine this setting with PAYLOAD_CONTINUE_ON_EXCEPTION, as shown here:

```
context.SetRuntimeProperty(IServiceContext.USER_TRANSACTION_HINT,
                           IServiceContext.TRANSACTION_REQUIRED);
context.SetRuntimeProperty(IServiceContext.PAYLOAD_PROCESSING_POLICY,
                           IServiceContext.PAYLOAD_CONTINUE_ON_EXCEPTION);
```

The expected behavior is that the payload policy must be honored first, then the transaction policy. For example, suppose that we use the Object service to create objects based on a DataPackage that has two DataObject trees. We use PAYLOAD_CONTINUE_ON_EXCEPTION with transaction support to create the objects. At runtime, a leaf in the first DataObject tree fails and all others succeed. In this case only the objects in the second DataObject tree would be created; the creation of the first

DataObject tree would be rolled back. If no transaction support were used, some leaves from the first DataObject tree would be created, as well as the entire second DataObject tree.

OperationOptions

DFS services generally take an OperationOptions object as the final argument when calling a service operation. OperationOptions contains profiles and properties that specify behaviors for the operation. The properties have no overlap with properties set in the service context's RuntimeProperties. The profiles can potentially overlap with properties stored in the service context. In the case that they do overlap, the profiles in OperationOptions always take precedence over profiles stored in the service context. The profiles stored in the service context take effect when no matching profile is stored in the OperationOptions for a specific operation. The override of profiles in the service context takes place on a profile-by-profile basis: there is no merge of specific settings stored within the profiles.

As a recommended practice, a service client should avoid storing profiling information or properties in the service operation that are likely to be modified by specific service instances. This avoids possible side-effects caused by modifications to a service context shared by multiple services. It is likely that ContentTransferProfile will not change and so should be included in the service context. Other profiles are better passed within OperationOptions.

OperationOptions are discussed in more detail under the documentation for specific service operations. For more information on core profiles, see PropertyProfile, page 93, ContentProfile, page 97, PermissionProfile, page 101, and RelationshipProfile, page 110. Other profiles are covered under specific services in the *Enterprise Content Services Reference.*

Handling SOAP faults in the .NET productivity layer

The DFS .NET productivity layer wherever possible avoids manipulation of serialized SOAP fault data. Specifically, it does not translate a FaultException representing a SOAP fault into a ServiceException. Instead, serialized exception details in the service response are wrapped in a FaultException<T> and thrown on the client. T in this case is a type representing SOAP fault data that implements System.Xml.Serialization.IXmlSerializable, such as ServiceException. These service exceptions representing DFS SOAP faults are not native C# exceptions and do not extend the Exception class. Therefore they are not themselves thrown, but are instead are thrown as FaultException<ServiceException>, FaultException<CoreServiceException>, and so on.

The following shows a sample of catching such exceptions:

```
public void TestCreateObjectWithUnknownType()
{
    ObjectService m_service = GetCoreService<IObjectService>();
    DataObject testObj = new DataObject(new ObjectIdentity
        getDefaultRepository()), "invalid_type");
    try
    {
        m_service.Create(new DataPackage(testObj), null);
        Assert.Fail("Should Fail to create object with unknown type");
    }
```

```
    catch (FaultException<SerializableException> ex)
    {
    Console.WriteLine(String.Format("Got expected FaultException[{0}]
        with message: {1}\n", ex.Detail, ex.Message));
    }
    catch (Exception exx)

    {
        Console.WriteLine(exx.StackTrace);
        Assert.Fail("Unexpected Exception: " + exx.GetType().ToString());
    }
}
```

To obtain detailed information about the exception stack, examine the Detail.Nodes property of the exception, which is an array of XmlNode objects. The following shows a node from the SerializableException instance caught in the preceding example:

```
<exceptionBean>
  <attribute>
    <name>messageId</name>
    <type>java.lang.String</type>
    <value>E_CREATING_OBJECT_TREE_FAILED_WITH_TREE_DUMP</value>
  </attribute>
  <exceptionClass>com.emc.documentum.fs.services.core.CoreServiceException
  </exceptionClass>
  <genericType>java.lang.Exception</genericType>
  <message>Create operation failed for object:[id =null] PROPERTIES</message>
  <messageId>E_CREATING_OBJECT_TREE_FAILED_WITH_TREE_DUMP</messageId>
</exceptionBean>
```

Chapter 7

DFS Data Model

The DFS data model comprises the object model for data passed to and returned by Enterprise Content Services. This chapter covers fundamental aspects of the data model and important concepts related to it. This chapter is a supplement to the API documentation, which provides more comprehensive coverage of DFS classes. This topic covers the following topics:

DataPackage

The DataPackage class defines the fundamental unit of information that contains data passed to and returned by services operating in the DFS framework. A DataPackage is a collection of DataObject instances, which is typically passed to, and returned by, Object service operations such as create, get, and update. Object service operations process all the DataObject instances in the DataPackage sequentially.

DataPackage example

The following sample instantiates, populates, and iterates through a data package.

Example 7-1. Java: DataPackage

Note that this sample populates a DataPackage twice, first using the addDataObject convenience method, then again by building a list then setting the DataPackage contents to the list. The result is that the DataPackage contents are overwritten; but the purpose of this sample is to simply show two different ways of populating the DataPackage, not to do anything useful.

```
DataObject dataObject = new DataObject(new ObjectIdentity("myRepository"));
DataPackage dataPackage = new DataPackage(dataObject);


// add a data object using the add method
DataObject dataObject1 = new DataObject(new ObjectIdentity("myRepository"));
dataPackage.addDataObject(dataObject1);


//build list and then set the DataPackage contents to the list
ArrayList<DataObject> dataObjectList = new ArrayList<DataObject>();
dataObjectList.add(dataObject);
dataObjectList.add(dataObject1);
dataPackage.setDataObjects(dataObjectList);

for (DataObject dataObject2 : dataPackage.getDataObjects())
{
    System.out.println("Data Object: " + dataObject2);
}
```

Example 7-2. C#: DataPackage

```
DataObject dataObject = new DataObject(new ObjectIdentity("myRepository"));
DataPackage dataPackage = new DataPackage(dataObject);

DataObject dataObject1 = new DataObject(new ObjectIdentity("myRepository"));
dataPackage.AddDataObject(dataObject1);

foreach (DataObject dataObject2 in dataPackage.DataObjects)
{
    Console.WriteLine("Data Object: " + dataObject2);
}
```

DataObject

A DataObject is a representation of an object in an ECM repository. In the context of EMC Documentum technology, the DataObject functions as a DFS representation of a persistent repository object, such as a dm_sysobject or dm_user. Enterprise Content Services (such as the Object service) consistently process DataObject instances as representations of persistent repository objects.

A DataObject instance is potentially large and complex, and much of the work in DFS service consumers will be dedicated to constructing the DataObject instances. A DataObject can potentially contain comprehensive information about the repository object that it represents, including its identity, properties, content, and its relationships to other repository objects. In addition, the DataObject instance may contain settings that instruct the services about how the client wishes parts of the DataObject to be processed. The complexity of the DataObject and related parts of the data model, such as Profile classes, are design features that enable and encourage simplicity of the service interface and the packaging of complex consumer requests into a minimal number of service interactions.

For the same reason DataObject instances are consistently passed to and returned by services in simple collections defined by the DataPackage class, permitting processing of multiple DataObject instances in a single service interaction.

DataObject related classes

Table 16, page 83 shows object types that can be contained by a DataObject.

Table 16. DataObject related classes

Class	Description
ObjectIdentity	An ObjectIdentity uniquely identifies the repository object referenced by the DataObject. A DataObject can have 0 or 1 identities. For more details see ObjectIdentity, page 84.
PropertySet	A PropertySet is a collection of named properties, which correspond to the properties of a repository object represented by the DataObject. A DataObject can have 0 or 1 PropertySet instances. For more information see Property, page 87.
Content	Content objects contain data about file content associated with the data object. A DataObject can contain 0 or more Content instances. A DataObject without content is referred to as a "contentless DataObject." For more information see Content model and profiles, page 95.
Permission	A Permission object specifies a specific basic or extended permission, or a custom permission. A DataObject can contain 0 or more Permission objects. For more information see Permissions, page 100
Relationship	A Relationship object defines a relationship between the repository object represented by the DataObject and another repository object. A DataObject can contain 0 or more Relationship instances. For more information, see Relationship, page 102.
Aspect	The Aspect class models an aspect that can be attached to, or detached from, a persistent repository object.

DataObject type

A DataObject instance in normal DFS usage corresponds to a typed object defined in the repository. The type is specified in the type setting of the DataObject using the type name defined in the repository (for example dm_sysobject or dm_user). If the type is not specified, services will use an implied type, which is dm_document.

DataObject construction

The construction of DataObject instances will be a constant theme in examples of service usage throughout this document. The following typical example instantiates a DataObject, sets some of its properties, and assigns it some content. Note that because this is a new DataObject, only a repository name is specified in its ObjectIdentity.

Example 7-3. Java: DataObject construction

```
ObjectIdentity objIdentity = new ObjectIdentity(repositoryName);
DataObject dataObject = new DataObject(objIdentity, "dm_document");

PropertySet properties = dataObject.getProperties();
properties.set("object_name", objName);
properties.set("title", objTitle);
properties.set("a_content_type", "gif");

dataObject.getContents().add(new FileContent("c:/temp/MyImage.gif", "gif"));

DataPackage dataPackage = new DataPackage(dataObject);
```

Example 7-4. C#: DataObject construction

```
ObjectIdentity objIdentity = new ObjectIdentity(repositoryName);
DataObject dataObject = new DataObject(objIdentity, "dm_document");

PropertySet properties = dataObject.Properties;
properties.Set("object_name", objName);
properties.Set("title", objTitle);
properties.Set("a_content_type", "gif");

dataObject.Contents.Add(new FileContent("c:/temp/MyImage.gif", "gif"));

DataPackage dataPackage = new DataPackage(dataObject);
```

ObjectIdentity

The function of the ObjectIdentity class is to uniquely identify a repository object. An ObjectIdentity instance contains a repository name and an identifier that can take various forms, described in the following table listing the ValueType enum constants.

ValueType	Description
OBJECT_ID	Identifier value is of type ObjectId, which is a container for the value of a repository r_object_id attribute, a value generated by Content Server to uniquely identify a specific version of a repository object.
OBJECT_PATH	Identifier value is of type ObjectPath, which contains a String expression specifying the path to the object, excluding the repository name. For example /MyCabinet/MyFolder/MyDocument.
QUALIFICATION	Identifier value is of type Qualification, which can take the form of a DQL expression fragment. The Qualification is intended to uniquely identify a Content Server object.
OBJECT_KEY	Identifier value is of type ObjectKey, which contains a PropertySet, the properties of which, joined by logical AND, uniquely identity the repository object.

When constructing a DataObject to pass to the create operation, or in any case when the DataObject represents a repository object that does not yet exist, the ObjectIdentity need only be populated with a repository name. If the ObjectIdentity does contain a unique identifier, it must represent an existing repository object.

Note that the ObjectIdentity class is generic in the Java client library, but non-generic in the .NET client library.

ObjectId

An ObjectId is a container for the value of a repository r_object_id attribute, which is a value generated by Content Server to uniquely identify a specific version of a repository object. An ObjectId can therefore represent either a CURRENT or a non-CURRENT version of a repository object. DFS services exhibit service- and operation-specific behaviors for handling non-CURRENT versions, which are documented under individual services and operations.

ObjectPath

An ObjectPath contains a String expression specifying the path to a repository object, excluding the repository name. For example /MyCabinet/MyFolder/MyDocument. An ObjectPath can only represent the CURRENT version of a repository object. Using an ObjectPath does not guarantee the uniqueness of the repository object, because Content Server does permit objects with identical names to reside within the same folder. If the specified path is unique at request time, the path is recognized as a valid object identity; otherwise, the DFS runtime will throw an exception.

Qualification

A Qualification is an object that specifies criteria for selecting a set of repository objects. Qualifications used in ObjectIdentity instances are intended to specify a single repository object. The criteria set in the qualification is expressed as a fragment of a DQL SELECT statement, consisting of the expression string following "SELECT FROM", as shown in the following example.

```
Qualification qualification =
    new Qualification("dm_document where object_name = 'dfs_sample_image'");
```

DFS services use normal DQL statement processing, which selects the CURRENT version of an object if the ALL keyword is not used in the DQL WHERE clause. The preceding example (which assumes for simplicity that the object_name is sufficient to ensure uniqueness) will select only the CURRENT version of the object named dfs_sample_image. To select a specific non-CURRENT version, the Qualification must use the ALL keyword, as well as specific criteria for identifying the version, such as a symbolic version label:

```
String nonCurrentQual = "dm_document (ALL) " +
                        "where object_name = 'dfs_sample_image' " +
                        "and ANY r_version_label = 'test_version'";
Qualification<String> qual = new Qualification<String>(nonCurrentQual);
```

ObjectIdentity subtype example

The following samples demonstrate the ObjectIdentity subtypes.

Example 7-5. Java: ObjectIdentity subtypes

```
String repName = "MyRepositoryName";
ObjectIdentity[] objectIdentities = new ObjectIdentity[4];

// repository only is required to represent an object that has not been created
objectIdentities[0] = new ObjectIdentity(repName);

// show each form of unique identifier
ObjectId objId = new ObjectId("090007d280075180");
objectIdentities[1] = new ObjectIdentity<ObjectId>(objId, repName);

Qualification qualification
        = new Qualification("dm_document where r_object_id = '090007d280075180'");
objectIdentities[2] = new ObjectIdentity<Qualification>(qualification, repName);

ObjectPath objPath = new ObjectPath("/testCabinet/testFolder/testDoc");
objectIdentities[3] = new ObjectIdentity<ObjectPath>(objPath, repName);

for (ObjectIdentity identity : objectIdentities)
{
    System.out.println(identity.getValueAsString());
}
```

Example 7-6. C#: ObjectIdentity subtypes

```
String repName = "MyRepositoryName";
ObjectIdentity[] objectIdentities = new ObjectIdentity[4];

// repository only is required to represent an object that has not been created
objectIdentities[0] = new ObjectIdentity(repName);

// show each form of unique identifier
ObjectId objId = new ObjectId("090007d280075180");
objectIdentities[1] = new ObjectIdentity(objId, repName);
Qualification qualification
        = new Qualification("dm_document where r_object_id = '090007d280075180'");
objectIdentities[2] = new ObjectIdentity(qualification, repName);

ObjectPath objPath = new ObjectPath("/testCabinet/testFolder/testDoc");
objectIdentities[3] = new ObjectIdentity(objPath, repName);

foreach (ObjectIdentity identity in objectIdentities)
{
    Console.WriteLine(identity.GetValueAsString());
}
```

ObjectIdentitySet

An ObjectIdentitySet is a collection of ObjectIdentity instances, which can be passed to an Object service operation so that it can process multiple repository objects in a single service interaction. An ObjectIdentitySet is analogous to a DataPackage, but is passed to service operations such as move, copy, and delete that operate only against existing repository data, and which therefore do not require any data from the consumer about the repository objects other than their identity.

ObjectIdentitySet example

The following code sample creates and populates an ObjectIdentitySet.

Example 7-7. Java: ObjectIdentitySet

```
String repName = "MyRepositoryName";
ObjectIdentitySet objIdSet = new ObjectIdentitySet();
ObjectIdentity[] objectIdentities = new ObjectIdentity[4];

// add some ObjectIdentity instances
ObjectId objId = new ObjectId("090007d280075180");
objIdSet.addIdentity(new ObjectIdentity(objId, repName));

Qualification qualification =
    new Qualification("dm_document where object_name = 'bl_upwind.gif'");
objIdSet.addIdentity(new ObjectIdentity(qualification, repName));

ObjectPath objPath = new ObjectPath("/testCabinet/testFolder/testDoc");
objIdSet.addIdentity(new ObjectIdentity(objPath, repName));

// walk through and see what we have
Iterator iterator = objIdSet.getIdentities().iterator();
while (iterator.hasNext())
{
   System.out.println("Object Identity: " + iterator.next());
}
```

Example 7-8. C#: ObjectIdentitySet

```
String repName = "MyRepositoryName";
ObjectIdentitySet objIdSet = new ObjectIdentitySet();
ObjectIdentity[] objectIdentities = new ObjectIdentity[4];

// add some ObjectIdentity instances
ObjectId objId = new ObjectId("090007d280075180");
objIdSet.AddIdentity(new ObjectIdentity(objId, repName));

Qualification qualification
        = new Qualification("dm_document where object_name =
                                            'bl_upwind.gif'");
objIdSet.AddIdentity(new ObjectIdentity(qualification, repName));

ObjectPath objPath = new ObjectPath("/testCabinet/testFolder/testDoc");
objIdSet.AddIdentity(new ObjectIdentity(objPath, repName));

// walk through and see what we have
IEnumerator<ObjectIdentity> identityEnumerator = objIdSet.
                                    Identities.GetEnumerator();
while (identityEnumerator.MoveNext())
{
  Console.WriteLine("Object Identity: " + identityEnumerator.Current);
}
```

Property

A DataObject optionally contains a PropertySet, which is a container for a set of Property objects. Each Property in normal usage corresponds to a property (also called attribute) of a repository object represented by the DataObject. A Property object can represent a single property, or an array of

properties of the same data type. Property arrays are represented by subclasses of ArrayProperty, and correspond to repeating attributes of repository objects.

Property model

The Property class is subclassed by data type (for example StringProperty), and each subtype has a corresponding class containing an array of the same data type, extending the intermediate abstract class ArrayProperty (see Figure 4, page 88).

Figure 4. Property class hierarchy

Property subtype example

The following sample shows instantiation of the various Property subtypes.

```
Property[] properties =
{
    new StringProperty("subject", "dangers"),
    new StringProperty("title", "Dangers"),
    new NumberProperty("short", (short) 1),
    new DateProperty("my_date", new Date()),
    new BooleanProperty("a_full_text", true),
    new ObjectIdProperty("my_object_id", new ObjectId("090007d280075180")),

    new StringArrayProperty("keywords",
                            new String[]{"lions", "tigers", "bears"}),
    new NumberArrayProperty("my_number_array", (short) 1, 10, 100L, 10.10),
    new BooleanArrayProperty("my_boolean_array", true, false, true, false),
    new DateArrayProperty("my_date_array", new Date(), new Date()),
    new ObjectIdArrayProperty("my_obj_id_array",
    new ObjectId("0c0007d280000107"), new ObjectId("090007d280075180")),
};
```

Transient properties

Transient properties are custom Property objects that are not interpreted by the services as representations of persistent properties of repository objects. You can therefore use transient properties to pass your own data to a service to be used for a purpose other than setting attributes on repository objects.

Note: Currently transient properties are implemented only by the Object Service validate operation.

To indicate that a Property is transient, set the isTransient property of the Property object to true.

One intended application of transient properties implemented by the services is to provide the client the ability to uniquely identify DataObject instances passed in a validate operation, when the instances have not been assigned a unique ObjectIdentity. The validate operation returns a ValidationInfoSet property, which contains information about any DataObject instances that failed validation. If the service client has populated a transient property of each DataObject with a unique identifier, the client will be able to determine which DataObject failed validation by examining the ValidationInfoSet.

Transient property example

The following sample would catch a ValidationException and print a custom id property for each failed DataObject to the console.

Example 7-9. Java: Transient properties

```
public void showTransient(ValidationInfoSet infoSet)
{
    List<ValidationInfo> failedItems = infoSet.getValidationInfos();
    for (ValidationInfo vInfo : failedItems)
    {
        System.out.println(vInfo.getDataObject()
                           .getProperties()
                           .get("my_unique_id"));
    }
}
```

Example 7-10. C#: Transient properties

```
public void ShowTransient(ValidationInfoSet infoSet)
{
    List<ValidationInfo> failedItems = infoSet.ValidationInfos;
    foreach (ValidationInfo vInfo in failedItems)
    {
      Console.WriteLine(vInfo.DataObject.Properties.Get("my_unique_id"));
    }
}
```

Loading properties: convenience API

As a convenience the Java client library will determine at runtime the correct property subclass to instantiate based on the data type passed to the Property constructor. For example, the following code adds instances of NumberProperty, DateProperty, BooleanProperty, and ObjectIdProperty to a PropertySet.

Example 7-11. Java: Loading properties

```
PropertySet propertySet = new PropertySet();

//Create instances of NumberProperty
propertySet.set("TestShortName", (short) 10);
propertySet.set("TestIntegerName", 10);
propertySet.set("TestLongName", 10L);
propertySet.set("TestDoubleName", 10.10);

//Create instance of DateProperty
propertySet.set("TestDateName", new Date());

//Create instance of BooleanProperty
propertySet.set("TestBooleanName", false);

//Create instance of ObjectIdProperty
propertySet.set("TestObjectIdName", new ObjectId("10"));

Iterator items = propertySet.iterator();

while (items.hasNext())
{
    Property property = (Property) items.next();
    {
        System.out.println(property.getClass().getName() +
                           " = " + property.getValueAsString());
    }
}
```

Example 7-12. C#: Loading properties

```
PropertySet propertySet = new PropertySet();

//Create instances of NumberProperty
propertySet.Set("TestShortName", (short) 10);
propertySet.Set("TestIntegerName", 10);
propertySet.Set("TestLongName", 10L);
propertySet.Set("TestDoubleName", 10.10);

//Create instance of DateProperty
propertySet.Set("TestDateName", new DateTime());

//Create instance of BooleanProperty
propertySet.Set("TestBooleanName", false);

//Create instance of ObjectIdProperty
propertySet.Set("TestObjectIdName", new ObjectId("10"));

List<Property> properties = propertySet.Properties;
foreach (Property p in properties)
{
    Console.WriteLine(typeof(Property).ToString() +
                      " = " +
                      p.GetValueAsString());
}
```

The NumberProperty class stores its value as a java.lang.Number, which will be instantiated as a concrete numeric type such as Short or Long. Setting this value unambiguously, as demonstrated in the preceding sample code (for example 10L or (short)10), determines how the value will be serialized in the XML instance and received by a service. The following schema shows the numeric types that can be serialized as a NumberProperty:

```
 <xs:complexType name="NumberProperty">
```

```
    <xs:complexContent>
      <xs:extension base="xscp:Property">
        <xs:sequence>
          <xs:choice minOccurs="0">
            <xs:element name="Short" type="xs:short"/>
            <xs:element name="Integer" type="xs:int"/>
            <xs:element name="Long" type="xs:long"/>
            <xs:element name="Double" type="xs:double"/>
          </xs:choice>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
```

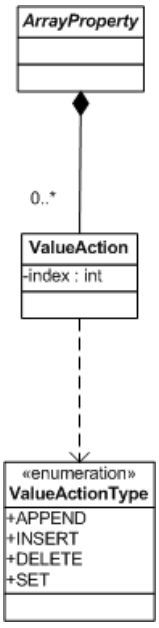
ArrayProperty

The subclasses of ArrayProperty each contain an array of Property objects of a specific subclass corresponding to a data type. For example, the NumberArrayProperty class contains an array of NumberProperty. The array corresponds to a repeating attribute (also known as repeating property) of a repository object.

ValueAction

Each ArrayProperty optionally contains an array of ValueAction objects that contain an ActionType-index pair. These pairs can be interpreted by the service as instructions for using the data stored in the ArrayProperty to modify the repeating attribute of the persistent repository object. The ValueAction array is synchronized to the ArrayProperty array, such that any position p of the ValueAction array corresponds to position p of the ArrayProperty. The index in each ActionType-index pair is zero-based and indicates a position in the repeating attribute of the persistent repository object. ValueActionType specifies how to modify the repeating attribute list using the data stored in the ArrayProperty.

Figure 5. ArrayProperty model



The following table describes how the ValueActionType values are interpreted by an update operation.

Value type	Description
APPEND	When processing ValueAction[p], the value at ArrayProperty[p] is appended to the end of repeating properties list of the persistent repository object. The index of the ValueAction item is ignored.
INSERT	When processing ValueAction[p], the value at ArrayProperty[p] is inserted into the repeating attribute list at position index. Note that all items in the list to the right of the insertion point are offset by 1, which must be accounted for in subsequent processing.
DELETE	The item at position index of the repeating attribute is deleted. When processing ValueAction[p] the value at ArrayProperty[p] must be set to a empty value (see Deleting a repeating property: use of empty value, page 92). Note that all items in the list to the right of the insertion point are offset by -1, which must be accounted for in subsequent processing.
SET	When processing ValueAction[p], the value at ArrayProperty[p] replaces the value in the repeating attribute list at position index.

Note in the preceding description of processing that the INSERT and DELETE actions will offset index positions to the right of the alteration, as the ValueAction array is processed from beginning to end. These effects must be accounted for in the coding of the ValueAction object, such as by ensuring that the repeating properties list is processed from right to left.

Deleting a repeating property: use of empty value

When using a ValueAction to delete a repeating attribute value, the value stored at position ArrayProperty[p], corresponding to ValueAction[p] is not relevant to the operation. However, the two

arrays must still line up. In this case, you should store an empty (dummy) value in ArrayProperty[p] (such as the empty string ""), rather than null.

PropertySet

A PropertySet is a container for named Property objects, which typically (but do not necessarily) correspond to persistent repository object properties.

You can restrict the size of a PropertySet returned by a service using the filtering mechanism of the PropertyProfile class (see PropertyProfile, page 93).

PropertySet example

Example 7-13. Java: PropertySet

```
Property[] properties =
{
    new StringProperty("subject", "dangers"),
    new StringProperty("title", "Dangers"),
    new StringArrayProperty("keywords",
                            new String[]{"lions", "tigers", "bears"}),
};
PropertySet propertySet = new PropertySet();
for (Property property : properties)
{
    propertySet.set(property);
}
```

Example 7-14. C#: PropertySet

```
Property[] properties =
{
    new StringProperty("subject", "dangers"),
    new StringProperty("title", "Dangers"),
    new StringArrayProperty("keywords",
                            new String[]{"lions", "tigers", "bears"}),
};
PropertySet propertySet = new PropertySet();
foreach (Property property in properties)
{
    propertySet.Set(property);
}
```

PropertyProfile

A PropertyProfile defines property filters that limit the properties returned with an object by a service. This allows you to optimize the service by returning only those properties that your service consumer requires. PropertyProfile, like other profiles, is generally set in the OperationOptions passed to a service operation (or it can be set in the service context).

You specify how PropertyProfile filters returned properties by setting its PropertyFilterMode. The following table describes the PropertyProfile filter settings:

PropertyFilterMode	Description
NONE	No properties are returned in the PropertySet. Other settings are ignored.
SPECIFIED_BY_INCLUDE	No properties are returned unless specified in the includeProperties list.
SPECIFIED_BY_EXCLUDE	All properties are returned unless specified in the excludeProperties list.
ALL_NON_SYSTEM	Returns all properties except system properties.
ALL	All properties are returned.

If the PropertyFilterMode is SPECIFIED_BY_INCLUDE, you can use processIncludedUnknown property of the PropertyFilter to control whether to process any property in the includedProperties list that is not a property of the repository type. If processIncludedUnknown is false, DFS ignore any such property specified in the includeProperties list. The default value of processIncludedUnknown is false.

Avoid unintended updates to system properties

Updates to system properties during an update or checkin can produce unexpected results and should be avoided unless you explicitly intend to change a system property. The update and checkin operations (and other operations as well) will attempt to update any properties that are populated in a DataObject provided by the operation. These properties can only be modified by a superuser, so the attempt will generally result in a permissions error. If the user making the update is a superuser, unintended changes to system properties may cause side effects.

When you initially populate the properties of the DataObject (for example, using the result of an Object service get or create operation), avoid setting the PropertyFilterMode to ALL, if you plan to pass the result into a checkin or update operation. Instead, you can set the property filter to ALL_NON_SYSTEM. (The default is operation-specific, but this is generally the default setting for Object service get and similar operations.)

If you do need to modify a system property, you should strip other system properties from the DataObject prior to the update.

PropertyProfile example

The following samples add a PropertyProfile to the operationOptions argument to be passed to an operation. The PropertyProfile will instruct the service to include only specified properties in the PropertySet of each returned DataObject.

Example 7-15. Java: PropertyProfile

```
PropertyProfile propertyProfile = new PropertyProfile();
propertyProfile.setFilterMode(PropertyFilterMode.SPECIFIED_BY_INCLUDE);
```

```
ArrayList<String> includeProperties = new ArrayList<String>();
includeProperties.add("title");
includeProperties.add("object_name");
includeProperties.add("r_object_type");
propertyProfile.setIncludeProperties(includeProperties);
OperationOptions operationOptions = new OperationOptions();
operationOptions.setPropertyProfile(propertyProfile);
```

Example 7-16. C#: PropertyProfile

```
PropertyProfile propertyProfile = new PropertyProfile();
propertyProfile.FilterMode = PropertyFilterMode.SPECIFIED_BY_INCLUDE;
List<string> includeProperties = new List<string>();
includeProperties.Add("title");
includeProperties.Add("object_name");
includeProperties.Add("r_object_type");
propertyProfile.IncludeProperties = includeProperties;
OperationOptions operationOptions = new OperationOptions();
operationOptions.PropertyProfile = propertyProfile;
```

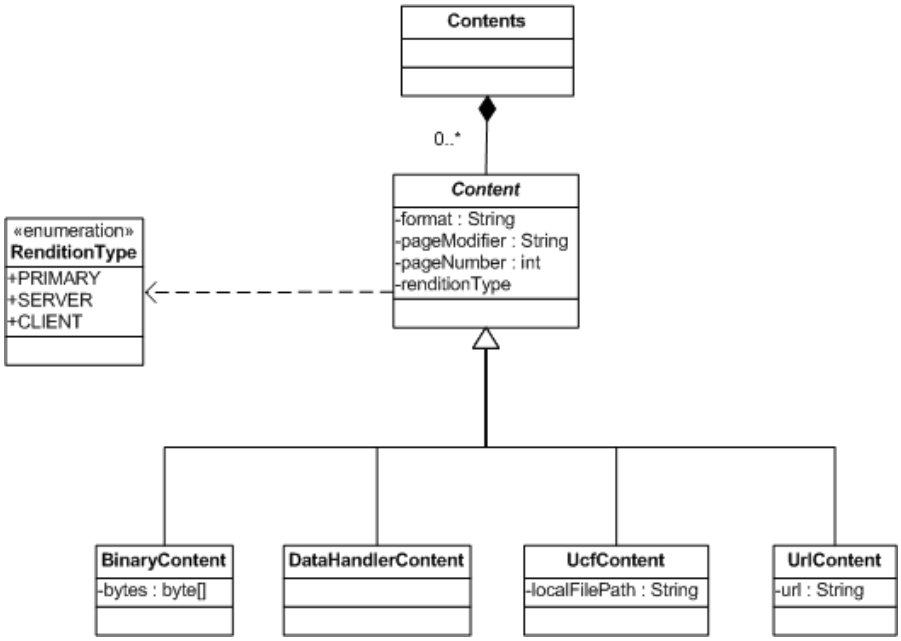
Content model and profiles

Content in a DataObject is represented by an instance of a subtype of the Content class. A DataObject contains a list of zero or more Content instances. The following sections describe the Content model and two profiles used to control content transfer: ContentProfile and ContentTransferProfile.

Content model

The DFS content model provides a content type corresponding to each support method of content transfer. The following diagram shows the model as defined in the DFS WSDLs.

Figure 6. DFS content model



A DataObject contains a Contents collection, in which each Content instance can represent the DataObject's primary content, a rendition created by a user (RenditionType.CLIENT), or a rendition created by Content Server (RenditionType.SERVER). A repository object can have only one primary content object and zero or more renditions.

The BinaryContent type includes a Base64–encoded byte array and is typically used with the Base64 content transfer mode:

```
<xs:complexType name="BinaryContent">
    <xs:complexContent>
      <xs:extension base="tns:Content">
        <xs:sequence>
          <xs:element name="Value" type="xs:base64Binary"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
```

In the DFS content model, MTOM data is typically represented by the DataHandlerContent type, which is defined as follows:

```
<xs:complexType name="DataHandlerContent">
  <xs:complexContent>
    <xs:extension base="tns:Content">
      <xs:sequence>
        <xs:element name="Value"
                    ns1:expectedContentTypes="*/*"
                    type="xs:base64Binary"
                    xmlns:ns1="http://www.w3.org/2005/05/xmlmime"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

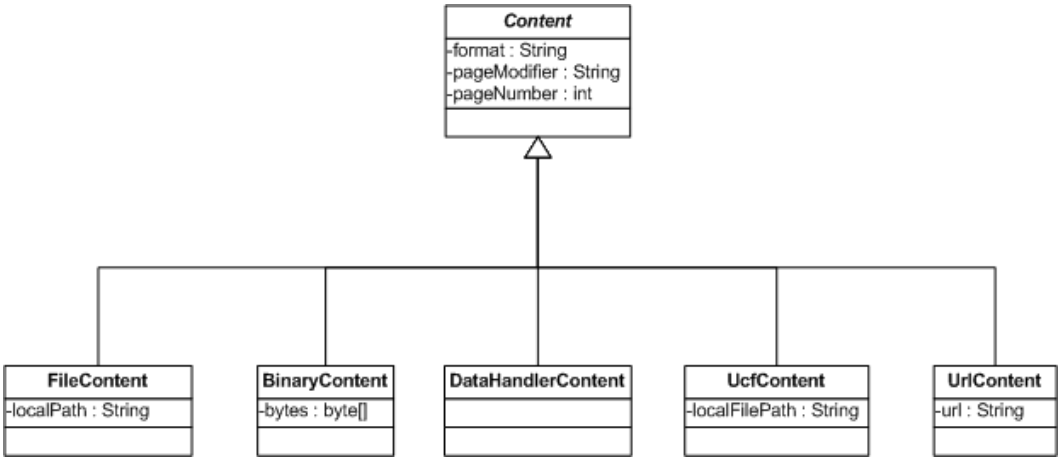
The UrlContent type includes a string representing the location of a content resource. This URL is used to download content from a repository through an Acclerated Content Services (ACS) server or

associated BOCS cache. The ACS URL is set to expire after a period that is configurable on the ACS server (the default setting is 6 hours), so they are not suitable for long-term storage and reuse.

```
<xs:complexType name="UrlContent">
  <xs:complexContent>
    <xs:extension base="tns:Content">
      <xs:sequence/>
      <xs:attribute name="url" type="xs:string" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

The DFS client productivity layer includes an additional class, FileContent, which is used as a convenience class for managing content files. FileContent is also the primary type returned to the productivity layer by services invoked in local mode.

Figure 7. Productivity Layer Content classes



ContentProfile

The ContentProfile class enables a client to set filters that control the content returned by a service. This has important ramifications for service performance, because it permits fine control over expensive content transfer operations.

ContentProfile includes three types of filters: FormatFilter, PageFilter, and PageModifierFilter. For each of these filters there is a corresponding variable that is used or ignored depending on the filter settings. For example, if the FormatFilter value is FormatFilter.SPECIFIED, the service will return content that has a format specified by the ContentProfile.format property. Each property corresponds to a setting in the dmr_content object that represents the content in the repository.

The following table describes the ContentProfile filter settings:

Value type	Value	Description
FormatFilter	NONE	No content is included. All other filters are ignored.

Value type	Value	Description
	SPECIFIED	Return only content specified by the format setting. The format property corresponds to the name or to the mime_type of a dm_format object installed in the repository.
	ANY	Return content in any format, ignoring format setting.
PageFilter	SPECIFIED	Return only page number specified by pageNumber setting. The pageNumber property corresponds to the dmr_content.page property in the repository for content objects that have multiple pages.
	ANY	Ignore pageNumber setting.
PageModifierFilter	SPECIFIED	Return only page number with specified pageModifier. The pageModifier property corresponds to the dmr_content.page_modifier attribute in the repository. This setting is used to distinguish different renditions of an object that have the same format (for example, different resolution settings for images or sound recordings).
	ANY	Ignore pageModifier setting.

Note that you can use the following DQL to get a list of all format names stored in a repository:

```
SELECT "name", "mime_type", "description" FROM "dm_format"
```

postTransferAction

You can set the postTransferAction property of a ContentProfile instance to open a document downloaded by UCF for viewing or editing.

- To open the document for edit, ensure the document is checked out before the UCF content transfer.
- If the document has not been checked out from the repository, you can open the document for viewing it (as read-only).

contentReturnType

The contentReturnType property of a ContentProfile is a client-side convenience setting used in the productivity layer. It sets the type of Content returned in the DataObject instances returned to the productivity layer by converting the type returned from a remote service (or returned locally if you are using the productivity layer in local mode). It does not influence the type returned in the SOAP envelope by a remote service.

Note: Setting the contentReturnType can result in a ContentTransformationException if you set contentReturnType to UrlContent, because it is not possible to transform a content stream to a UrlContent instance on the client.

ContentProfile example

The following sample sets a ContentProfile in operationOptions. The ContentProfile will instruct the service to exclude all content from each returned DataObject.

```
ContentProfile contentProfile = new ContentProfile();
contentProfile.setFormatFilter(FormatFilter.ANY);
OperationOptions operationOptions = new OperationOptions();
operationOptions.setContentProfile(contentProfile);
```

ContentTransferProfile

Settings in the ContentTransferProfile class determine the mode of content transfer, and also specify behaviors related to content transfer in a distributed environment. Distributed content transfer can take place when DFS delegates the content transfer to UCF, or when content is downloaded from an ACS server or BOCS cache using a UrlContent object.

Field	Data type	Description
transferMode	ContentTransferMode	The transfer mode. Possible values are MTOM, BASE64, and UCF.
geolocation	String	Geolocation represents an area on the network's topography (also referred to as *network location*). The Geolocation is used to determine the closest location of the content storage on a repository or in a BOCS cache.
isCachedContentTransferAllowed	boolean	If true, content can be read from or written to a BOCS cache.
isAsyncContentTransferAllowed	boolean	If true, content can written to a BOCS cache using asynchronous write.
activityInfo	ActivityInfo	ActivityInfo stores information provided by the AgentService and the UCF client that enables the DFS runtime to orchestrate a UCF transfer. It also controls whether a UCF client session is closed automatically after it is used by a service operation.
defaultTransferMode	ContentTransferMode	Transfer mode to use if none explicitly specified.
xmlApplicationName	String	The name of the XML application to use to process XML content. If this property is set to "ignore" the content will not be processed by an XML application in subsequent operations.

Permissions

A DataObject contains a list of Permission objects, which together represent the permissions of the user who has logged into the repository on the repository object represented by the DataObject. The intent of the Permission list is to provide the client with read access to the current user's permissions on a repository object. The client cannot set or update permissions on a repository object by modifying the Permission list and updating the DataObject. To actually change the permissions, the client would need to modify or replace the repository object's permission set (also called an Access Control List, or ACL).

Each Permission has a permissionType property can be set to BASIC, EXTENDED, or CUSTOM. BASIC permissions are *compound* (sometimes called hierarchical), meaning that there are levels of permission, with each level including all lower-level permissions. For example, if a user has RELATE permissions on an object, the user is also granted READ and BROWSE permissions. This principle does not apply to extended permissions, which have to be granted individually.

The following table shows the PermissionType enum constants and Permission constants:

Permission type	Permission	Description
BASIC	NONE	No access is permitted.
	BROWSE	The user can view attribute values of content.
	READ	The user can read content but not update.
	RELATE	The user can attach an annotation to object.
	VERSION	The user can version the object.
	WRITE	The user can write and update the object.
	DELETE	The user can delete the object.
EXTENDED	X_CHANGE_LOCATION	The user can change move an object from one folder to another. All users having at least Browse permission on an object are granted Change Location permission by default for that object.
	X_CHANGE_OWNER	The user can change the owner of the object.
	X_CHANGE_PERMIT	The user can change the basic permissions on the object.
	X_CHANGE_STATE	The user can change the document lifecycle state of the object.
	X_DELETE_OBJECT	The user can delete the object. The delete object extended permission is not equivalent to the base Delete permission. Delete Object extended permission does not grant Browse, Read, Relate, Version, or Write permission.
	X_EXECUTE_PROC	The user can run the external procedure associated with the object. All users having at least Browse permission on an object are granted Execute Procedure permission by default for that object.

Note: The granted property of a Permission is reserved for future use to designate whether a Permission is explicitly not granted, that is to say, whether it is explicitly denied. In EMC Documentum 6, only granted permissions are returned by services.

PermissionProfile

The PermissionProfile class enables the client to set filters that control the contents of the Permission lists in DataObject instances returned by services. By default, services return an empty Permission list: the client must explicitly request in a PermissionProfile that permissions be returned.

The ContentProfile includes a single filter, PermissionTypeFilter, with a corresponding permissionType setting that is used or ignored depending on the PermissionTypeFilter value. The permissionType is specified with a Permission.PermissionType enum constant.

The following table describes the permission profile filter settings:

Value type	Value	Description
PermissionType-Filter	NONE	No permissions are included
	SPECIFIED	Include only permissions of the type specified by the PermissionType attribute
	ANY	Include permissions of all types

Compound (hierarchical) permissions

Content Server BASIC permissions are *compound* (sometimes called hierarchical), meaning that there are conceptual levels of permission, with each level including all lower-level permissions. For example, if a user has RELATE permissions on an object, the user is also implicitly granted READ and BROWSE permissions on the object. This is a convenience for permission management, but it complicates the job of a service consumer that needs to determine what permissions a user has on an object.

The PermissionProfile class includes a useCompoundPermissions setting with a default value of false. This causes any permissions list returned by a service to include all BASIC permissions on an object. For example, if a user has RELATE permissions on the object, a Permissions list would be returned containing three BASIC permissions: RELATE, READ, and BROWSE. You can set useCompoundPermissions to true if you only need the highest-level BASIC permission.

PermissionProfile example

The following example sets a PermissionProfile in operationOptions, specifying that all permissions are to be returned by the service.

```
PermissionProfile permissionProfile = new PermissionProfile();
permissionProfile.setPermissionTypeFilter(PermissionTypeFilter.ANY);
OperationOptions operationOptions = new OperationOptions();
```

```
operationOptions.setPermissionProfile(permissionProfile);
```

Relationship

Relationships allow the client to construct a single DataObject that specifies all of its relationships to other objects, existing and new, and to get, update, or create the entire set of objects and their relationships in a single service interaction.

The Relationship class and its subclasses, ObjectRelationship and ReferenceRelationship, define the relationship that a repository object (represented by a DataObject instance) has, or is intended to have, to another object in the repository (represented within the Relationship instance). The repository defines object relationships using different constructs, including generic relationship types represented by hardcoded strings (folder and virtual_document) and dm_relation objects, which contain references to dm_relation_type objects. The DFS Relationship object provides an abstraction for dealing with various metadata representations in a uniform manner.

This document will use the term *container DataObject* when speaking of the DataObject that contains a Relationship. It will use the term *target object* to refer to the object specified within the Relationship. Each Relationship instance defines a relationship between a container DataObject and a target object. In the case of the ReferenceRelationship subclass, the target object is represented by an ObjectIdentity; in the case of an ObjectRelationship subclass, the target object is represented by a DataObject. Relationship instances can therefore be nested, allowing the construction of complex DataObject graphs.

There are important restrictions to be aware of when retrieving a data graph—see Restrictions when retrieving deep relationships, page 111.

ReferenceRelationship and ObjectRelationship

The create and update Object service operations can be used to modify instances of ReferenceRelationship and ObjectRelationship. These service operations use distinct rules when processing instances of ReferenceRelationship and ObjectRelationship.

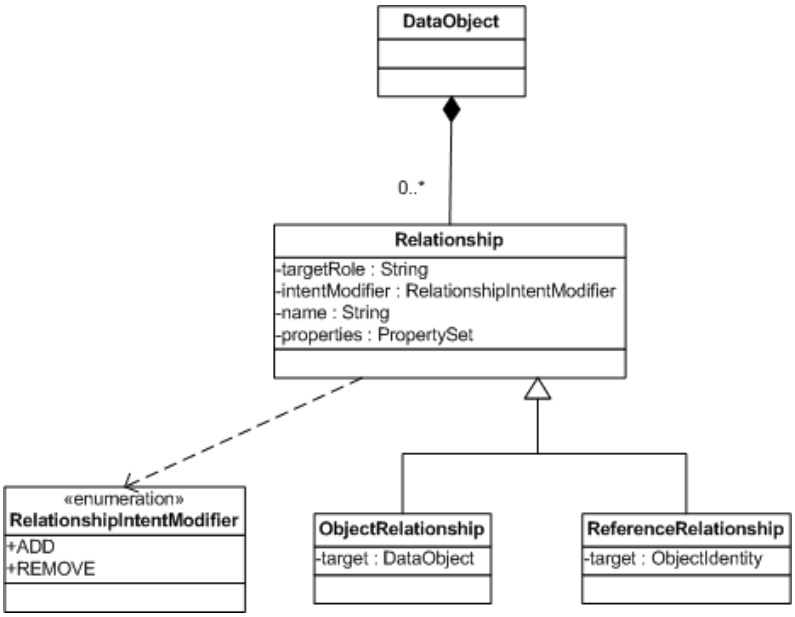
A ReferenceRelationship represents a relationship to an existing repository object, and is specified using an ObjectIdentity. A ReferenceRelationship can be used to create a relationship between two objects, but it cannot be used to update or create target objects. A common use case would be linking a repository object (as it is created or updated) into an existing folder.

An ObjectRelationship represents a relationship to a new or existing repository object. An ObjectRelationship is used by the update operation to either update or create target objects. If an ObjectRelationship received by an update operation represents a new repository object, the object is created. If the ObjectRelationship represents an existing repository object, the object is updated. A possible use case would be the creation of a new folder and a set of new documents linked to the folder.

Relationship model

Figure 8, page 103 shows the model of Relationship and related classes.

Figure 8. Relationship model



Relationship properties

The following table describes the properties of the Relationship class:

Property	Type	Description
targetRole	String	Specifies the role of the target object in the Relationship. For example, in relationships between a folder and an object linked into the folder the roles are parent and child.
intentModifer	RelationshipIn-tentModifier	Specifies how the client intends for the Relationship object to be handled by an update operation.
name	String	The name of the relationship. The Relationship class defines the following constants for the names of common relationship types: RELATIONSHIP_FOLDER, VIRTUAL_DOCUMENT_RELATIONSHIP, LIGHT_OBJECT_RELATIONSHIP, and DEFAULT_RELATIONSHIP. DEFAULT_RELATIONSHIP is set to RELATIONSHIP_FOLDER. In relationships based on dm_relation objects, the dm_relation_type name is the relationship name.
properties	PropertySet	If the relationship supports custom properties, these properties can be provided in the PropertySet. The relationship implementation should support a separate

Property	Type	Description
		persistent object in this case. For example: a subtype of dm_relation with custom attributes.

RelationshipIntentModifier

The following table describes the possible values for the RelationshipIntentModifier.

IntentModifier value	Description
ADD	Specifies that the relation should be added by an update operation if it does not exist, or updated if it does exist. This is the default value: the intentModifier of any Relationship is implicitly ADD if it is not explicitly set to REMOVE.
REMOVE	This setting specifies that a relationship should be removed by an update operation.

Relationship targetRole

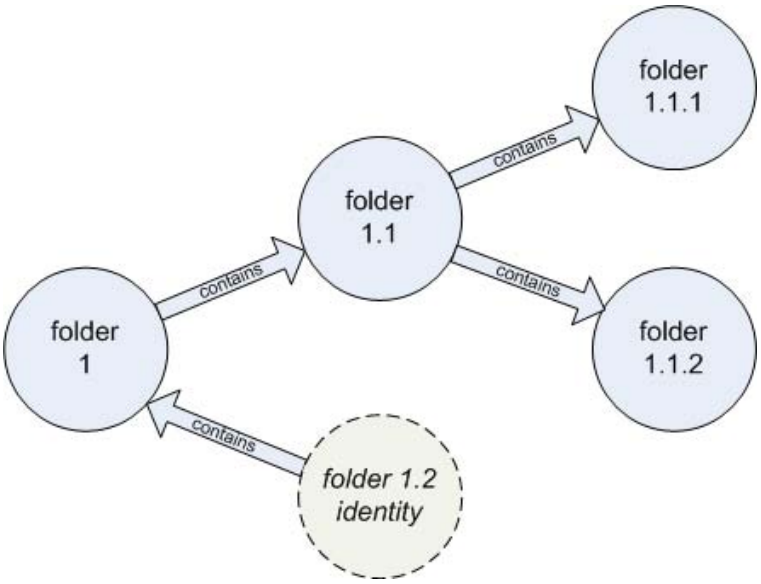
Relationships are directional, having a notion of source and target. The targetRole of a Relationship is a string representing the role of the target in a relationship. In the case of folders and VDMs, the role of a participant in the relationship can be parent or child. The following table describes the possible values for the Relationship targetRole.

TargetRole value	Description
Relationship.ROLE_PARENT	Specifies that the target object has a parent relationship to the container DataObject. For example, if a DataObject represents a dm_document, and the target object represents a dm_folder, the targetRole of the Relationship should be "parent". This value is valid for folder and virtual document relationships, as well as relationships based on a dm_relation object.
Relationship.ROLE_CHILD	Specifies that the target object has a child relationship to the container DataObject. For example, if a DataObject represents a dm_folder, and the target object represents a dm_document, the targetRole of the Relationship would be child. This value is valid for folder and virtual document relationships, as well as relationships based on a dm_relation object.

DataObject as data graph

A DataObject, through the mechanism of Relationship, comprises a data graph or tree of arbitrary depth and complexity. When the DataObject is parsed by a service, each DataObject directly contained in the DataPackage is interpreted as the root of the tree. A ReferenceRelationship, because it does not nest a DataObject, is always necessarily a leaf of the tree. An ObjectRelationship can be branch or leaf. Figure 9, page 105 shows a complex DataObject consisting of a set of related folders.

Figure 9. Relationship tree



The order of branching is determined not by hierarchy of parent-child relationships, but by the nesting of Relationship instances within DataObject instances. In some service processing it may be useful to reorder the graph into a tree based on parent-child hierarchy. Some services do this reordering and parse the tree from the root of the transformed structure.

DataObject graph structural types

A DataObject can have any of the following structures:

- A simple standalone DataObject, which contains no Relationship instances.
- A DataObject with references, containing only instances of ReferenceRelationship.
- A compound DataObject, containing only instances of ObjectRelationship.
- A compound DataObject with references, containing both ReferenceRelationship and ObjectRelationship instances.

Standalone DataObject

A standalone DataObject has no specified relationships to existing repository objects or to other DataObject instances. Standalone DataObject instances would typically be the result of a get operation or used to update an existing repository object. They could also be created in the repository independently of other objects, but normally a new object would have at least one ReferenceRelationship to specify a folder location. Figure 10, page 106 represents an object of this type.

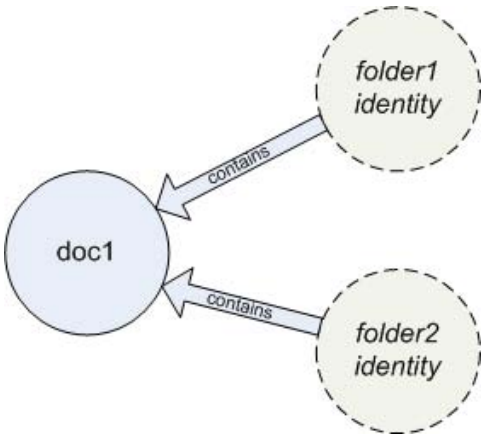
Figure 10. Standalone DataObject

DataObject with references

A DataObject with references models a repository object (new or existing) with relationships to existing repository objects. References to the existing objects are specified using objects of class ObjectIdentity.

As an example, consider the case of a document linked into two folders. The DataObject representing the document would need two ReferenceRelationship instances representing dm_folder objects in the repository. The relationships to the references are directional: from parent to child. The folders must exist in the repository for the references to be valid. Figure 11, page 106 represents an object of this type.

Figure 11. DataObject with references



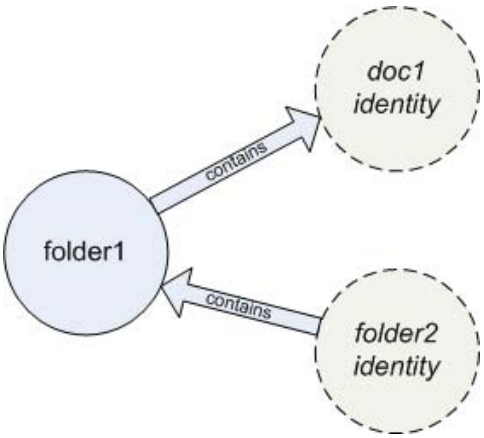
To create this object with references you could write code that does the following:

1. Create a new DataObject: doc1.
2. Add to doc1 a ReferenceRelationship to folder1 with a targetRole of "parent".
3. Add to doc1 a ReferenceRelationship to folder2 with a targetRole of "parent".

In most cases the client would know the ObjectId of each folder, but in some cases the ObjectIdentity can be provided using a Qualification, which would eliminate a remote query to look up the folder ID.

Let's look at a slightly different example of an object with references (Figure 12, page 107). In this case we want to model a new folder within an existing folder and link an existing document into the new folder.

Figure 12. DataObject with parent and child references



To create this DataObject with references you could write code that does the following:

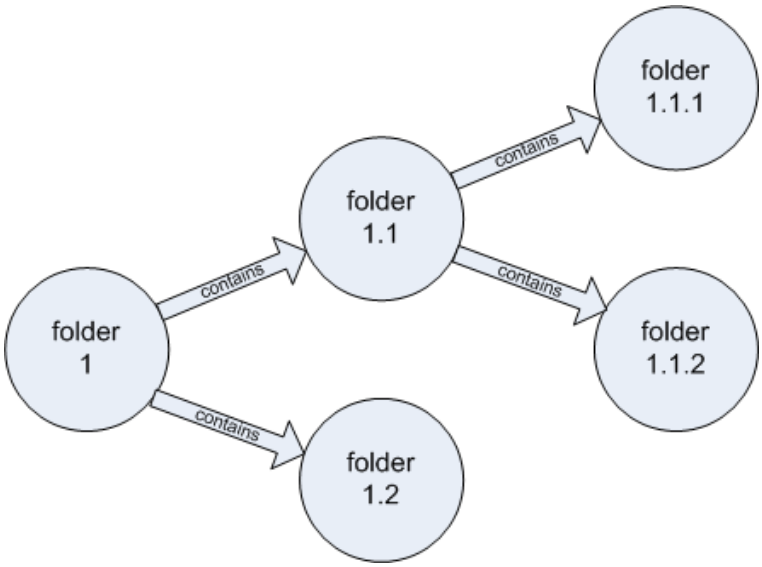
1. Create a new DataObject: folder1.
2. Add to folder1 a ReferenceRelationship to folder2 with a targetRole of "parent".
3. Add to folder1 a ReferenceRelationship to doc1 with a targetRole of "child".

Compound DataObject instances

In many cases it is relatively efficient to create a complete hierarchy of objects and then create or update it in the repository in a single service interaction. This can be accomplished using a compound DataObject, which is a DataObject containing ObjectRelationship instances.

A typical case for using a compound DataObject would be to replicate a file system's folder hierarchy in the repository. Figure 13, page 108 represents an object of this type.

Figure 13. Compound DataObject



To create this compound DataObject you could write code that does the following:

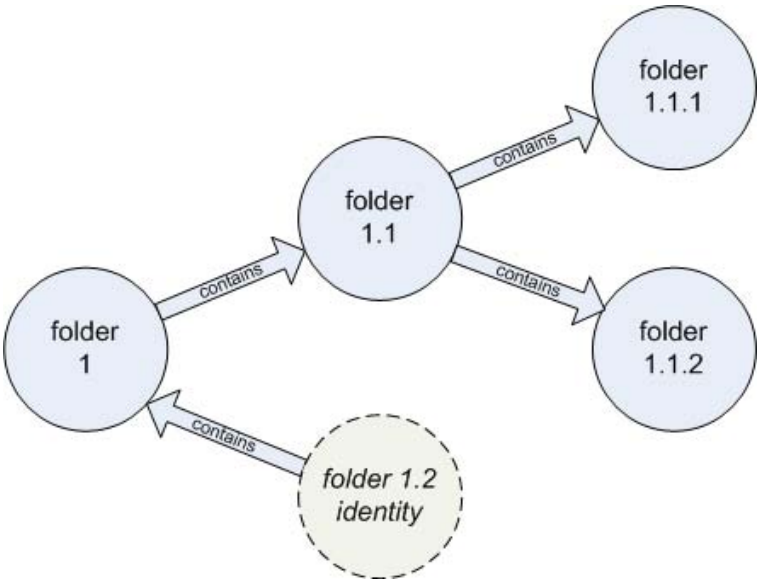
1. Create a new DataObject, folder 1.
2. Add to folder 1 an ObjectRelationship to a new DataObject, folder 1.1, with a targetRole of "child".
3. Add to folder 1.1 an ObjectRelationship to a new DataObject, folder 1.1.1, with a targetRole of "child".
4. Add to folder 1.1 an ObjectRelationship to a new DataObject, folder 1.1.2, with a targetRole of "child".
5. Add to folder 1 an ObjectRelationship to a new DataObject, folder 1.2, with a targetRole of "child".

In this logic there is a new DataObject created for every node and attached to a containing DataObject using a child ObjectRelationship.

Compound DataObject with references

In a normal case of object creation, the new object will be linked into one or more folders. This means that a compound object will also normally include at least one ReferenceRelationship. Figure 14, page 109 shows a compound data object representing a folder structure with a reference to an existing folder into which to link the new structure.

Figure 14. Compound object with references



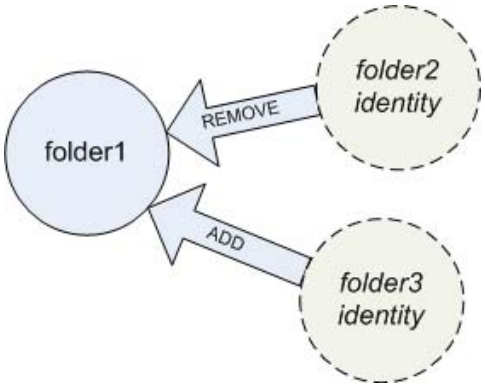
To create this compound DataObject you could write code that does the following:

1. Create a new DataObject, folder 1.
2. Add to folder 1 an ObjectRelationship to a new DataObject, folder 1.1, with a targetRole of "child".
3. Add to folder 1.1 an ObjectRelationship to a new DataObject, folder 1.1.1, with a targetRole of "child".
4. Add to folder 1.1 an ObjectRelationship to a new DataObject, folder 1.1.2, with a targetRole of "child".
5. Add to folder 1 a ReferenceRelationship to an existing folder 1.2, with a targetRole of "parent".

Removing object relationships

The Relationship intentModifier setting allows you to explicitly specify how an update operation processes ReferenceRelationship objects. The default setting of intentModifier for all Relationship instances is ADD, which means that the update operation will handle the ReferenceRelation using default processing. Setting intentModifier to REMOVE requests that the update service remove an existing relation. Figure 15, page 110 illustrates it.

Figure 15. Removing a relationship



The preceding diagram shows that a new PARENT relation to folder 3 is added to folder 1, and an existing relation with folder 2 is removed. This has the effect of linking folder1 into folder3 and removing it from folder2. The folder2 object is not deleted.

To configure the data object you would:

1. Create a new DataObject, folder1.
2. Add to folder1 a ReferenceRelationship to folder2, with an intentModifier set to REMOVE.
3. Add to folder1 a ReferenceRelationship to folder3, with a targetRole of "parent".

RelationshipProfile

A RelationshipProfile is a client optimization mechanism that provides fine control over the size and complexity of DataObject instances returned by services. By default, the Object service get operation returns DataObject containing no Relationship instances. To alter this behavior, you must provide a RelationshipProfile that explicit sets the types of Relationship instances to return.

ResultDataMode

The RelationshipProfile.resultDataMode setting determine whether the Relationship instances contained in a DataObject returned by an Object service get operation are of type ObjectRelationship or ReferenceRelationship. If they are of type ObjectRelationship they will contain actual DataObject instances; if they are of type ReferenceRelationship, they will contain only an ObjectIdentity. The following table describe the possible values of resultDataMode:

resultDataMode value	Description
REFERENCE	Return all Relationship instances as ReferenceRelationship, which contain only the ObjectIdentity of the related object.
OBJECT	Return all relations as ObjectRelationship objects, which contain actual DataObject instances.

Note that if resultDataMode is set to REFERENCE, the depth of relationships retrieved can be no greater than 1. This is because the related objects retrieved will be in the form of an ObjectIdentity, and so cannot nest any Relationship instances.

Relationship filters

RelationshipProfile includes a number of filters that can be used to specify which categories of Relationship instances are returned as part of a DataObject. For some of the filters you will need to specify the setting in a separate property and set the filter to SPECIFIED. For example, to filter by relationName, set nameFilter to SPECIFIED, and use the relationName property to specify the relationship name string.

The filters are ANDed together to specify the conditions for inclusion of a Relationship instance. For example, if targetRoleFilter is set to RelationshipProfile.ROLE_CHILD and depthFilter is set to 1, only proximate child relationships will be returned by the service.

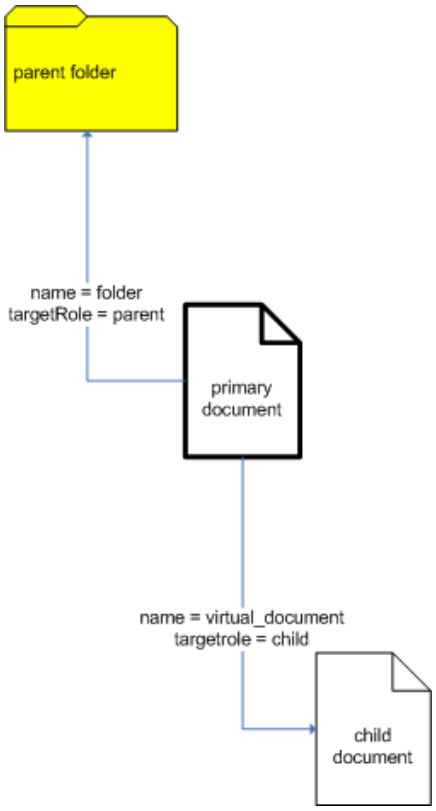
The following table describes the filters and their settings.

Value type	Value	Description
RelationshipNameFilter	SPECIFIED	Only Relationship instances with the name specified in the relationName property will be included.
	ANY	relationName property is ignored, and Relationship instances are not filtered by name.
TargetRoleFilter	SPECIFIED	Only relations with the target role specified in the targetRole attribute will be included.
	ANY	Do not filter Relationship instances by targetRole setting (that is, ignore targetRole setting).
DepthFilter	SINGLE	Return only the specified object itself, with no relationships. This is the default behavior.
	SPECIFIED	Return Relationship instances to the level specified in the depth property. A depth of 1 will return the closest level of relationship (for example a containing folder or child object).
	UNLIMITED	Return Relationship instances without regard to depth property, to indeterminate level.

Restrictions when retrieving deep relationships

When you retrieve the proximate relationships of an object (where depth = 1), there are no restrictions on the relationships returned in the graph: all relationships are returned, regardless of their name and targetRole. To take a simple example, you can retrieve the relationships of a document that has a folder relationship to a parent folder and a virtual_document relationship to a child document.

Figure 16. No restrictions on proximate relationships



However, relationships more than one step removed from the primary DataObject (where depth > 1) will be returned in a relationship graph only if they have the same relationship name and targetRole as the first relationship on the branch. Let's look at a couple of examples of how this works. In all of the examples we will assume the following settings in the RelationshipProfile:

```
resultDataMode = ResultDataMode.OBJECT
targetRoleFile = TargetRoleFilter.ANY
nameFilelter = RelationshipNameFilter.ANY
depthFilter = DepthFilter.UNLIMITED
```

Note that to retrieve any deep relationships resultDataMode must equal ResultDataMode.OBJECT. The following code retrieves a DataObject with the preceding settings:

Example 7-17. Retrieving all relationships

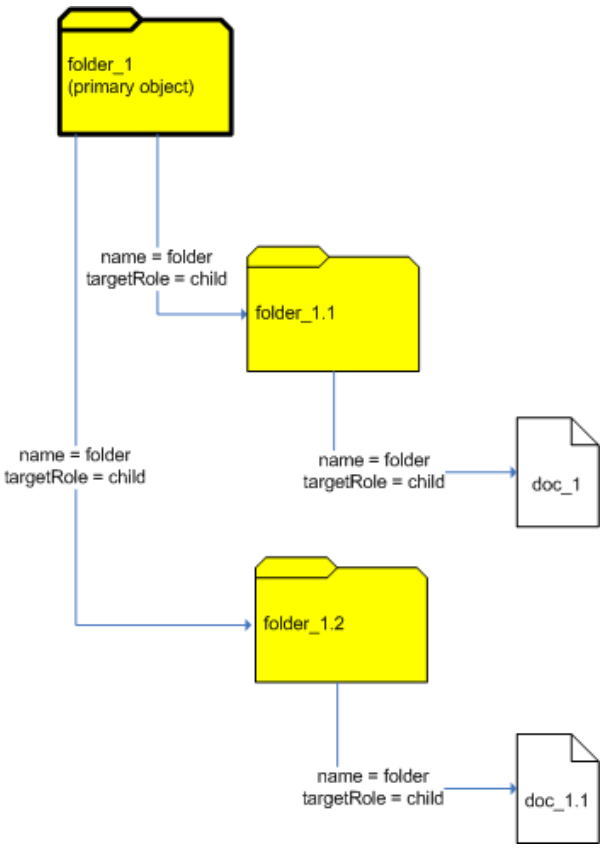
```
public DataObject getObjectWithDeepRelationships (ObjectIdentity objIdentity)
       throws ServiceException
{
    RelationshipProfile relationProfile = new RelationshipProfile();
    relationProfile.setResultDataMode(ResultDataMode.OBJECT);
    relationProfile.setTargetRoleFilter(TargetRoleFilter.ANY);
    relationProfile.setNameFilter(RelationshipNameFilter.ANY);
    relationProfile.setDepthFilter(DepthFilter.UNLIMITED);
    OperationOptions operationOptions = new OperationOptions();
    operationOptions.setRelationshipProfile(relationProfile);

    ObjectIdentitySet objectIdSet = new ObjectIdentitySet(objIdentity);
    DataPackage dataPackage = objectService.get(objectIdSet, operationOptions);

    return dataPackage.getDataObjects().get(0);
}
```

Let's start with a case where all relationships have the same relationship name (folder).

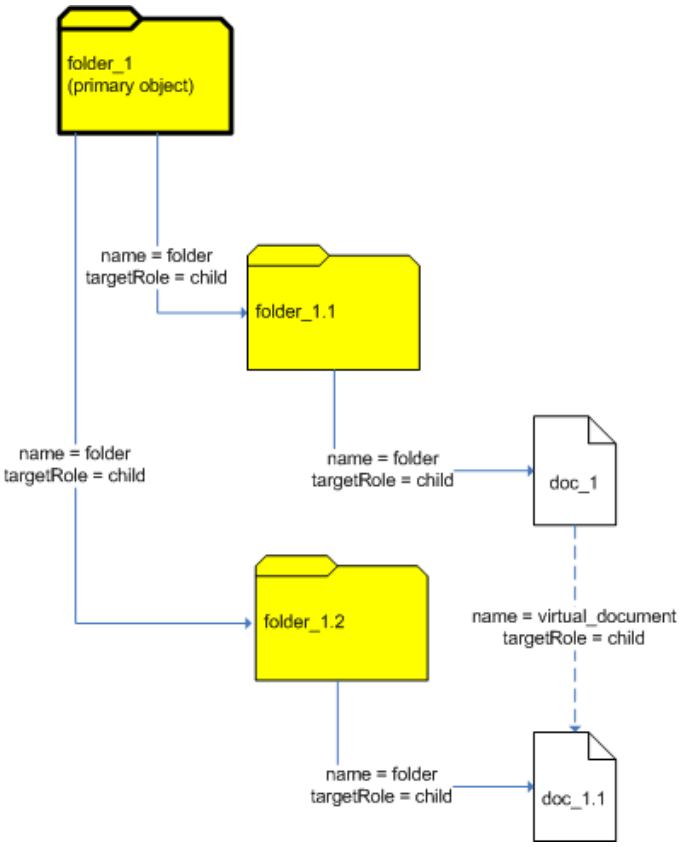
Figure 17. Restriction on deep relationships—targetRole



The primary object in this case is folder_1.2. As you can see, both of its proximate relationships are retrieved. On the child branch the deep relationship (to folder_1.2.1.1) is retrieved, because both the name and targetRole of the deep relationship is the same as the first relationship on the branch. However, on the parent branch, the relationship to folder_1.1 is not retrieved, because the targetRole of the relationship to folder_1.1 (child) is not the same as the targetRole of the first relationship on the branch (parent).

Let's look at another example where the relationship name changes, rather than the targetRole. In this example, we want to retrieve the relationships of a folder that has two child folders. Each child folder contains a document, and one of the documents is a virtual document that contains the other.

Figure 18. Restrictions on deep relationships—name



As before, both proximate relationships are retrieved. The deep folder relationships to the documents are also retrieved. But the virtual_document relationship is not retrieved, because its relationship name (virtual_document) is not the same as the name of the first relationship on the branch (folder).

Custom relationships

It is sometimes useful to create a custom relationship by extending the dm_relation object, which allows you to add custom properties to the relationship. You can extend the dm_relation object independently of defining a custom dm_relation_type object. To extend dm_relation, you could use Composer, or you could use a DQL similar to the following:

```
CREATE TYPE acme_custom_relation (str_attr string(32),
                                  bool_attr boolean,
                                  repeat_attr string(32) REPEATING)
WITH SUPERTYPE dm_relation PUBLISH
```

You can reference a custom relationship in the name property of a DFS Relationship object using the syntax:

```
<dm_relation_type name>/<dm_relation subtype name>
```

Let's look at an example of how you might use such an extended relationship. Suppose you wanted to create a custom object type called acme_geoloc to contain geographic place names and locations that can be used to display positions in maps. This geoloc object contains properties such as place name, latitude, and longitude. You want to be able to associate various documents, such as raster maps,

tour guides, and hotel brochures with an acme_geoloc object. Finally, you also want to be able to capture metadata about the relationship itself.

To enable this, you could start by making the following modifications in the repository using Composer:

- Create an acme_geoloc type (with no supertype), with properties "name", "latitude", and "longitude".
- Create an instance of dm_relation_type, which you might call acme_geoloc_relation_type. In this instance, set the parent_type property to "dm_document" and the child_type property to "acme_geoloc".
- Create a subtype of dm_relation called acme_geoloc_relation. Add a couple attributes to this type to track metadata about the relationship: rel_class (string) and is_validated (boolean).

Once these objects are created in the repository, your application can create relationships at runtime between document (dm_document) objects and acme_geoloc objects. By including the relationship in DataObject instances, your client application can choose to include geolocation information about the document for display in maps, and also examine custom metadata about the relationship itself. The following Java sample code creates an acme_geoloc object, a document, and a relationship of type acme_geoloc_relation_type between the document and the acme_geoloc.

Example 7-18. Java: Using a custom relationship

```
public DataPackage createCustomRelationshipAndLinkedDoc()
    throws ServiceException
{
    // define a geoloc object
    DataObject geoLocObject = new DataObject(new ObjectIdentity
             (defaultRepositoryName), "acme_geoloc");
    PropertySet properties = new PropertySet();
    properties.set("name", "TourEiffel");
    properties.set("latitude", "48512957N");
    properties.set("longitude", "02174016E");
    geoLocObject.setProperties(properties);

    // define a document
    DataObject docDataObj = new DataObject(new ObjectIdentity
                      (defaultRepositoryName),"dm_document");
    PropertySet docProperties = new PropertySet();
    docProperties.set("object_name", "T-Eiffel");
    docProperties.set("title", "Guide to the Eiffel Tower");
    docDataObj.setProperties(docProperties);

    // set relationship properties
    PropertySet relPropertySet = new PropertySet();
    relPropertySet.set("rel_class", "guidebook");
    relPropertySet.set("is_validated", "1");

    // add a relationship of the document to the geoloc
    ObjectRelationship objRelationship = new ObjectRelationship();
    objRelationship.setTarget(geoLocObject);
    objRelationship.setName("acme_geoloc_relation_type/
                             acme_geoloc_relation");
    objRelationship.setTargetRole(Relationship.ROLE_CHILD);
    objRelationship.setRelationshipProperties(relPropertySet);
    docDataObj.getRelationships().add(
    new ObjectRelationship(objRelationship));

    //set up property profile to include relationship properties
    PropertyProfile propertyProfile = new PropertyProfile();
```

```
    propertyProfile.setFilterMode(PropertyFilterMode.ALL_NON_SYSTEM);

    // set up the relationship filter to return the doc and folder
    RelationshipProfile relationProfile = new RelationshipProfile();
    relationProfile.setResultDataMode(ResultDataMode.OBJECT);
    relationProfile.setTargetRoleFilter(TargetRoleFilter.ANY);
    relationProfile.setNameFilter(RelationshipNameFilter.SPECIFIED);
    relationProfile.setRelationName("acme_geoloc_relation_type");
    relationProfile.setDepthFilter(DepthFilter.SPECIFIED);
    relationProfile.setDepth(1);
    relationProfile.setPropertyProfile(propertyProfile);
    OperationOptions operationOptions = new OperationOptions();
    operationOptions.setRelationshipProfile(relationProfile);

    // invoke the operation
    return objectService.create(new DataPackage(docDataObj),
                                operationOptions);
}
```

Note: It is important to note that the syntax "acme_geoloc_relation_type/acme_geoloc_relation" is used in the name property of the relationship passed to the create operation, but it is not used in the relationship name filter in the RelationshipProfile. The name filter instead uses only the name of the dm_relation_type (“acme_geoloc_relation_type”). This current limitation of DFS implies that if you have multiple dm_relation subtypes that have the same relation_name value (that is, that reference the same dm_relation_type), they cannot be disambiguated by the name filter. For example, suppose you have two dm_relation subtypes:

- acme_geoloc_relation
- acme_books_geoloc_relation

If there are objects of both of these types in the repository, and they both reference the same dm_relation_type in their relation_name property, it will not be possible to indicate in the relationship name filter which of the relationship names to filter on. To work around this limitation, use a custom dm_relation_type and make sure that only instances of your custom dm_relation subtype reference your custom dm_relation_type.

Aspect

The Aspect class models an aspect, and provides a means of attaching an aspect to a persistent object, or detaching an aspect from a persistent object during a service operation.

Aspects are a mechanism for adding behavior and/or attributes to a Documentum object *instance* without changing its type definition. They are similar to TBOs, but they are not associated with any one document type. Aspects also are late-bound rather than early-bound objects, so they can be added to an object or removed as needed.

Aspects are a BOF type (dmc_aspect_type). Like other BOF types, they have these characteristics:

- Aspects are installed into a repository.
- Aspects are downloaded on demand and cached on the local file system.
- When the code changes in the repository, aspects are automatically detected and new code is “hot deployed” to the DFC (and therefore DFS) client.

The aspect class has the following properties:

Property	Type	Description
name	String	The name of the aspect.
intentModifier	AspectIntent-Modifier	An enum value that governs how a service operation processes the DataObject containing the Aspect instance. ATTACH, the default setting, means to attach the aspect to the persistent object. DETACH means to detach the aspect.

Other classes related to DataObject

This chapter has presented the most common and complex classes related to DataObject, but is not comprehensive. Other classes related to DataObject are covered in the API documentation on the SDK, and in some cases under the service with which they are most closely associated in the *Enterprise Content Service Reference.*

Chapter 8

Custom Service Development with DFS

This chapter is intended to introduce you to writing custom services in the DFS framework and how to use the DFS SDK build tools to generate a deployable EAR file. Sample custom services are also provided to get you started on developing your own custom services with DFS. This chapter contains the following sections:

Service design considerations

The following sections discuss a few of the design considerations you may need to take into account when planning your custom service.

SBO or POJO services

DFS services can be implemented either as Business Object Framework (BOF) Service-based Business Objects (SBOs), or as Plain Old Java Objects (POJOs). The following two factors may have bearing on your decision regarding which approach to take.

- your organization's current investment in SBOs
- the degree from which your organization would benefit from the SBO deployment model

If you have existing SBOs that are used in DFC clients or projected as EMC Documentum 5.3 web services, the optimal route to DFS may be to convert the existing services into DFS services. However, bear in mind that not all SBOs are suitable for projection as web services, and those that are technically suitable may still be lacking an optimal SOA design. As an alternative strategy you could preserve current SBOs and make their functionality available as a DFS service by creating DFS services as facades to the existing SBOs.

The SBO approach may also be of value if you wish to design services that are deployed across multiple repositories and multiple DFC client applications (including WDK-based applications). An SBO implementation is stored in a single location, the global registry, from which it is dynamically downloaded to client applications. If the implementation changes, the changes can be deployed in a single location. The BOF runtime framework automatically propagates the changed implementation to all clients. (Note that the SBO *interface* must be deployed to each DFC client.)

If neither of these considerations is compelling, POJO services may be the more attractive choice, as it removes the small technical overhead and vendor-specific requirements of implementing and deploying SBOs. Note that choosing POJO services will in no way preclude the effective use of BOF objects that extend the Content Server type system (Type-based Business Objects and aspects).

DFS object model

Your custom DFS service will be more intuitive to use in combination with DFS core services if it makes appropriate use of the DFS object model (see Chapter 7, DFS Data Model) and of design principles embodied in the DFS object model.

For example, a service should always return a DFS DataPackage rather than a specialized object representing a DFC typed object. Services should always be designed so that no DFC client is required on the service consumer.

Avoid extending the DFS data model

We strongly recommend that custom data models do not extend the DFS data model classes. We recommend using aggregation rather than inheritance to leverage the existing DFS data model in custom classes. In the case of profile classes, we recommend using properties passed in OperationOptions or the ServiceContext as an alternative to creating custom profiles. This maximizes interoperability and enables use of the pre-packaged JAXB bindings.

The well-behaved service implementation

There are intrinsic differences between an efficient local interaction and an efficient remote interaction. A well-behaved service should be optimized to support an efficient remote interaction, and should exhibits the following characteristics (note that this is not an exhaustive list).

- The service should have an appropriate level of granularity. The most general rule is that the service granularity should be determined by the needs of the service consumer. However, in practice services are generally more coarse-grained than methods in tightly bound client/server applications. They should avoid "chattiness", be sensitive to round-trip overhead, and anticipate relatively low bandwidth and high latency.
- As mentioned previously, if the service is intended to be used as an extension of DFS services, it should use the DFS object model where possible, and conform to the general design features of the DFS services.
- The service should specify stateless operations that perform a single unambiguous function that the service consumer requires. The operation should stand alone and not be overly dependent on consumer calls to auxiliary services.
- The service should specify parameters and return values that are easily bound to XML, and which are faithfully transformed in interactions between the client and the service.

Not all intrinsic Java types map into identical XML intrinsic types; and not all intrinsic type arrays exhibit are transformed identically to and from XML. Service developers should therefore be aware of the tables Table 17, page 121 and Table 18, page 121 when designing service interfaces.

Table 17. Java intrinsic type to XML mappings

Java intrinsic type	Mapped XML type
boolean	boolean
byte	byte
char	**int**
double	double
float	float
int	int
long	long
short	short

Table 18. Java instrinsic type to XML mappings for arrays

Java array	XML equivalent
boolean[]	boolean
byte[]	byte[]
char[]	**List<Integer>**
double[]	**List<Double>**
float[]	**List<Float>**

Java array	XML equivalent
int[]	**List<Integer>**
long[]	**List<Long>**
short[]	**List<Short>**

DFC sessions in DFS services

In DFS sessions are handled by the service layer and are not exposed in the DFS client API. DFS services, however, do and must use managed sessions in their interactions with the DFC layer. A DFS service that uses DFC absolutely must get its instance of the DFC session manager (that is, an instance of the IDfSessionManager interface) through the DFS layer, using the getSessionManager static method of the DfcSessionManager class. This turns over much of the complexity of dealing with session managers, identities, and sessions to the DFS framework. DFS maintains a cache of session managers that are associated by a token with a service context kept in thread-local storage. DfcSessionManager.getSessionManager retrieves a session manager from the cache based on the token stored in the serviceContext, and takes care of the details of populating the session manager with identities stored in the service context. The service context itself is created based on data passed in SOAP headers from remote clients, or on data passed by a local client during service instantiation.

From the viewpoint of the custom DFS service, the essential thing is to get the session manager using DfcSessionManager.getSessionManager, then invoke the session manager to get a session on a repository. To get a session, the service needs to pass a string identifying the repository to the IDfSessionManager.getSession method, so generally a service will need to receive the repository name from the caller in one of its parameters. Once the service method has the session, it can invoke DFC methods on the session within a try clause and catch any DfException thrown by DFC. In the catch clause it should wrap the exception in a custom DFS exception (see Creating a custom exception, page 131), or in a generic ServiceException, so that the DFS framework can handle the exception appropriately and serialize it for remote consumers. The session must be released in a finally clause to prevent session leakage. This general pattern is shown in the listing below.

```
import com.emc.documentum.fs.rt.context.DfcSessionManager;
...

public void myServiceMethod(DataObject dataObject) throws ServiceException
{
    IDfSessionManager manager = null;
    IDfSession session = null;
    try
    {
        manager = DfcSessionManager.getSessionManager();
        session = manager.getSession(dataObject.getIdentity().getRepositoryName());
        // do DFC stuff with DFC session
    }
    catch (DfException e)
    {
        throw new ServiceException("E_EXCEPTION_STRING", e, dataObject.getIdentity());
    }
    finally
    {
        if (manager != null && session != null)
        {
            manager.release(session);
        }
    }
}
```

If your DFS application does not include custom services, or if your custom services do not use DFC, then you need not be too concerned about programmatic management of sessions. However, it's desirable to understand what DFS is doing with sessions because some related aspects of the runtime behavior are configurable using DFS and DFC runtime properties. As stated above, DFS maintains a cache of session managers. This cache is cleaned up at regular intervals (by default every 20 minutes), and the cached session managers expire at regular intervals (by default every 60 minutes). The two intervals can be modified in dfs-runtime.properties by changing dfs.crs.perform_cleanup_every_x_minutes and dfs.crs.cache_expiration_after_x_minutes. Once the session is obtained, it is managed by the DFC layer, so configuration settings that influence runtime behavior in regard to sessions, such as whether the sessions are pooled and how quickly their connections time out, are in dfc.properties (and named dfc.session.*). These settings are documented in the dfcfull.properties file, and DFC session management in general is discussed in the *Documentum Foundation Classes Development Guide.*

Note that for each request from a service consumer, DFS will use only one IDfSessionManager instance. All underlying DFC sessions are managed (and may be cached, depending on whether session pooling is enabled) by this instance. If there are multiple simultaneous DFS requests, there should theoretically be an equivalent number of active DFC sessions. However, the number of concurrent sessions may be limited by configuration settings in dfc.properties, or by external limits imposed by the OS or network on the number of available TCP/IP connections.

Creating a custom service with the DFS SDK build tools

DFS allows you to extend its functionality by writing custom services on top of the DFS infrastructure. Custom services can also chain in DFS as well as calls to other APIs, such as DFC. It is helpful to build and run the HelloWorldService and AcmeCustomService examples before writing your own custom services. For more information on these sample custom services, see Exploring the Hello

World service, page 134 and Exploring AcmeCustomService, page 136. To create a custom service with the DFS SDK build tools:

1. Create a service class and annotate it correctly as described in Annotating a service, page 124. The class must be annotated for the DFS SDK build tools to correctly build and package the service.
2. Determine if you want the service namespace to be automatically generated or if you want to specify the service namespace explicitly. Fore more information, see Service namespace generation, page 129.
3. Implement your service by using the principles that are described in The well-behaved service implementation, page 121. Refer to DFS exception handling, page 130 if you need guidance on creating and handling custom exceptions.
4. Define where you want the service to be addressable at, which is described in Defining the service address, page 133.
5. Build and package your service with the DFS SDK build tools as described in .Building and packaging a service into an EAR file, page 134.

Annotating a service

Class annotation

DFS specifies two Java annotations that you must annotate your service class with so the DFS SDK build tools know how to build and package your service. The annotations, @DfsBofService and @DfsPojoService, are defined in the package com.emc.documentum.fs.rt.annotations. To annotate a service class, insert the @DfsPojoService or @ DfsBofService annotation immediately above the service class declaration.

```
import com.emc.documentum.fs.rt.annotations.DfsPojoService;

@DfsPojoService()
public class AcmeCustomService implements IAcmeCustomService
{
   // service implementation
}
```

For an SBO, use the @DfsBofService annotation:

```
import com.emc.documentum.fs.rt.annotations.DfsBofService;

@DfsBofService()
public class MySBO extends DfService implements IMySBO
{
    //SBO service implementation
}
```

The annotation attributes, described in the following tables, provide overrides to default DFS SDK build tools behavior.

Table 19. DfsBofService attributes

Attribute	Description
serviceName	The name of the service. Required to be non-empty.
targetNamespace	Overrides the default Java-package-to-XML-namespace conversion algorithm. Optional.
requiresAuthentication	When set to "false", specifies that this is an open service, requiring no user authentication. Default value is "true".
useDeprecatedExcep-tionModel	Set to true if you want to maintain backwards compatibility with the DFS 6.0 SP1 or earlier exception model. Default value is “false”.

Table 20. DfsPojoService attributes

Attribute	Description
implementation	Name of implementation class. Required to be non-empty if the annotation applies to an interface declaration.
targetNamespace	Overrides the default Java-package-to-XML-namespace conversion algorithm. Optional.
targetPackage	Overrides the default Java packaging algorithm. Optional.
requiresAuthentication	When set to "false", specifies that this is an open service, requiring no user authentication. Optional; default value is "true".
useDeprecatedExcep-tionModel	Set to true if you want to maintain backwards compatibility with the DFS 6.0 SP1 or earlier exception model. Default value is “false”.

Note: Although DFS leverages JAX-WS, it does not support JSR-181 annotations. This is due to the difference in emphasis between DFS (service orientation approach) and JAX-WS (web service implementation). DFS promotes an XML-based service model and adapts JAX-WS tools (specifically wsgen and wsimport) to this service model.

Data type and field annotation

Classes that define data types that are passed to and returned by services must conform to the definition of a Javabean, and must be annotated with JAXB annotations. The following shows the AcmeServiceInfo class from the AcmeCustomService sample service. Note the specification of the namespace and its correspondence to the package name.

```
package com.acme.services.samples.common;

import javax.xml.bind.annotation.*;
import java.util.List;

@XmlType(name = "AcmeServiceInfo", namespace =
  "http://common.samples.services.acme.com/")
@XmlAccessorType(XmlAccessType.FIELD)
public class AcmeServiceInfo
{
    public boolean isSessionPoolingActive()
    {
```

```
        return isSessionPoolingActive;
    }

    public void setSessionPoolingActive(boolean sessionPoolingActive)
    {
        isSessionPoolingActive = sessionPoolingActive;
    }

    public boolean isHasActiveSessions()
    {
        return hasActiveSessions;
    }

    public void setHasActiveSessions(boolean hasActiveSessions)
    {
        this.hasActiveSessions = hasActiveSessions;
    }

    public List getRepositories()
    {
        return repositories;
    }

    public void setRepositories(List repositories)
    {
        this.repositories = repositories;
    }

    public String getDefaultSchema()
    {
        return defaultSchema;
    }

    public void setDefaultSchema(String defaultSchema)
    {
        this.defaultSchema = defaultSchema;
    }

    @XmlElement(name = "Repositories")
    private List repositories;
    @XmlAttribute
    private boolean isSessionPoolingActive;
    @XmlAttribute
    private boolean hasActiveSessions;
    @XmlAttribute
    private String defaultSchema;
}
```

Best practices for data type naming and annotation

The following recommendations support predictable and satisfactory XML generation of XML data types from Java source, which will in turn support predictable and satisfactory proxy generation from the WSDL using Visual Studio and other tools.

Data type annotation

When annotating data type classes, the following annotations are recommended:

- @XmlType:

  ```
  @XmlType(name = "AcmeServiceInfo",
           namespace = "http://common.samples.services.acme.com/")
  ```

  Note that specifying the namespace is mandatory.

- @XmlAccessorType(XmlAccessType.FIELD)
- @XmlEnum (for enumerated types)
- For complex types that have subtypes, use @XmlSeeAlso({*subtype_0*, *subtype_1*, ...*subtype_n*}). For example, the Relationship class has the following annotation:

  ```
  @XmlSeeAlso({ReferenceRelationship.class, ObjectRelationship.class})
  ```

Fields and accessors

When naming fields and accessors, the following conventions are recommended:

- With naming lists and arrays, use plurals; for example:

  ```
  String value
  List<String> values
  ```

- As a basic requirement of Javabeans and general Java convention, a field's accessors (getters and setters) should incorporate the exact field name. This leads to desired consistency between the field name, method names, and the XML element name.

  ```
  @XmlAttribute
  private String defaultSchema;

  public String getDefaultSchema()
  {
      return defaultSchema;
  }

  public void setDefaultSchema(String defaultSchema)
  {
      this.defaultSchema = defaultSchema;
  }
  ```

- Annotate primitive and simple data types (int, boolean, long, String, Date) using @XmlAttribute.
- Annotate complex data types and lists using @XmlElement, for example:

  ```
  @XmlElement(name = "Repositories")
  private List repositories;

  @XmlElement(name = "MyComplexType")
  private MyComplexType myComplexTypeInstance;
  ```

- Fields should work without initialization.
- The default of boolean members should be false.

Things to avoid

The following should be avoided when implementing classes that bind to XML types.

- Avoid exposing complex collections as an XML type, other than List<Type>. One-dimensional arrays are also safe.
- Avoid adding significant behaviors to a type, other than convenience methods such as map interfaces and constructors.
- Avoid use of the @XmlElements annotation. This annotation results in an <xsd:choice>, to which inheritance is preferred. Annotate the base class with @XmlSeeAlso instead (see Data type annotation, page 127).

The following conditions can also lead to problems either with the WSDL itself, or with .NET WSDL import utilities.

- Use of the @XmlRootElement annotation can cause namespace problems with JAXB 2.1. As a result, the .NET WSDL import utility may complain about "incompatibility of types."
- It is highly recommended that you always use the @XmlAccessorType(XmlAccessType.FIELD) to annotate data type classes. If you use the default value for @XmlAccessType (which is PROPERTY), the service generation tools will parse all methods beginning with "get" and "set", which makes it difficult to control how the text following "get" and "set" is converted to XML. If one then adds an explicit @XmlElement or @XmlAttribute on a field that already has a getter and setter, the field is likely to be include more than once in the XML schema with slightly different naming conventions.
- Exercise caution using the @XmlContent annotation. Not all types can support it. We recommend using it only for representations of long strings.

Transactions in a custom service

DFS supports transactional service operations within the context of a single DFC instance. If your custom services invokes other DFS services remotely, the remote services will execute in separate DFC instances. If your custom service is transactional, these remote operations will not be included in the transaction. However, if you invoke other DFS services locally, they will be executed within the same DFC instance as your service, so they will be included in the transaction.

For your custom service operation to execute as a transaction, the service consumer must have set the IServiceContext.USER_TRANSACTION_HINT runtime property equal to IServiceContext.TRANSACTION_REQUIRED.

To see how this works, look at the following method, and assume that it is running as a custom service operation. We set IServiceContext.USER_TRANSACTION_HINT runtime property as IServiceContext.TRANSACTION_REQUIRED before the first service call. This method invokes the create operation twice, each time creating an object. If one of the calls fail, then the transaction will be rolled back.

It is important to reiterate that testTransaction represents a method in a separate service. If the testTransaction method is called through the DFS runtime or through a SOAP call, the transactions will work as described. If testTransaction is called from within a "public static void main" method, each step will run in a separate transaction.

Example 8-1. Transactional custom service test

```
public void testTransaction(DataObject object1, DataObject object2)
             throws ServiceException
    {
      IServiceContext context = ContextFactory.getInstance().getContext();
      context.setRuntimeProperty(IServiceContext.USER_TRANSACTION_HINT,
            IServiceContext.TRANSACTION_REQUIRED);
        IObjectService service = ServiceFactory.getInstance().
           getLocalService(IObjectService.class, context);

        DataPackage dp1 = service.create(new
                          DataPackage(object1), null);
        DataPackage dp2 = service.create(new
                          DataPackage(object2), null);

        ObjectIdentity objectIdentity1 =
                dp1.getDataObjects().
                                  get(0).getIdentity();
        ObjectIdentity objectIdentity2 =
                dp2.getDataObjects().
                                 get(0).getIdentity();
        System.out.println("object created:
                " + objectIdentity1.
                                  getValue().toString());
        System.out.println("object created:
                " + objectIdentity2.
                                 getValue().toString());
}
```

Including a resources file in a service

To include a resource in a service it is necessary to get the ClassLoader for the current thread, for example:

```
Thread.currentThread().getContextClassLoader().getResource("some.properties");
```

Service namespace generation

The DFS design tools use service artifact packaging conventions to generate a consistent and logical namespace and address for the service. The ability to run a service in remote and local modes depends on the consistency between the package structure and the service namespace.

If a target namespace is not specified for a service, the default target namespace is generated by reversing the package name nodes and prepending a ws (to avoid name conflicts between original and JAX-WS generated classes). For example, if the service package name is com.acme.services.samples. the DFS SDK build tools generate the following target namespace for the service:

```
http://ws.samples.services.acme.com/
```

If the annotated service implementation class, AcmeCustomService, is in a package named com.acme.services.samples, the service artifacts (such as IAcmeCustomService) are generated in the package com.acme.services.samples.client, and the artifacts created by JAX-WS, including the generated service implementation class, are placed in the com.acme.services.samples.ws package. The target namespace of the service would be http://ws.samples.services.acme.com.

You can override this namespace generation by specifying a value for the targetNamespace attribute for the service annotation that you are using (@DfsPojoService or @DfsBofService). For more information on overriding the target namespace for a service, see Overriding default service namespace generation, page 130.

Overriding default service namespace generation

As mentioned in Service namespace generation, page 129, if the targetNamespace attribute is not specified in the service annotation, the DFS SDK build tools will generate the namespace by reversing the package name and prepending "ws".

To change this behavior, specify a value for the targetNamespace attribute of the @DfsPojoService or @DfsBofService annotation that is different from the default target namespace (this approach is used in the AcmeCustomService sample).

For example, the AcmeCustomService class is declared in the com.acme.services.samples.impl package and the targetNamespace is defined as http://samples.services.acme.com:

```
package com.acme.services.samples.impl;

import com.emc.documentum.fs.rt.annotations.DfsPojoService;

@DfsPojoService(targetNamespace = http://samples.services.
                                  acme.com)
public class AcmeCustomService
.
.
.
```

With this input, the DFS SDK build tools generate the service interface and other DFS artifacts n the com.acme.services.samples.client package. It places the service implementation and other files generated by JAX-WS in the com.acme.services.samples package. The service namespace would be "http://samples.services.acme.com" as specified in the service annotation attribute.

Note: A conflict occurs when you have two services that have the following namespaces: http://a.b.c.d and http://b.c.d/a In this case, when JAX-WS tries to generate the client proxies for these two services, they will be generated in the same package (d.c.b.a), so you will only be able to call the first service in the classpath. Avoid assigning namespaces in this way to prevent this situation.

DFS exception handling

DFS supports marshalling and unmarshalling of exceptions from the service to the consumer. DFS encapsulates the stack trace, exception message, error code, and other fields in a serializable DfsExceptionHolder object and passes it over the wire to the DFS client runtime. The DFS client runtime can then re-create the exception with the data that is contained in DfsExceptionHolder and notify the consumer of a service error. Exceptions that are part of the DFS object model are already capable of being marshalled and unmarshalled from the server to the client. You can also create your own custom exceptions when developing custom services. An example of a custom exception is included in the AcmeCustomService sample.

Creating a custom exception

In order for a custom exception to be properly initialized on the client side, all its attributes must be sent over the wire. The following requirements must be met in order for the exception handling to work properly:

- All instance variables in the exception class must be JAXB serializable; they have to be part of the java.lang package or properly JAXB annotated.
- All instance variables in the exception class must be properly set on the server side exception instance, either through explicit setters or through a constructor, so that they make it into the serialized XML.
- DFS requires the exception to have a constructor accepting the error message as a String. Optionally, this constructor can have an argument of type Throwable for chained exceptions. In other words, there must be a constructor present with the following signature: (String , Throwable) or (String).
- The exception class must have proper getter and setter methods for its instance variables (except for the error message and cause since these are set in the constructor).
- The exception class must have a field named exceptionBean of type List<DfsExceptionHolder> and accessor and mutator methods for this field. The field is used to encapsulate the exception attributes, which is subsequently sent over the wire. If this field is not present, the exception attributes will not be properly serialized and sent over the wire.
- If you do not explicitly declare your custom exception in the throws clause of a method (a RuntimeException for instance), a ServiceException is sent down the wire in its place.

When the exception is unmarshalled on the client, the DFS client runtime attempts to locate the exception class in the classpath and initialize it using a constructor with the following signature: (String , Throwable) or (String). If that attempt fails, the client runtime will throw a generic UnrecoverableException that is created with the following constructor: UnrecoverableException(String, Throwable).

To create a custom exception:

1. Create a class for your custom exception with an appropriate constructor as described in the requirements.
2. Add a field named exceptionBean of type List<DfsExceptionHolder> to the exception class. Ensure accessor and mutator methods exist for this field.
3. Define the fields that you want the exception to contain. Ensure accessor and mutator methods exist for these fields and that each field is JAXB serializable.
4. Write your service and throw your exception where needed.

The DFS runtime will receive the DfsExceptionHolder object and re-create and throw the exception on the client side.

Custom exception examples

Example 8-2. Custom exception example 1

This class extends ServiceException, so there is no need to define an exceptionBean parameter and its accessor methods, because they are already defined in ServiceException. You still need a constructor with a signature of (String, Throwable) or (String).

```
public class CustomException extends ServiceException
{
   public CustomException(String errorCode, Throwable cause,
                          Object... args)
   {
      super(errorCode, cause, args);
   }

   //constructer used for client instantiation
   public CustomException(String errorCode, Throwable cause)
   {
      super(errorCode, cause);
   }
}
```

Example 8-3. Custom exception example 2

This exception does not extend ServiceException, so the exceptionBean field is required.

```
public class CustomException2 extends Exception
{
    public CustomException2(String errorCode, Throwable cause,
                          Object... args){
        super(errorCode, cause);
        this.args = args;
    }
    public CustomException2(String errorCode, Throwable cause){
        super(errorCode, cause);
    }
  public List<DfsExceptionHolder> getExceptionBean(){
        return exceptionBean;
    }
    public void setExceptionBean(List<DfsExceptionHolder> exceptionBean){
        this.exceptionBean = exceptionBean;
    }
    public Object[] getArgs ()
    {
        return args;
    }
    public void setArgs (Object[] args)
    {
        this.args = args;
    }
    private Object[] args;
    private List<DfsExceptionHolder> exceptionBean;
}
```

Defining custom resource bundles

The process of defining a new resource bundle consists of defining a new property in dfs-runtime.properties, incrementing the index of the property.

In dfs-runtime.properties:

```
resource.bundle = dfs-messages
resource.bundle.1 = dfs-services-messages
resource.bundle.2 = dfs-bpm-services-messages
```

In local-dfs-runtime.properties.

```
resource.bundle.3 = my-custom-services-messages
```

Note: A limitation of this approach is that if a new resource bundle is required for core services in a future release, it would be defined as "resource.bundle.3" and the one defined in local-dfs-runtime.properties would override it. If you define a custom resource bundle, be aware that this could cause a future migration issue.

For more information see “DFS runtime property configuration” in the *Documentum Foundation Services Deployment Guide.*

Defining the service address

The service address generation depends on parameters set in DFS tools to designate two nodes of the package structure as (1) the context root of the service, and (2) as the service module. The following service address is generated for the AcmeCustomService sample where "services" is specified as the context root and "samples" is specified as the service module.

```
http://127.0.0.1:7001/services/samples/AcmeCustomService?wsdl
```

When instantiating a service, a Java client application can pass the module name and the fully-qualified context root to ServiceFactory.getRemoteService, as shown here:

```
 mySvc = serviceFactory.getRemoteService(IAcmeCustomService.class,
                                         context,
                                         "samples",
                                         "http://localhost:7001/services");
```

Alternatively, the client can call an overloaded method of getRemoteService that does not include the module and contextRoot parameters. In this case the client runtime obtains the module and contextRoot values from the dfs-client.xml configuration file, which specifies default service addressing values. The dfs-client.xml used by AcmeCustomService is located in resources\config. Its contents are shown here:

```
<DfsClientConfig defaultModuleName="samples" registryProviderModuleName="samples">
    <ModuleInfo name="samples"
                protocol="http"
                host="127.0.0.1"
                port="7001"
                contextRoot="services">
    </ModuleInfo>
</DfsClientConfig>
```

The order of precedence is as follows. The DFS runtime will first use parameters passed in the getRemoteService method. If these are not provided, it will use the values provided in the DfsClientConfig configuration file.

Building and packaging a service into an EAR file

Once you have implemented your service, you can input the source files into the DFS SDK build tools. The build tools will generate an EAR file that you can deploy in your application server. It is helpful to view the sample build scripts for the HelloWorldService and AcmeCustomService to get an idea of how to use the Ant tasks. To build and package a service into an EAR file:

1. Call the generateModel task, specifying as input the annotated source. For more information on calling this task, see generateModel task, page 146
2. Call the generateArtifacts task, specifying as input the annotated source and service model. For more information on calling this task, see generateArtifacts task, page 147
3. Call the buildService task to build and package JAR files for your service implementation classes. For more information on calling this task, see buildService task, page 148
4. Call the packageServiceTask to package all source artifacts into a deployable EAR or WAR file. For more information on calling this task, see packageService task, page 149

Exploring the Hello World service

The following procedures describe how to code, package, deploy, and test the Hello World service. The service does not demonstrate Documentum functionality, but serves as a starting point for understanding how to build custom DFS services with the DFS SDK tools. The service and consumer code as well as an Ant build.xml file to build the service are provided in the DFS SDK.

Pre-requisites

Ensure the correct version of the ant is set up:

- Download ant 1.7.0 from http://ant.apache.org and unzip it on your local machine.
  - Add the System variable ANT_HOME=C:\apache-ant-1.7.0
  - Add the System variable ANT_OPTS=-Xmx512m -Xms512m -XX:MaxPermSize=128m
  - Add %ANT_HOME%\bin to Path

Building the Hello World service

Building the Hello World service involves running the Ant tasks in the DFS SDK tools on the service code. To build the Hello World service:

1. Extract the emc-dfs-sdk-6.7.zip file to a location of your choosing. This location will be referred to as %DFS_SDK%. The root directory of the samples, %DFS_SDK%/samples will be referred to as %SAMPLES_LOC%.

2. View the %SAMPLES_LOC%/Services/HelloWorldService/src/service/com/service/example/HelloWorldService.java file. Notice the annotation just before the class declaration. This annotation is used by the DFS SDK tools to generate code and package the service:

   ```
   @DfsPojoService(targetNamespace = "http://example.service.com",
     requiresAuthentication = true)
   ```

   Note that this service would also work if requiresAuthentication were set to false; we set it to true only to demonstrate the more typical setting in a DFS service. For more information on annotations, see Annotating a service, page 124.

3. View the %SAMPLES_LOC$/Services/HelloWorldService/build.properties file. Notice the module.name (example) and context.root (services) properties. The values that are defined for these properties along with the class name are used to create the service URL. For example, using the default values in build.properties, the address of the Hello World service will be http://host:port/services/example/HelloWorldService.

4. View the %SAMPLES_LOC/Services/HelloWorldService/build.xml file. The artifacts and package targets build the service. They call the generateModel, generateArtifacts, buildService, and packageService tasks. For more information on these tasks, see Chapter 9, The DFS Build Tools

   In the packageService task, modify the library settings to reference all of the jar files in the dfc folder, as follows:

   ```
   <pathelement location="${dfs.sdk.libs}/dfc/*.jar"/>
   ```

5. Edit the %DFS_SDK%/etc/dfc.properties file and specify, at a minimum, correct values for the dfc.docbroker.host[0] and dfc.docbroker.port[0] properties. The dfc.properties file is packaged with the service during the build.

   The Hello World service requires access to the connection broker to authenticate the user. If the service did not require authentication, you would not have to set values in the dfc.properties file, because the Hello World service does not access a repository. Authentication is intentionally set to true to demonstrate the steps necessary to build an authenticated service. As an optional exercise, you can choose to change the annotation for the service and not require authentication (requiresAuthentication=false). In this case, you do not have to specify anything in the dfc.properties file, because the service never interacts with a connection broker.

6. From the command prompt, enter the following commands:

   ```
   cd %SAMPLES_LOC%/Services/HelloWorldService
   ant artifacts package
   ```

   The %SAMPLES_LOC%/Services/HelloWorldService/build/example.ear file should appear after the build is successful.

7. Copy the %SAMPLES_LOC%/Services/HelloWorldService/build/example.ear file to the %DOCUMENTUM_HOME%\jboss4.2.0\server\DctmServer_DFS\deploy directory (default deploy location for the DFS application server) or to wherever you deploy your web applications.

8. Restart the DFS application server. Once the server is restarted, the Hello World service should be addressable at http://<host>:<port>/services/example/HelloWorldService.

You can test the deployment by typing http://<host>:<port>/services/example/HelloWorldService?wsdl into a browser (replacing <host>:<port> with the correct domain).

Testing the Hello World service with the sample consumer

The Hello World example also comes with a sample consumer that calls the Hello World service's sayHello operation. To test the Hello World service:

1. Edit the %SAMPLES_LOC%/Services/HelloWorldService/src/client-remote/src/com/client/samples/HelloWorldClient.java file and specify correct values for the following variables:
   - contextRoot
   - moduleName
   - repository
   - user
   - password
2. View the %SAMPLES_LOC/Services/HelloWorldService/build.xml file. You will run the compile and run.client targets. Notice that the these targets include the example.jar file in the classpath. This jar file is generated during the building of the service. All consumers that will utilize the productivity layer (DFS SDK) must be provided with this jar file. The example.jar file can be included for local or remote consumers and the example-remote.jar file can be included only for remote consumers.
3. From the command prompt, enter the following commands:

```
cd %SAMPLES_LOC%/HelloWorldService
    ant run
```

   The run target runs both the compile and run.client targets. After the run target completes, you should see the string "response = Hello John", which indicates a successful call to the service.

Exploring AcmeCustomService

AcmeCustomService serves as a minimal example that demonstrates fundamental techniques that you need to develop your own services. This section provides a brief tour of the AcmeCustomService sample and shows you how to generate, deploy, and test the AcmeCustomService service. AcmeCustomService is located in the %DFS_SDK%/samples/AcmeCustomService directory where %DFS_SDK% is the location of your DFS SDK.

Overview of AcmeCustomService

The AcmeCustomService class displays basic concepts on how to write a custom DFS service. It gets a DFC session manager to begin using the DFC API, invokes (chains in) a core DFS service from your custom service (the Schema service), and populates an AcmeCustomInfo object with information that is obtained from these two sources. For information on how the class for the AcmeCustomInfo object is implemented and annotated, see Data type and field annotation, page 125 .The service also contains an operation to test the custom exception handling functionality of DFS.

The getAcmeServiceInfo method gets a DFC session manager and populates the AcmeServiceInfo object with data from the session manager:

```
IDfSessionManager manager = DfcSessionManager.getSessionManager();
```

A reference to AcmeCustomService's own service context is then obtained. Notice the use of the getContext method rather than the newContext method. The getContext method enables the calling service to share identities and any other service context settings with the invoked service:

```
IServiceContext context = ContextFactory.getInstance().getContext();
```

The context, explicit service module name ("core"), and context root ("http://127.0.0.1:8888/services") is passed to the getRemoteService method to get the Schema service. (You may need to change the hardcoded address of the remotely invoked schema service, depending on your deployment.)

```
ISchemaService schemaService
            = ServiceFactory.getInstance()
            .getRemoteService(ISchemaService.class,
                              context,
                              "core",
                              "http://127.0.0.1:8888/services");
```

Note: It is also possible to invoke DFS services locally rather than remotely in your custom service, if the service JARs from the SDK have been packaged in your custom service EAR file. There are a number of potential advantages to local invocation of the DFS services—improved performance, the ability to share a registered service context between your custom service and the DFS services, and the ability to include invocation of the DFS services within a transaction. (see Transactions in a custom service, page 128). To enable local invocation of services, make sure that the local JAR files for the service (for example emc-dfs-services.jar for core service and emc-dfs-search-service.jar for search services) are included in the call to the packageService task in the Ant build.xml. Do not include *-remote.jar files.

The getSchemaInfo operation of the Schema service is called and information from this request is printed out:

```
schemaInfo = schemaService.getSchemaInfo(repositoryName, null, operationOptions);
```

The testExceptionHandling() method demonstrates how you can create and throw custom exceptions. The method creates a new instance of CustomException and throws it. The client side runtime catches the exception and recreates it on the client, preserving all of the custom attributes. You must follow certain guidelines to create a valid custom exception that can be thrown over the wire to the client. For more information on how to create a DFS custom exception, see DFS exception handling, page 130. The CustomException class is located in the %SAMPLES_LOC%/AcmeCustomService/src/service/com/acme/services/samples/common directory.

```
package com.acme.services.samples.impl;

import com.acme.services.samples.common.AcmeServiceInfo;
import com.documentum.fc.client.IDfSessionManager;
import com.documentum.fc.client.IDfSessionManagerStatistics;
import com.emc.documentum.fs.datamodel.core.OperationOptions;
import com.emc.documentum.fs.datamodel.core.schema.SchemaInfo;
import com.emc.documentum.fs.rt.annotations.DfsPojoService;
import com.emc.documentum.fs.rt.context.ContextFactory;
import com.emc.documentum.fs.rt.context.IServiceContext;
import com.emc.documentum.fs.rt.context.ServiceFactory;
import com.emc.documentum.fs.rt.context.impl.DfcSessionManager;
import com.emc.documentum.fs.services.core.client.ISchemaService;
import com.acme.services.samples.common.CustomException;
```

```
import java.util.ArrayList;
import java.util.Iterator;

@DfsPojoService(targetNamespace = "http://samples.services.acme.com")
public class AcmeCustomService
{

  public void testExceptionHandling() throws Exception
  {
   System.out.println("Throwing the custom exception:\n");
   throw new CustomException("testExceptionHandling() was called",
   System.currentTimeMillis(), "John Smith",
   new Exception("Chained exception"));
  }

    public AcmeServiceInfo getAcmeServiceInfo() throws Exception
    {
        // use DFC
        IDfSessionManager manager = DfcSessionManager.
                                    getSessionManager();
        IDfSessionManagerStatistics managerStatistics =
                              manager.getStatistics();

        AcmeServiceInfo acmeServiceInfo = new AcmeServiceInfo();
        acmeServiceInfo.setHasActiveSessions(
          managerStatistics.hasActiveSessions());
        acmeServiceInfo.setSessionPoolingActive(
          managerStatistics.isSessionPoolActive());

        ArrayList<String> docbaseList = new ArrayList<String>();
        Iterator docbaseIterator = managerStatistics.getDocbases();
        while (docbaseIterator.hasNext())
        {
            docbaseList.add((String)docbaseIterator.next());
        }
        acmeServiceInfo.setRepositories(docbaseList);

        // use core service
        IServiceContext context = ContextFactory.getInstance().
                                              getContext();
        ISchemaService schemaService
            = ServiceFactory.getInstance()
            .getRemoteService(ISchemaService.class,
                              context,
                              "core",
                              "http://127.0.0.1:8888/services");
        OperationOptions operationOptions = null;
        SchemaInfo schemaInfo;
        String repositoryName = docbaseList.get(0);
        schemaInfo = schemaService.getSchemaInfo(repositoryName,
          null, operationOptions);
        acmeServiceInfo.setDefaultSchema(schemaInfo.getName());

        return acmeServiceInfo;
    }
}
```

Preparing to build AcmeCustomService

Before you build AcmeCustomService, ensure that certain properties are set correctly in the %DFS_SDK%/samples/Services/AcmeCustomService/build.properties and %DFS_SDK%/etc/dfc.properties file:

1. Edit the %DFS_SDK%/samples/Services/AcmeCustomService/build.properties file and ensure that the values for the dfs.sdk.home and autodeploy.dir properties are correct. For more information see build.properties, page 139.
2. Edit the %DFS_SDK%/etc/dfc.properties file and specify the correct values for dfc.docbroker.host[0] and dfc.docbroker.port[0] at a minimum. For more information see dfc.properties, page 140
3. Edit the %DFS_SDK%/samples/Services/AcmeCustomService/src/service/com/acme/services/samples/impl/AcmeCustomService.java file and modify the call to the Schema service if the host and port are incorrect. The code assumes that you have DFS deployed at localhost on port 8888. To find out if DFS is deployed at the location that you specify in the code, request the Schema service WSDL at http://host:port/services/core/SchemaService?wsdl. If the WSDL is returned, the service is deployed as expected.

```
ISchemaService schemaService
            = ServiceFactory.getInstance()
            .getRemoteService(ISchemaService.class,
            context,"core",
            "http://localhost:8888/services");
```

build.properties

The build.properties file contains property settings that are required by the Ant build.xml file. To build AcmeCustomService, there is no need to change any of these settings, unless you have moved the AcmeCustomService directory to another location relative to the root of the SDK. In this case, you need to change the dfs.sdk.home property. If you want AcmeCustomService to be automatically copied to the deploy directory of the JBoss application server when you run the deploy target, specify the directory in the autodeploy.dir property.

```
# EMC DFS SDK 6.7 build properties template
dfs.sdk.home=../..
# Compiler options
compiler.debug=on
compiler.generate.no.warnings=off
compiler.args=
compiler.max.memory=128m
fork = true
nonjava.pattern = **/*.java,**/.svn,**/_svn
# Establish the production and tests build folders
build.folder = build
module.name = samples
context.root = services

#Debug information
debug=true
keep=true
verbose=false
extension=true
```

```
#Deploy params
autodeploy.dir=C:/Documentum/jboss4.2.0/server/DctmServer_DFS/deploy
```

dfc.properties

The service-generation tools package a copy of dfc.properties within the service EAR file. The properties defined in this dfc.properties file configure the DFC client utilized by the DFS service runtime. The copy of dfc.properties is obtained from the DFS SDK etc directory. The dfc.properties must specify the address of a docbroker that can provide access to any repositories required by the service and its clients, for example:

```
dfc.docbroker.host[0]=10.8.13.190
dfc.docbroker.port[0]=1489
```

The docbroker can be specified by IP address or by computer name.

Building and deploying the AcmeCustomService

The %DFS_SDK%/samples/Services/AcmeCustomService/build.xml file contains the Ant tasks to generate service artifacts and deployable service archive files from the Java source files and configuration files. This procedure describes how to build and deploy AcmeCustomService to the JBoss application server that is bundled with DFS. To build and deploy the AcmeCustomService:

1. From the %DFS_SDK%/samples/Services/AcmeCustomService directory in the command prompt, enter the following command:

   ```
   ant clean artifacts package deploy
   ```

   You can also run the targets individually and examine the output of each step. For more information on the targets, see build.xml, page 140. The deploy target copies the EAR file to the directory that you specified in the build.properties file. JBoss should automatically detect the EAR file and deploy it. If this does not happen, restart the server.

2. When the EAR file is done deploying, request the AcmeCustomService WSDL by going to http://host:port/services/samples/AcmeCustomService?wsdl. A return of the WSDL indicates a successful deployment. The default port for the JBoss application server is 8888.

AcmeCustomService is now deployed and ready to accept consumer requests. You can run the sample consumer to test AcmeCustomService's functionality.

build.xml

The Ant build.xml file drives all stages of generating and deploying the custom service. It contains the targets shown in Table 21, page 141, which can be run in order to generate and deploy the custom service.

Table 21. Sample service build.xml Ant targets

Ant target	Description
clean	Deletes the build directory in which the service binaries are generated.
artifacts	Executes the generateModel task (see generateModel task, page 146) to create the service definition; executes the generateArtifacts task (see generateArtifacts task, page 147) to generate the service class files, WSDL, and XML schemas.
package	Executes the buildService task (see buildService task, page 148) to build the service jar files for remote and local invocation; executes the packageService task (see packageService task, page 149) to build the service EAR file for deployment to the application server.
deploy	Copies the EAR file generated by the packageService task to the JBoss deploy directory defined as a directory path in the autodeploy.properties file.
run	Compiles and runs the service test consumer.

Running the AcmeCustomService test consumer

The DFS SDK includes two test consumers of AcmeCustomService: one Java consumer, which can invoke the custom service locally or remotely, and a C# sample consumer that invokes the service remotely.

The AcmeCustomService build.xml file includes an Ant target that compiles and runs the Java test service consumer. As delivered, the consumer calls the service remotely, but it can be altered to call the service locally by commenting out the serviceFactory.getRemoteService method and uncommenting the serviceFactory.getLocalService method.

Note: If you are developing consumers in .NET or using some other non-Java platform, you might want to test the service using the Java client library, because you can use local invocation and other conveniences to test your service more quickly. However, it is still advisable to create test consumers on your target consumer platform to confirm that the JAXB markup has generated a WSDL from which your tools generate acceptable proxies.

To run the AcmeCustomService test consumer:

1. In Java, edit the %DFS_SDK%/samples/Services/AcmeCustomService/src/client-remote/com/acme/services/samples/client/AcmeCustomServiceDemo class and provide appropriate values for the following code:

Example 8-4. Java: Service test consumer hardcoded values

```
// replace these values
repoId.setRepositoryName("YOUR_REPOSITORY_NAME");
repoId.setUserName("YOUR_USER_NAME");
repoId.setPassword("YOUR_PASSWORD");
```

   In .NET, edit the %DFS_SDK%/samples/Services/AcmeCustomService/src/client-remote.net/Program.cs class and provide appropriate values for the following code.

Example 8-5. C#: Service test consumer hardcoded values

```
repoId.RepositoryName = "yourreponame";
repoId.UserName = "yourusername";
repoId.Password = "yourpwd";
```

2. Edit the Java or .NET code and specify values for the following code. The call to getRemoteService assumes that the instance of JBoss that you are deploying to is running on the local host on port 8888. Change these values if they are incorrect.

Example 8-6. Java: Sample service invocation

```
// modify if your app server is running somewhere else
mySvc = serviceFactory.getRemoteService(
     IAcmeCustomService.class, context, "samples",
     http://localhost:8888/services"
);
```

Example 8-7. C#: Sample service invocation

```
mySvc = serviceFactory.GetRemoteService<IAcmeCustomService>(
     context, "samples",
     http://localhost:8888/services");
```

3. Run the Java consumer at the command prompt from the %DFS_SDK%/samples/Services/AcmeCustomService directory:

   ```
   ant run
   ```

   Run the .NET consumer in Visual Studio.

dfs-client.xml

The dfs-client.xml file contains properties used by the Java client runtime for service addressing. The AcmeCustomService test consumer provides the service address explicitly when instantiating the service object, so does not use these defaults. However, it's important to know that these defaults are available and where to set them. The %DFS_SDK%/etc folder must be included in the classpath for clients to utilize dfs-client.xml. If you want to place dfs-client.xml somewhere else, you must place it in a directory named config and its parent directory must be in the classpath. For example, if you place the dfs-client.xml file in the c:/myclasspath/config/dfs-client.xml directory, add c:/myclasspath to your classpath.

```
<DfsClientConfig defaultModuleName="samples"
                 registryProviderModuleName="samples">
    <ModuleInfo name="samples"
                protocol="http"
                host="127.0.0.1"
                port="8888" contextRoot="services">
    </ModuleInfo>
</DfsClientConfig>
```

Note: .NET consumers use app.config instead of dfs-client.xml, as application configuration infrastructure is built into .NET itself. See Configuring a .NET client, page 68.

Creating a service from a WSDL

DFS allows you to create services with a WSDL-first approach by using the DFS build tools to generate code from a WSDL. The DFS build tools provide the generateService task, which takes a WSDL as input and outputs a Java service stub that you can implement. The data model classes are also generated from the given WSDL. After generating the service stub and data model classes, you can implement, build, and package your service in the same way as in code first service development. To create a service from a WSDL:

1. Call the generateService task and specify, at a minimum, values for wsdllocation and destDir. For more information on the generateService task, see generateService task, page 149

   ```
   <target name ="generateServiceStubs">
      <generateService
          wsdlUri="http://host:port/contextroot/module/
                   ServiceName?wsdl"
          destDir="outputDir" />
    </target>
   ```

   The service stub and the data model classes are placed in a directory structure that is determined by their target namespaces. For example, if the WSDL has a target namespace of http://module.contextroot.fs.documentum.emc.com, the service stub will be placed in the *outputDir*/com/emc/documentum/fs/contextroot/module/impl directory. The data model classes are output in a similar fashion.

2. Edit the service stub, which is located in the *targetNamespace*/impl directory. Each method in the service stub throws an UnsupportedOperationException as a placeholder. It is up to you to implement the logic and exception handling for each method.

At least one method should throw a ServiceException. This is because DFS encapsulates runtime exceptions (exceptions that are not declared in the throws clause) in a ServiceException on the server and passes it to the client. Throwing a ServiceException ensures that the class included with the service when you build and package it.

3. Once the service is implemented, you can use the DFS build tools to build and package the service.

Chapter 9

The DFS Build Tools

The DFS build tools rely on a set of Ant tasks that can help you create and publish services and generate client support for a service. When developing your own services, you might need to extend the classpaths of these tasks to include libraries that are required by your service. To see how the tasks are used in the context of a build environment, examine the build.xml file in the AcmeCustomService sample.

Apache Ant

The DFS design-time tools for generating services rely on Apache Ant, and were created using Ant version 1.7.0. You will need to have installed Ant 1.7.0 or higher in your development environment to run the DFS tools. Make sure that your path environment variable includes a path to the Ant bin directory.

Avoiding out of memory errors when running Ant scripts

To avoid out of memory errors when running DFS build tools in Ant, you may need to set the ANT_OPTS environment variable. For running in a UNIX shell, DOS, or Cygwin, set ANT_OPTS to "-Xmx512m -XX:MaxPermSize=128m" to solve an OutOfMem and PermGen space error.

For running in Eclipse, set the Window -> Preferences -> Java -> Installed JREs -> Edit > Default VM Parameter to "Xmx512m -XX:MaxPermSize=128m"

Referencing the tasks in your Ant build script

The DFS SDK provides an Ant taskdef file that defines all of the Ant tasks that come with the DFS build tools. To call the Ant tasks, include the taskdef file in your Ant build script (requires Ant 1.7) like in the following example:

```
<taskdef file="${dfs.sdk.libs}/emc-dfs-tasks.xml"/>
```

You can then call the individual tasks as described in the sample usage for each task:

generateModel task

The generateModel Ant task takes the annotated source code as input to the tools and generates a service model XML file named *{contextRoot}-{serviceName}*-service-model.xml, which describes service artifacts to be generated by subsequent processes. The generateModel task is declared in the emc-dfs-tasks.xml task definition file as follows:

```
<taskdef name="generateModel" classname="com.emc.documentum.fs.tools.
                                          GenerateModelTask">
    <classpath location="${dfs.sdk.home}/lib/java/emc-dfs-tools.jar"/>
    <classpath location="${dfs.sdk.home}/lib/java/emc-dfs-rt.jar"/>
    <classpath location="${dfs.sdk.home}/lib/java/utils/aspectjrt.jar"/>
</taskdef>
```

The generateModel task takes the following arguments:

Argument	Description
contextRoot	Attribute representing the root of the service address. For example, in the URL http://127.0.0.1:8080/services/ "services" signifies the context root.
moduleName	Attribute representing the name of the service module.
destDir	Attribute representing a path to a destination directory into which to place the output service-model XML.
<services>	An element that provides a list (a <fileset>), specifying the annotated source artifacts.
<classpath>	An element providing paths to binary dependencies.

In the sample service build.xml, the generateModel task is configured and as follows:

```
<generateModel contextRoot="${context.root}"
               moduleName="${module.name}"
               destdir="${project.artifacts.folder}/src">
     <services>
        <fileset dir="${src.dir}">
            <include name="**/*.java"/>
        </fileset>
    </services>
    <classpath>
        <pathelement location="${dfs.sdk.libs}/dfc/dfc.jar"/>
        <path refid="project.classpath"/>
    </classpath>
```

```
</generateModel>
```

generateArtifacts task

The generateArtifacts Ant task takes the source modules and service model XML as input, and creates all output source artifacts required to build and package the service. These include the service interface and implementation classes, data and exception classes, runtime support classes, and service WSDL with associated XSDs.

The generateArtifacts task is declared in the emc-dfs-tasks.xml task definition file as follows:

```
<taskdef name="generateArtifacts"
         classname="com.emc.documentum.fs.tools.build.ant.
                    GenerateArtifactsTask">
    <classpath location="${dfs.sdk.home}/lib/java/emc-dfs-rt.jar"/>
    <classpath location="${dfs.sdk.home}/lib/java/emc-dfs-tools.jar"/>
    <classpath location="${dfs.sdk.home}/lib/java/dfc/aspectjrt.jar"/>
</taskdef>
```

The generateArtifacts task takes the following arguments.

Argument	Description
serviceModel	Attribute representing a path to the service model XML created by the generateModel task.
destDir	Attribute representing the folder into which to place the output source code. Client code is by convention placed in a "client" subdirectory, and server code in a "ws" subdirectory.
<src>	Element containing location attribute representing the location of the annotated source code.
<classpath>	An element providing paths to binary dependencies.

In the sample service build.xml, the generateArtifacts task is configured and executed as follows:

```
 <generateArtifacts
     serviceModel=
"${project.artifacts.folder}/src/${context.root}-${module.name}-service-
                                                          model.xml"
   destdir="${project.artifacts.folder}/src"
   api="rich">
     <src location="${src.dir}"/>
     <classpath>
           <path location="${basedir}/${build.folder}/classes"/>
           <path location="${dfs.sdk.home}/lib/emc-dfs-rt.jar"/>
           <path location="${dfs.sdk.home}/lib/emc-dfs-services.jar"/>
           <pathelement location="${dfs.sdk.home}/lib/dfc/dfc.jar"/>
           <fileset dir="${dfs.sdk.home}/lib/ucf">
              <include name="**/*.jar"/>
           </fileset>
           <path location="${dfs.sdk.home}/lib/jaxws/jaxb-api.jar"/>
           <path location="${dfs.sdk.home}/lib/jaxws/jaxws-tools.jar"/>
           <path location="${dfs.sdk.home}/lib/commons/commons-lang-
                                                        2.1.jar"/>
           <path location="${dfs.sdk.home}/lib/commons/commons-io-
                                                        1.2.jar"/>
      </classpath>
</generateArtifacts>
```

buildService task

The buildService tasks takes the original annotated source, as well as output from the buildArtifacts task, and builds two JAR files:

- A remote client package: {moduleName}-remote.jar
- A server (and local client) package: {moduleName}.jar

The buildService task is declared in the emc-dfs-tasks.xml task definition file as follows:

```
<taskdef name="buildService" classname="
  com.emc.documentum.fs.tools.build.ant.BuildServiceTask">
  <classpath location="${dfs.sdk.home}/lib/java/emc-dfs-
                                          tools.jar"/>
  <classpath location="${dfs.sdk.home}/lib/java/emc-
                                          dfs-rt.jar"/>
  <classpath location="${dfs.sdk.home}/lib/java/dfc/
                                       aspectjrt.jar"/>
</taskdef>
```

The buildService task takes the following arguments.

Argument	Description
serviceName	Attribute representing the name of the service module.
destDir	Attribute representing the folder into which to place the output JAR files.
<src>	Element containing location attribute representing the locations of the input source code, including the original annotated source and the source output by generateArtifacts.
<classpath>	Element providing paths to binary dependencies.

In the sample service build.xml, the buildService task is configured as follows:

```
<buildService serviceName="${service.name}"
              destDir="${basedir}/${build.folder}"
              generatedArtifactsDir="${project.resources.folder}">
    <src>
       <path location="${src.dir}"/>
       <path location="${project.artifacts.folder}/src"/>
    </src>

    <classpath>
       <pathelement location="${dfs.sdk.home}/lib/dfc/dfc.jar"/>
       <path refid="project.classpath"/>
    </classpath>
 </buildService>
```

Method name conflict on remote client generation

When generating multiple services in the same package that have the same method names, the generation on the client side overwrites the generated classes where there are name conflicts. The services must be in separate packages or have different method names.

packageService task

The packageService packages all service artifacts into an EAR file that is deployable to the application server. The packageService task is declared in the emc-dfs-tasks.xml task definition file as follows:

```
<taskdef name="packageService"
    classname="com.emc.documentum.fs.tools.build.ant.
                                PackageServiceTask">
    <classpath location="${dfs.sdk.home}/lib/java/
                                emc-dfs-tools.jar"/>
    <classpath location="${dfs.sdk.home}/lib/java/
                                emc-dfs-rt.jar"/>
    <classpath location="${dfs.sdk.home}/lib/java/dfc/aspectjrt.jar"/>
</taskdef>
```

The packageService task takes the following arguments:

Argument	Description
deployment-Name	Attribute representing the name of the service module. You can specify a .ear or .war (for Tomcat deployment) file extension depending on the type of archive that you want.
destDir	Attribute representing the folder into which to place the output archives.
generatedArti-factsFolder	Path to folder in which the WSDL and associated files have been generated.
<libraries>	Element specifying paths to binary dependencies.
<resources>	Element providing paths to resource files.

In the sample service build.xml, the packageService task is configured as follows:

```
<packageService deploymentName="${service.name}"
                destDir="${basedir}/${build.folder}"
                generatedArtifactsDir="${project.resources.folder}">
    <libraries>
        <pathelement location="${basedir}/${build.folder}/
                                    ${service.name}.jar"/>
        <pathelement location="${dfs.sdk.home}/lib/emc-dfs-rt.jar"/>
        <pathelement location="${dfs.sdk.home}/lib/emc-dfs-
                                    services.jar"/>
        <pathelement location="${dfs.sdk.home}/lib/dfc/dfc.jar"/>
    </libraries>
    <resources>
        <path location="${dfs.sdk.home}/etc/dfs.properties"/>
    </resources>
 </packageService>
```

generateService task

The generateService Ant task takes a given WSDL as input and generates a DFS annotated service stub and its data model. You can use these generated files to create your custom service and input them into the DFS runtime tools to generate and package your service. The location of the WSDL can either be local (file://) or remote (http://). The generateService task is declared in the emc-dfs-tasks.xml task definition file as follows:

```
<taskdef name="generateService"
```

```
  classname="com.emc.documentum.fs.tools.GenerateServiceTask">
  <classpath location="${dfs.sdk.home}/lib/java/emc-dfs-
                                                tools.jar"/>
  <classpath location="${dfs.sdk.home}/lib/java/emc-dfs-rt.jar"/>
  <classpath location="${dfs.sdk.home}/lib/java/utils/
                                        aspectjrt.jar"/>
</taskdef>
```

The generateService task takes the following arguments:

Argument	Description
wsdlUri	The local (file://) or remote (http://) location of the WSDL
destDir	Attribute representing the folder into which to place the output source code.
debug	The debug mode switch (“on” or “off”)
verbose	The verbose mode switch (“on” or “off”)

You can call the generateService task within a target as follows:

```
<generateService
 wsdllocation="${wsdl.location}"
 destDir="${dest.dir}"
 verbose="true"
 debug="false"/>
```

generateRemoteClient task

The generateRemoteClient Ant task takes a given WSDL as input and generates client proxies for the service described by the WSDL. The client proxies that are generated differ from the client libraries that are provided in the DFS client productivity layer. For more information on the differences, see WSDL-first consumption of services, page 56. You can use these generated files to help you create your consumer. The location of the WSDL can either be local (file://) or remote (http://). The generateRemoteClient task is declared in the emc-dfs-tasks.xml task definition file as follows:

```
<taskdef name="generateRemoteClient"
   classname="com.emc.documentum.fs.tools.GenerateRemoteClientTask">
   <classpath location="${dfs.sdk.home}/lib/java/emc-dfs-tools.jar"/>
   <classpath location="${dfs.sdk.home}/lib/java/emc-dfs-rt.jar"/>
   <classpath location="${dfs.sdk.home}/lib/java/utils/aspectjrt.jar"/>
</taskdef>
```

The generateRemoteClient task takes the following arguments:

Argument	Description
wsdlUri (required)	The local (file://) or remote (http://) location of the WSDL
destdir (required)	Attribute representing the folder into which to place the output source code.
serviceProtocol	Either http or https (default is http)
serviceHost	The host where the service is located. This value defaults to the WSDL host, so if the WSDL is a local file, specify the host where the service is located.

Argument	Description
servicePort	The port of the service host. This value defaults to the WSDL host port, so if the WSDL is a local file, specify the port where the service is located.
serviceContext-Root	The context root where the service is deployed. This value defaults to the WSDL context root, so if the WSDL is a local file, specify the context root where the service is located.
serviceModule-Name	The name of the service module. This value defaults to the WSDL service module, so if the WSDL is a local file, specify the module where the service is located.

All attributes except for wsdlUri and destdir are used to override values that are generated from the WSDL by the generateRemoteClient task.

You can call the generateRemoteClient task within a target as follows:

```
<generateRemoteClient
   wsdlUri="${wsdl.location}"
   destdir="${dest.dir}"
   serviceProtocol="true"
   serviceHost="localhost"
   servicePort="8888"
   serviceContextRoot="services"
   serviceModuleName="core" />
```

generatePublishManifest task

The generatePublishManifest task generates an XML manifest file that is taken as input by the DFS Publish Utility. The generatePublishManifest task is declared in the emc-dfs-tasks.xml task definition file as follows:

```
 <taskdef name="generatePublishManifest"
 classname="com.emc.documentum.fs.tools.registry.ant.GeneratePublishManifestTask">
  <classpath location="${dfs.sdk.home}/lib/java/emc-dfs-tools.jar" />
  <classpath location="${dfs.sdk.home}/lib/java/emc-dfs-rt.jar" />
  <classpath location="${dfs.sdk.home}/lib/java/utils/aspectjrt.jar" />
  <classpath location="${dfs.sdk.home}/lib/java/jaxr/jaxr-impl.jar" />
  <classpath location="${dfs.sdk.home}/lib/java/jaxr/jaxr-api.jar" />
  <classpath location="${dfs.sdk.home}/lib/java/jaxws/jaxb-api.jar" />
  <classpath location="${dfs.sdk.home}/lib/java/jaxws/jaxb-impl.jar" />
  <classpath location="${dfs.sdk.home}/lib/java/jaxws/jaxb1-impl.jar" />
  <classpath location="${dfs.sdk.home}/lib/java/jaxws/jsr181-api.jar" />
  <classpath location="${dfs.sdk.home}/lib/java/jaxws/jsr173_api.jar" />
 <classpath location="${dfs.sdk.home}/lib/java/commons/commons-lang-2.1.jar" />
 </taskdef>
```

Argument	Description
file	The output service manifest file
organization	The organization to publish the services under

Argument	Description
<modules>	An element containing the location of the service model file of the services that you want to publish. You can have multiple <modules> elements. Each <modules> element contains <pathelement> elements that specify the location of the service model with the “location” attribute.
<publishset>	An element containing the services that you want to publish and the catalog and categories that you want the services to be under. Each <publishset> element contains <service>, <catalog> and <category> elements that define the services to publish and what catalog and category to publish them under. A catalog is a collection of services. You can create categories in each catalog to organize services. You can publish the same or different services in multiple catalogs and categorize them differently in each catalog by using multiple <publishset> elements. The <service> module accepts the “name” and “module” attribute, while the <catalog> and <category> elements accept the “name” attribute.

The generatePublishManifest task requires the <modules> and <publishset> as nested elements. An example of how to call the task is shown in the following sample:

```
<target name="generateManifest">
   <generatePublishManifest file="example-publish-manifest.xml" organization="EMC">
      <modules>
         <pathelement location="services-example-service-model.xml"/>
      </modules>
      <publishset>
         <service name="MyService1" module="example"/>
         <service name="MyService2" module="example"/>
         <catalog name="Catalog1"/>
         <category name="Category1"/>
         <category name="Category2"/>
      </publishset>
   </generatePublishManifest>
</target>
```

Packaging multiple service modules in one EAR file

You can package multiple service modules (a bundle of related services) in one EAR file for easier deployment, while maintaining different URLs for each service module. When service modules are

packaged in the same EAR file, they can all share a common cache of ServiceContext objects, which allows you to register a context once and call any service in any module within the same EAR file.

To package multiple service modules in one EAR file:

1. Run the generateModel Ant task for each of the service modules that you want to create. Ensure that you specify appropriate values for the following parameters:
   - contextRoot — Specify the same value for each service module that you want to create. A good value to use is "services."
   - moduleName — Specify different values for each service module that you want to create. This value is unique to each service module and creates different service URLs for each of your service modules.
   - destDir — Specify the same value for each service module that you want to create. Using the same destination directory ensures that the service modules get packaged into the same EAR file.

   For example, if you want to create service modules with URLs at /services/core, /services/bpm, and /services/search, your generateModel tasks might look like the following:

Example 9-1. generateModel task examples

```
<generateModel contextRoot="services"
        moduleName="core"
               destdir="build/services">
...
</generateModel>

<generateModel contextRoot="services"
        moduleName="bpm"
               destdir="build/services">
...
</generateModel>

<generateModel contextRoot="services"
        moduleName="search"
               destdir="build/services">
...
</generateModel>
```

2. Run the generateArtifacts Ant task for each service module that you want to create. For example, given the output generated by the example above, your generateArtifacts tasks should look like the following:

```
<generateArtifacts serviceModel="build/services/services-
core-service-model.xml"
  destdir="build/services">
  ...
</generateArtifacts>

<generateArtifacts  serviceModel="build/services/services-bpm-service-model.xml"
  destdir="build/services">
  ...
</generateArtifacts>

<generateArtifacts  serviceModel="build/services/services-search-service-model.xml"
  destdir="build/services">
  ...
</generateArtifacts>
```

3. Run the buildService Ant task for each service of the service modules that you want to create. For example, given the output generated by the examples above, your buildService tasks should look like the following:

```
<buildService serviceName="core"
  destdir="dist/services"
  generatedArtifactsDir="build/services">
...
</generateArtifacts>

<buildService serviceName="bpm"
  destdir="dist/services"
  generatedArtifactsDir="build/services">
  ...
</generateArtifacts>

<buildService serviceName="search"
  destdir="dist/services"
  generatedArtifactsDir="build/services">
  ...
</generateArtifacts>
```

4. Run the packageService task once to package all of your service modules together in the same EAR file. For example, given the output generated by the examples above, your packageService task should look like the following:

```
<packageService deploymentName="emc-dfs"
  destDir="dist/services"
  generatedArtifactsDir="build/services">
  ...
</packageService>
```

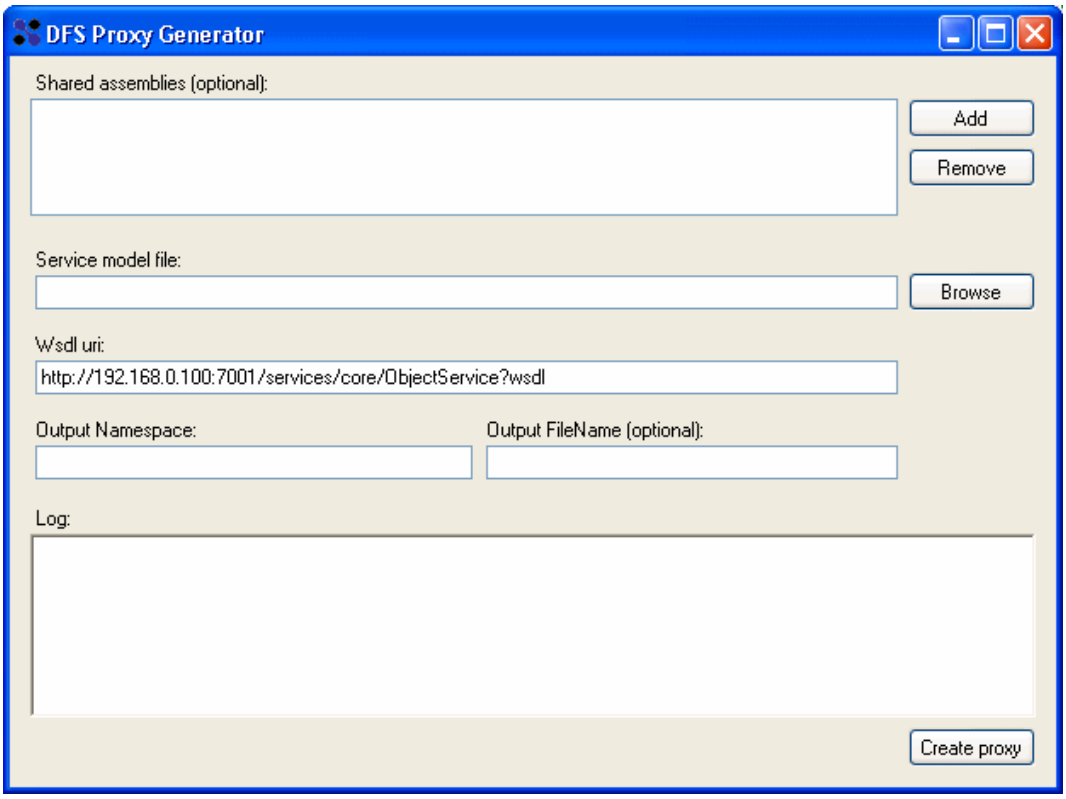
   You should now have all of your service modules packaged into one EAR file, which can be deployed in your application server.

Generating C# proxies

To generate C# proxies for the custom service, use the DfsProxyGen.exe utility supplied in the DFS SDK. DfsProxyGen is a Windows form application that generates C# proxies based on a DFS service WSDL and the generateArtifacts ant task (see generateArtifacts task, page 147). You will need to build and deploy the service before creating the C# proxies.

Note: You must run the DfsProxyGen utility locally and not from a network drive.

Figure 19. DfsProxyGen form



To generate C# proxies:

1. In the **Shared assemblies** field, add any shared assemblies used by the service. (There are none for AcmeCustomService.) For more information on this see Creating shared assemblies for data objects shared by multiple services, page 156.
2. In the **Service model file** field, browse to the service model file created by the generateArtifacts ant task. For AcmeCustomService this will be emc-dfs-sdk-6.7\samples\AcmeCustomService\resources\services-samples-service-model.xml.
3. In the **Wsdl uri** field, supply the name of the WSDL of the deployed service, for example http://localhost:7001/services/samples/AcmeCustomService?wsdl. Only URLs are permitted, not local file paths, so you should use the URL of the WSDL where the service is deployed.
4. In the Output namespace, supply a namespace for the C# proxy (for example samples.services.acme).
5. Optionally supply a value in the **Output FileName** field. If you don't supply a name, the proxy file name will be the same as the name of the service, for example AcmeCustomService.cs.
6. Click **Create proxy.**

The results of the proxy generation will appear in the **Log** field. If the process is successful, the name and location of the result file will be displayed.

Creating shared assemblies for data objects shared by multiple services

If you are creating multiple services that share data objects, you will want to generate C# proxies for the shared classes only once and place them in a shared assembly. The following procedure describes how to do this, based on the following scenario: you have created two services ServiceA and ServiceB; the two services share two data object classes, DataClass1 and DataClass2.

1. Run DfsProxyGen against the WSDL and service model file for ServiceA.

   This will generate the proxy source code for the service and its data classes DataClass1 and DataClass2.

2. Create a project and namespace for the shared classes, DataClass1 and DatasClass2, that will be used to build the shared assembly. Cut DataClass1 and DataClass2 from the generated proxies source generated for ServiceA, and add them to new source code file(s) in the new project.

3. Annotate the shared data classes using XmlSerializer's [XmlType()] attribute, specifying the WSDL namespace of the shared classes (for example XmlType(Namespace=http://myservices/datamodel/)].

4. Build an assembly from the shared datamodel project.

5. Run DfsProxyGen against the WSDL and service model for ServiceB, referencing the shared assembly created in step 4 in the **Shared assemblies** field.

Chapter 10

Content Transfer

DFS supports standard WS transfer modes (Base64 and MTOM), as well as proprietary technologies (UCF and ACS) that optimize transfer of content in distributed environments. This chapter will cover content transfer generally, with an emphasis on MTOM and Base64, as well as accessing content from ACS (Accelerated Content Services).

UCF content transfer is covered in a separate chapter (see Chapter 11, Content Transfer with Unified Client Facilities).

For related information see Content model and profiles, page 95.

Content transfer is an area where the productivity layer (PL) provides a lot of functionality, so there are significant differences in client code using the productivity layer and client code based on the WSDL alone. This chapter provides examples showing how to do it both ways. The WSDL-based samples in this chapter were written using JAX-WS RI 2.1.2.

This chapter includes the following topics:

Base64 content transfer

Base64 is an established encoding for transfer of opaque data inline within a SOAP message (or more generally within XML). The encoded data is tagged as an element of the xs:base64Binary XML schema data type. Base64 encoded data is not optimized, and in fact is known to expand binary data by a factor of 1.33x original size. This makes Base64 inefficient for sending larger data files. As a rule, it is optimal to use Base64 for content smaller than around 5K bytes. For larger content files, it is more optimal to use MTOM.

A DFS Base64 message on the wire encodes binary data within the Contents element of a DataObject. The following is an HTTP POST used to create an object with content using the DFS object service create method.

```
POST /services/core/ObjectService HTTP/1.1
SOAPAction: ""
Content-Type: text/xml;charset="utf-8"
Accept: text/xml, multipart/related, text/html, image/gif, image/jpeg,
        *; q=.2, */*; q=.2
User-Agent: JAX-WS RI 2.1.3-b02-
Host: localhost:8888
Connection: keep-alive
Content-Length: 996463

<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header>
    <wsse:Security
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
            wss-wssecurity-secext-1.0.xsd">
      <wsse:BinarySecurityToken
QualificationValueType="http://schemas.emc.com/documentum#ResourceAccessToken"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
           wss-wssecurity-utility-1.0.xsd"
wsu:Id="RAD">USITFERRIJ1L1C/10.13.33.174-1231455862108-42519027325738173642
      </wsse:BinarySecurityToken>
    </wsse:Security>
  </S:Header>
  <S:Body>
  <ns8:create xmlns:ns2="http://rt.fs.documentum.emc.com/"
              xmlns:ns3="http://core.datamodel.fs.documentum.emc.com/"
              xmlns:ns4="http://properties.core.datamodel.fs.documentum.
                         emc.com/"
              xmlns:ns5="http://content.core.datamodel.fs.documentum.
                         emc.com/"
              xmlns:ns6="http://profiles.core.datamodel.fs.documentum.
                         emc.com/"
              xmlns:ns7="http://query.core.datamodel.fs.documentum.emc.com/"
              xmlns:ns8="http://core.services.fs.documentum.emc.com/">
      <dataPackage>
        <ns3:DataObjects transientId="14615126"
                         type="dm_document">
          <ns3:Identity repositoryName="Techpubs"
                        valueType="UNDEFINED"/>
          <ns3:Properties isInternal="false">
            <ns4:Properties xmlns:xsi="http://www.w3.org/2001/
                                       XMLSchema-instance"
                            xsi:type="ns4:StringProperty"
                            isTransient="false"
                            name="object_name">
              <ns4:Value>MyImage</ns4:Value>
            </ns4:Properties>
            <ns4:Properties xmlns:xsi="http://www.w3.org/2001/
                                       XMLSchema-instance"
                            xsi:type="ns4:StringProperty"
                            isTransient="false"
                            name="title">
              <ns4:Value>MyImage</ns4:Value>
            </ns4:Properties>
            <ns4:Properties xmlns:xsi="http://www.w3.org/2001/
                                       XMLSchema-instance"
                            xsi:type="ns4:StringProperty"
                            isTransient="false"
                            name="a_content_type">
```

```
            <ns4:Value>gif</ns4:Value>
          </ns4:Properties>
        </ns3:Properties>
        <ns3:Contents xmlns:xsi="http://www.w3.org/2001/
                                 XMLSchema-instance"
                      xsi:type="ns5:BinaryContent"
                      pageNumber="0"
                      format="gif">
          <ns5:renditionType xsi:nil="true"/>
          <ns5:Value>R0lGODlhAAUABIc...[Base64-encoded content]
          </ns5:Value>
        </ns3:Contents>
      </ns3:DataObjects>
    </dataPackage>
  </ns8:create>
 </S:Body>
</S:Envelope>
```

MTOM content transfer

MTOM, an acronym for SOAP Message Transmission Optimization Mechanism, is a widely adopted W3C recommendation. For more information see http://www.w3.org/TR/soap12-mtom/. The MTOM recommendation and the related XOP (XML-binding Optimized Packaging) standard together describe how to send large binaries with a SOAP envelope as separate MIME-encoded parts of the message.

For most files, MTOM optimization is beneficial; however, for very small files (typically those under 5K), there is a serious performance penalty for using MTOM, because the overhead of serializing and deserializing the MTOM multipart message is greater than the benefit of using the MTOM optimization mechanism.

An MTOM message on the wire consists of a multipart message. The parts of the message are bounded by a unique string (the boundary). The first part of the message is the SOAP envelope. Successive parts of the message contain binary attachments. The following is an HTTP POST used to create an object with content using the DFS object service create method. Note that the Value element within the DFS Contents element includes an href pointing to the Content-Id of the attachment.

```
POST /services/core/ObjectService HTTP/1.1
Cookie: JSESSIONID=0FF8ED8C5E6C3E01DEA6A2E52571203E
SOAPAction: ""
Content-Type:
  multipart/related;
  start="<rootpart*27995ec6-ff6b-438d-b32d-0b6b78cc475f@example.jaxws.
         sun.com>";
  type="application/xop+xml";
  boundary="uuid:27995ec6-ff6b-438d-b32d-0b6b78cc475f";
  start-info="text/xml"
Accept: text/xml, multipart/related, text/html, image/gif, image/jpeg,
        *; q=.2, */*; q=.2
User-Agent: JAX-WS RI 2.1.3-b02-
Host: localhost:8888
Connection: keep-alive
Transfer-Encoding: chunked

[begin part including SOAP envelope]
--uuid:27995ec6-ff6b-438d-b32d-0b6b78cc475f
Content-Id: <rootpart*27995ec6-ff6b-438d-b32d-0b6b78cc475f@example.jaxws.
            sun.com>
```

```
Content-Type: application/xop+xml;charset=utf-8;type="text/xml"
Content-Transfer-Encoding: binary

<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header>
    <wsse:Security
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
            wss-wssecurity-secext-1.0.xsd">
      <wsse:BinarySecurityToken
QualificationValueType="http://schemas.emc.com/documentum#ResourceAccessToken"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
           wss-wssecurity-utility-1.0.xsd"
wsu:Id="RAD">USITFERRIJ1L1C/10.13.33.174-1231455862108-4251902732573817364-2
      </wsse:BinarySecurityToken>
    </wsse:Security>
  </S:Header>
  <S:Body>
    <ns8:create xmlns:ns8="http://core.services.fs.documentum.
                           emc.com/"
                xmlns:ns7="http://query.core.datamodel.fs.documentum.
                           emc.com/"
                xmlns:ns6="http://profiles.core.datamodel.fs.documentum.
                           emc.com/"
                xmlns:ns5="http://content.core.datamodel.fs.documentum.
                           emc.com/"
                xmlns:ns4="http://properties.core.datamodel.fs.documentum.
                           emc.com/"
                xmlns:ns3="http://core.datamodel.fs.documentum.emc.com/"
                xmlns:ns2="http://rt.fs.documentum.emc.com/">
      <dataPackage>
        <ns3:DataObjects transientId="8125444" type="dm_document">
          <ns3:Identity repositoryName="Techpubs"
                        valueType="UNDEFINED">
          </ns3:Identity>
          <ns3:Properties isInternal="false">
            <ns4:Properties xmlns:xsi="http://www.w3.org/2001/
                                       XMLSchema-instance"
                            xsi:type="ns4:StringProperty"
                            isTransient="false"
                            name="object_name">
              <ns4:Value>MyImage</ns4:Value>
            </ns4:Properties>
            <ns4:Properties xmlns:xsi="http://www.w3.org/2001/
                                       XMLSchema-instance"
                            xsi:type="ns4:StringProperty"
                            isTransient="false"
                            name="title">
              <ns4:Value>MyImage</ns4:Value>
            </ns4:Properties>
            <ns4:Properties xmlns:xsi="http://www.w3.org/2001/
                                       XMLSchema-instance"
                           xsi:type="ns4:StringProperty"
                           isTransient="false"
                           name="a_content_type">
              <ns4:Value>gif</ns4:Value>
            </ns4:Properties>
          </ns3:Properties>
          <ns3:Contents xmlns:xsi="http://www.w3.org/2001/
                                   XMLSchema-instance"
                        xsi:type="ns5:DataHandlerContent"
                        pageNumber="0"
                        format="gif">
            <ns5:renditionType xsi:nil="true"></ns5:renditionType>
            <ns5:Value>dc
```

```
              <Include xmlns="http://www.w3.org/2004/08/xop/include"
                     href="cid:85f284b5-4f2c-4e68-8d08-de160a5b47c6@example.
                          jaxws.sun.com"/>
            </ns5:Value>
          </ns3:Contents>
        </ns3:DataObjects>
      </dataPackage>
    </ns8:create>
  </S:Body>
</S:Envelope>

[boundary of binary content]
--uuid:27995ec6-ff6b-438d-b32d-0b6b78cc475f
Content-Id: <85f284b5-4f2c-4e68-8d08-de160a5b47c6@example.jaxws.sun.com>
Content-Type: image/gif
Content-Transfer-Encoding: binary

GIF89a[binary data...]

[end of multipart message]
--uuid:27995ec6-ff6b-438d-b32d-0b6b78cc475f--
0
```

Memory limitations associated with MTOM content transfer mode

The DFS .NET client is based on Windows Communication Framework (WCF), which provides two modes for MTOM content transfer: buffered and streaming. To enable streaming, WCF requires that the parameter that holds the data to be streamed must be the only parameter in the method (such as Get or create). This conflicts with the design of DFS, such that DFS can only use the MTOM buffer mode with a .NET client. This results in unusually high memory requirements, especially when trying to transfer large content payloads when ACS is unavailable, because the entire content must be buffered in memory before transfer. Normally a .NET client will use ACS if it available for content download operations (see Content types returned by DFS, page 164), so under typical conditions the memory limitation is not encountered. However, if ACS content is unavailable, or if the client attempts to upload a very large content stream to the server using MTOM content transfer mode, the server's capacity to buffer the content may be exceeded.

Workarounds

There are several options for working around this limitation:

- First, for content download operations, enable ACS/BOCS and make use of it. To ensure that the urlContent type is returned by DFS, use the urlReturnPolicy setting as described under Content types returned by DFS, page 164. The client can use the urlContent returned by DFS to request content transfer from the ACS server.
- For content upload operations, use UCF as the content transfer mode. UCF will orchestrate content transfer in both directions between the client and the ACS server.
- If you don't wish to use either of the preceding workarounds, make sure that both the DFS .NET client and JVM that runs DFS server have enough memory to buffer the

content. However, be aware that in this case the application will be limited to transfer of content in the range of hundreds of megabytes for a 32-bit JVM, because on most modern 32-bit Windows systems the maximum heap size will range from 1.4G to 1.6G (see http://www.oracle.com/technetwork/java/hotspotfaq-138619.html#gc_heap_32bit). Although this specific limitation will not apply to a 64-bit versions of Windows, the issue will still exist if you do not ensure that there is sufficient heap space to buffer very large objects in memory.

- You can create a custom service. Due to a WCF limitation (see http://msdn.microsoft.com/en-us/library/ms789010.aspx) wherein the stream data transfer mode is supported only when there is a single parameter in the web service method signature. Therefore, in the custom service, all parameters must be wrapped into a single custom class object containing all input parameters of a method as follows:

```
@DfsPojoService()
public class StreamingService
{
    public DataPackage create(DataRequest request) throws
    ServiceException
    {
        // DataRequest wraps DataPackage and OperationOptions,
        the DataPackage might contain large content
        // do something with the content uploaded
        ......
    }
}

@XmlType(name = "DataRequest", namespace =
     "http://streaming.fs.documentum.emc.com")
@XmlAccessorType(XmlAccessType.FIELD)
public class DataRequest
{
    private DataPackage dataPackage;
    private OperationOptions options;
    public DataPackage getDataPackage()
    {
        return dataPackage;
    }
    public void setDataPackage(DataPackage dataPackage)
    {
        this.dataPackage = dataPackage;
    }
   public OperationOptions getOptions()
    {
        return options;
    }
    public void setOptions(OperationOptions options)
    {
        this.options = options;
    }
}
```

  In the `App.config` file, to enable streaming, set the *transferMode* attribute of **DfsDefaultService** binding to **Streamed**.

Note:

- For downloading and uploading content, increase the time-out related attributes (closeTimeout, openTimeout, receiveTimeout, and sendTimeout) for DfsDefaultService binding based on the requirement.
- For content downloading, in App.config file, increase the value of the *maxReceivedMessageSize* attribute of **DfsDefaultService** binding to a larger value such as 1000000000 bytes. This

modification is required because the *maxReceivedMessageSize* attribute determines the maximum size, in bytes, for a message that can be received on a channel configured with streamed binding.

ContentTransferMode

The DFS ContentTransferMode is a setting that is used to influence the transfer mode used by DFS. This section discusses how this setting applies in various types of clients. For a closely related discussion of what types and content transfer mechanisms can be expected from DFS based on this setting and other factors, see Content types returned by DFS, page 164.

In the DFS WSDLs ContentTransferMode is defined as follows:

```
<xs:simpleType name="ContentTransferMode">
  <xs:restriction base="xs:string">
    <xs:enumeration value="BASE64"/>
    <xs:enumeration value="MTOM"/>
    <xs:enumeration value="UCF"/>
  </xs:restriction>
</xs:simpleType>
```

This type binds to enumerated types in Java and .NET clients.

The way the ContentTransferMode setting functions varies, depending on the type of client.

WSDL-based clients — In WSDL-based clients, ContentTransferMode influences the data type and content transfer format that DFS uses to marshall content in HTTP responses. In a WSDL-based client, ContentTransferMode only applies to content download from DFS, and in this context, only Base64 and MTOM are relevant. To use the UCF content transfer mechanism, you need to write client-side code that delegates the content transfer to UCF. WSDL-based clients in particular need to be aware that DFS will use UrlContent in preference to MTOM or Base64 to transfer content, if ACS is available (see Content types returned by DFS, page 164).

Remote productivity-layer clients — In a remote productivity-layer client, the ContentTransferMode setting affects both content upload and content download. During content upload (for example in a create or update operation), the PL runtime uses the transfer mechanism specified by ContentTransferMode and converts any Content object passed to the service proxy into a data type appropriate to the content transfer mechanism. On download, client code can convert the returned Content object to a file or byte array using PL convenience methods (that is Content#getAsFile and Content#getAsByteArray). If the ContentTransferMode setting is UCF, the DFS client runtime will delegate the transfer to UCF (for both upload and download).

Local productivity-layer clients — In the context of a local productivity-layer client, the ContentTransferMode setting is not significant—the transfer of content is handled by the local DFC client, so the SOAP transfer standards cannot be used. UCF content transfer is also not used in a local productivity-layer client. (However, note that UCF may be used *on the browser* in a web application that uses the local DFS API.) A local client does not need to include a ContentTransferProfile in the service context, and if does do so, the profile is ignored.

ContentTransferMode precedence

The ContentTransferMode setting can be stored locally on the client and passed to the service in a number of different contexts.

- In a ContentTransferProfile stored in the service context.
- In a ContentTransferProfile passed in OperationOptions

The value passed in OperationOptions will take precedence over the setting in the service context.

Note: Currently, you will not be able to use ContentTransferMode in Content instance.

Content types returned by DFS

The content types and content transfer format that DFS uses to return content to a remote client are influenced by, but not controlled by the ContentTransferMode setting.

Table 22. Content type returned to remote client

ContentTransferMode setting	Type returned ACS available	Type returned ACS unavailable
Base64	UrlContent	BinaryContent (Base64)
MTOM	UrlContent	DataHandlerContent (MTOM)
UCF	UcfContent	UcfContent

So as you can see, ContentTransferMode actually specifies the fallback remote transfer mechanism to use when ACS is not available. ACS may be unavailable globally because it was never deployed or because it is switched off. It may also be unavailable for a specific content format, depending on ACS configuration settings.

You can gain finer control over this behavior using the urlReturnPolicy property of ContentProfile. The value of urlReturnPolicy is an enum constant of type UrlReturnPolicy, as described in the following table:

Value	Behavior
ALWAYS	Return UrlContent where URL content is available; fail with exception where URL content is not available.
NEVER	Return actual content; never return UrlContent.
ONLY	Return UrlContent where URL content is available; return no content in DataObject where URL content is not available.
PREFER	Return UrlContent where URL content is available; return actual content where URL content is not available.

The default value is PREFER.

If you are writing a WSDL-only client that does not use the productivity layer, then your code needs to be aware at runtime of the content type returned by DFS and handle it appropriately. If you are using the productivity layer, the PL provides convenience methods that support handling all Content

subtypes in a uniform way, and transparently handle the streaming of UrlContent to the client. To see some sample code comparing these two approaches see Downloading content using Base64 and MTOM, page 167 and Downloading UrlContent, page 169.

UCF content transfer

UCF content transfer is a special case, in that it is a proprietary EMC technology, so client-side support is not provided in standard WS consumer frameworks. If you are using the productivity layer (either Java or .NET), this support is provided for you. UCF is integrated into the productivity layer, and the PL runtime transparently delegates content transfer to UCF if the client specifies UCF as the ContentTransferMode. If the client downloads content from DFS using this mechanism, DFS will return UcfContent in a response to a get request sent to the ObjectService. The UcfContent contains a path to the file downloaded by UCF, as well as other information about the content transfer—the actual content is downloaded in a separate HTTP response to a request sent by UCF. To get all this to happen without the productivity layer, you need to write an integration with the UCF client classes. For more information on UCF content transfer see Chapter 11, Content Transfer with Unified Client Facilities.

Content transfer using DFS locally

If you are writing a *local* productivity-layer client, then content transfer is handled by the underlying DFC client, which returns the content to the DFS client layer either as UrlContent or as FileContent—Base64 and MTOM cannot be used, because no XML marshalling or unmarshalling takes place. UCF content transfer is also not used in a local productivity-layer client. As in a remote productivity-layer client, the client code does not need to handle the streaming of UrlContent from ACS, but can just use the Content.getAsFile or Content.getAsByteArray methods.

Uploading content using Base64 or MTOM

When using a DFS service to upload content to the repository, the client needs to make sure that MTOM, if required, is enabled in the client framework, and that an appropriate data type is passed to the content transfer operation. The way you do this differs, depending on whether you are using a standard WSDL-based client or the DFS productivity layer.

If you are using a WSDL-based client, you will need to use the API provided by your framework to enable MTOM (or not), and explicitly provide an appropriate subtype of Content in the DataObject instances passed to a DFS operation. The following example, from a plain JAX-WS client, passes a DataObject containing content stored in an existing file to the Object service create method as BinaryContent.

Example 10-1. Uploading Base64 content using plain JAX-WS

```
public ObjectIdentity createContent (String filePath, String format)
    throws IOException, SerializableException
{
    File testFile = new File(filePath);
```

```
    byte[] byteArray = new byte[(int) testFile.length()];
    FileInputStream fileInputStream = new FileInputStream(testFile);
    ObjectIdentity objIdentity = new ObjectIdentity();
    objIdentity.setRepositoryName(
        ((RepositoryIdentity)
            (m_serviceContext.getIdentities().get(0))).
                                getRepositoryName());
    DataObject dataObject = new DataObject();
    dataObject.setIdentity(objIdentity);
    dataObject.setType("dm_document");
    PropertySet properties = new PropertySet();
    dataObject.setProperties(properties);
    StringProperty objNameProperty = new StringProperty();
    objNameProperty.setName("object_name");
    objNameProperty.setValue("MyImage-" + System.currentTimeMillis());
    properties.getProperties().add(objNameProperty);

    // the following represents typical usage
    // it is also ok to use MTOM to transfer a BinaryContent representation
    // and BASE_64 to transfer DataHandlerContent
    if (m_transferMode == ContentTransferMode.MTOM)
    {
        // calls helper method shown below
        dataObject.getContents().add(getDataHandlerContent(byteArray,
                                                           format));
    }
    else if (m_transferMode == ContentTransferMode.BASE_64)
    {
        // calls helper method shown below
        dataObject.getContents().add(getBinaryContent(byteArray,
                                                      format));
    }
    DataPackage dataPackage = new DataPackage();
    dataPackage.getDataObjects().add(dataObject);

    System.out.println("Invoking the create operation on the
                                                 Object Service.");
    dataPackage = m_servicePort.create(dataPackage, null);
    return dataPackage.getDataObjects().get(0).getIdentity();
}

private DataHandlerContent getDataHandlerContent (byte[] byteArray,
                                                  String format)
{
    DataSource byteDataSource = new ByteArrayDataSource(byteArray,
                                                        "gif");
    DataHandler dataHandler = new DataHandler(byteDataSource);
    DataHandlerContent dataHandlerContent = new DataHandlerContent();
    dataHandlerContent.setFormat(format);
    dataHandlerContent.setValue(dataHandler);
    return dataHandlerContent;
}

private BinaryContent getBinaryContent (byte[] byteArray,
                                        String format)
{
    BinaryContent binaryContent = new BinaryContent();
    binaryContent.setFormat(format);
    binaryContent.setValue(byteArray);
    return binaryContent;
}
```

The transfer mode used to send the content over the wire is determined by the client framework—in the case of this example by whether MTOM is enabled on the JAX-WS

ServicePort. The following snippet shows one means of enabling MTOM by passing an instance of javax.xml.ws.soap.MTOMFeature when getting the service port from the service.

```
String objectServiceURL = contextRoot + "/core/ObjectService";
ObjectService objectService = new ObjectService(
            new URL(objectServiceURL),
            new QName("http://core.services.fs.documentum.emc.com/",
                                                    "ObjectService"));
servicePort = objectService.getObjectServicePort
                            (new MTOMFeature());
```

If you are using the productivity layer, the productivity layer runtime checks the ContentTransferMode setting and takes care of converting the content type to an appropriate subtype before invoking the remote service. The transfer mode used for the upload is determined by the runtime, also based on the ContentTransferMode setting.

Example 10-2. Uploading content using the Java PL

```
public DataPackage createWithContentDefaultContext(String filePath)
    throws ServiceException
{
     File testFile = new File(filePath);

    if (!testFile.exists())
    {
        throw new IOException("Test file: " + testFile.toString() +
                                              " does not exist");
    }

    ObjectIdentity objIdentity = new ObjectIdentity
                                  (defaultRepositoryName);
    DataObject dataObject = new DataObject(objIdentity, "dm_document");
    PropertySet properties = dataObject.getProperties();
    properties.set("object_name", "MyImage");
    properties.set("title", "MyImage");
    properties.set("a_content_type", "gif");
    dataObject.getContents().add(new FileContent(testFile.
                                  getAbsolutePath(),"gif"));

    OperationOptions operationOptions = null;
    return objectService.create(new DataPackage(dataObject),
                                        operationOptions);
}
```

Downloading content using Base64 and MTOM

When using a service to download content remotely, it is important to correctly configure two profiles, which can be set in the service context or passed in the OperationOptions argument to the service method. By default (to avoid unwanted and expensive content transfer) no content is included in objects returned by DFS. To make sure content is returned in the HTTP response, you need to explicitly set the formatFilter property in a ContentProfile. The following snippet shows typical profile settings:

```
ContentTransferProfile contentTransferProfile =
                       new ContentTransferProfile();
contentTransferProfile.setTransferMode(ContentTransferMode.MTOM);

ContentProfile contentProfile = new ContentProfile();
```

```
contentProfile.setFormatFilter(FormatFilter.ANY);
```

For more information see ContentProfile, page 97 and ContentTransferProfile, page 99.

The following is an example of a WSDL-based client method (JAX-WS) that shows content download using the object service get operation. This method examines the type of the Content returned by the operation, extracts the content value as a byte array and writes it to a file.

Example 10-3. Downloading content with plain JAX-WS

```
public File getContentAsFile (ObjectIdentity objectIdentity,
            File targetFile)
 throws IOException, SerializableException
{
    ObjectIdentitySet objectIdentitySet = new ObjectIdentitySet();
    objectIdentitySet.getIdentities().add(objectIdentity);

    ContentTransferProfile contentTransferProfile =
                           new ContentTransferProfile();
    contentTransferProfile.setTransferMode(m_transferMode);

    ContentProfile contentProfile = new ContentProfile();
    contentProfile.setFormatFilter(FormatFilter.ANY);

    OperationOptions operationOptions = new OperationOptions();
    operationOptions.getProfiles().add(contentTransferProfile);
    operationOptions.getProfiles().add(contentProfile);

    DataPackage dp = m_servicePort.get(objectIdentitySet,
                     operationOptions);

    Content content = dp.getDataObjects().get(0).getContents().get(0);
    OutputStream os = new FileOutputStream(targetFile);
    if (content instanceof UrlContent)
    {
        //Handle URL content -- see following section
    }
    else if (content instanceof BinaryContent)
    {
        BinaryContent binaryContent = (BinaryContent) content;
        os.write(binaryContent.getValue());
    }
    else if (content instanceof DataHandlerContent)
    {
        DataHandlerContent dataHandlerContent = (DataHandlerContent) content;
        InputStream inputStream = dataHandlerContent.getValue().getInputStream();
        if (inputStream != null)
        {
            int byteRead;
            while ((byteRead = inputStream.read()) != -1)
            {
                os.write(byteRead);
            }
            inputStream.close();
        }
    }
    os.close();
    return targetFile;
}
```

The following productivity layer example does something similar; however it can use the Content#getAsFile convenience method to get the file without knowing the concrete type of the Content object.

Example 10-4. Downloading content using the productivity layer

```
public File getContentAsFile (ObjectIdentity objectIdentity,
                              String geoLoc,
                              ContentTransferMode transferMode)
    throws ServiceException
{
    ContentTransferProfile transferProfile = new ContentTransferProfile();
    transferProfile.setGeolocation(geoLoc);
    transferProfile.setTransferMode(transferMode);
    serviceContext.setProfile(transferProfile);

    ContentProfile contentProfile = new ContentProfile();
    contentProfile.setFormatFilter(FormatFilter.ANY);
    OperationOptions operationOptions = new OperationOptions();
    operationOptions.setContentProfile(contentProfile);
    operationOptions.setProfile(contentProfile);

    ObjectIdentitySet objectIdSet = new ObjectIdentitySet();
    List<ObjectIdentity> objIdList = objectIdSet.getIdentities();
    objIdList.add(objectIdentity);

    DataPackage dataPackage = objectService.get(objectIdSet,
                              operationOptions);
    DataObject dataObject = dataPackage.getDataObjects().get(0);
    Content resultContent = dataObject.getContents().get(0);
    if (resultContent.canGetAsFile())
    {
        return resultContent.getAsFile();
    }
    else
    {
        return null;
    }
}
```

Downloading UrlContent

UrlContent objects contain a string representing an ACS (Accelerated Content Services) URL. These URLs can be used to retrieve content using HTTP GET, with the caveat that they are set to expire, so they can't be stored long term. In a distributed environment with configured network locations, ACS content transfer enables transfer of content from the nearest network location based on the geoLocation setting in the ContentTransferProfile.

A client can get UrlContent explicitly using the Object service getContentUrls operation. UrlContent can also be returned by any operation that returns content if an ACS server is configured and active on the Content Server where the content is being requested, and if the requested content is available via ACS. Clients that do not use the productivity layer should detect the type of the content returned by an operation and handle it appropriately. In addition, the ACS URL must be resolvable when downloading the UrlContent.

Note: The expiration time for an ACS URL can be configured by setting the default.validation.delta property in acs.properties. The default value is 6 hours. For more information see the *EMC Documentum Content Server Distributed Configuration Guide.*

A client that does not use the productivity layer needs to handle UrlContent that results from a get operation or a getContentUrls operation by explicitly downloading it from ACS. The following

JAX-WS sample extracts the UrlContent from the results a get operation, then passes the URL and a FileOutputStream to a second sample method, which downloads the ACS content to a byte array which it streams to the FileOutputStream.

Example 10-5. Downloading UrlContent using plain JAX-WS

```
public File getContentAsFile (ObjectIdentity objectIdentity,
                              File targetFile)
        throws IOException, SerializableException
{
    ObjectIdentitySet objectIdentitySet = new ObjectIdentitySet();
    objectIdentitySet.getIdentities().add(objectIdentity);

    ContentTransferProfile contentTransferProfile =
                           new ContentTransferProfile();
    contentTransferProfile.setTransferMode(m_transferMode);

    ContentProfile contentProfile = new ContentProfile();
    contentProfile.setFormatFilter(FormatFilter.ANY);

    OperationOptions operationOptions = new OperationOptions();
    operationOptions.getProfiles().add(contentTransferProfile);
    operationOptions.getProfiles().add(contentProfile);

    DataPackage dp = m_servicePort.get(objectIdentitySet,
                     operationOptions);

    Content content = dp.getDataObjects().get(0).getContents().get(0);
    OutputStream os = new FileOutputStream(targetFile);
    if (content instanceof UrlContent)
    {
        UrlContent urlContent = (UrlContent) content;
        // call private method shown below
        downloadContent(urlContent.getUrl(), os);
    }
    else if (content instanceof BinaryContent)
    {
        //handle binary content -- see preceding section
    }
    else if (content instanceof DataHandlerContent)
    {
       //handle DataHandlerContent -- see preceding section
    }
    os.close();
    return targetFile;
}
```

The following sample method does the work of reading the content from ACS to a buffer and streaming it to an OutputStream.

```
private void downloadContent (String url, OutputStream os)
    throws IOException
{
    InputStream inputStream;
    inputStream = new BufferedInputStream(new URL(url).openConnection().
                  getInputStream());

    int bytesRead;
    byte[] buffer = new byte[16384];
    while ((bytesRead = inputStream.read(buffer)) > 0)
    {
        os.write(buffer, 0, bytesRead);
    }
```

```
}
```

If on the other hand you are using the productivity layer, the PL runtime does most of the work behind the scenes. When retrieving content using a get operation, you can call getAsFile on the resulting content object without knowing its concrete type. If the type is UrlContent, the runtime will retrieve the content from ACS and write the result to a file.

The following example gets UrlContent explicitly using the Object service getContentUrls function and writes the results to a file.

Example 10-6. Getting UrlContent using the productivity layer

```
public void getObjectWithUrl (ObjectIdentity objIdentity)
    throws ServiceException, IOException
{
    objIdentity.setRepositoryName(defaultRepositoryName);
    ObjectIdentitySet objectIdSet = new ObjectIdentitySet();
    List<ObjectIdentity> objIdList = objectIdSet.getIdentities();
    objIdList.add(objIdentity);
    List urlList = objectService.getObjectContentUrls(objectIdSet);
    ObjectContentSet objectContentSet = (ObjectContentSet)
                                        urlList.get(0);
    Content content = objectContentSet.getContents().get(0);
    if (content.canGetAsFile())
    {
        // downloads the file using the ACS URL
        File file = content.getAsFile();
        System.out.println("File exists: " + file.exists());
        System.out.println(file.getCanonicalPath());
    }
    else
    {
        throw new IOException("Unable to get object " + objIdentity +
                              " as file.");
    }
}
```

Chapter 11

Content Transfer with Unified Client Facilities

Unified Client Facilities (UCF) is an EMC technology that orchestrates direct transfer of content between a client computer and a Documentum repository. UCF is fully integrated with DFS, and can be employed as the content transfer mechanism in many types of DFS consumer application. The DFS SDK provides client libraries to support UCF content transfer in Java and in .NET. The Java and .NET libraries are integrated into the DFS productivity layer runtime to simplify usage by productivity layer applications. Applications that do not use the productivity layer can use the UCF client libraries directly in their applications outside of the DFS productivity layer runtime. Web applications can package the UCF client libraries into an applet or an ActiveX object to enable UCF content transfer between a browser and a Content Server. Clients that use the .NET libraries *do not* need to have a Java Runtime Engine installed on their system.

This chapter discusses the use of UCF for content transfer in a DFS context, and includes the following sections:

Overview of Unified Client Facilities

UCF is a proprietary remote content transfer technology. UCF client is intended for a single user, either using a browser in a web application, or a using a thick client. Use of UCF is not supported on the middle tier of a distributed application. (Typically the middle tier would be a web application functioning as a DFS consumer.)

You may want to consider a list of its potential benefits when deciding whether and when to use it rather than the alternative content transfer modes (MTOM and Base64). Unified Client Facilities:

- Can be deployed through an applet or ActiveX object in web applications, which enables direct content transfer between the machine where the browser is located and a Content Server.
- Provides support for distributed content by providing back-end integration with ACS (Accelerated Content Services) and BOCS (Branch Office Caching Services). Both these technologies provide

performance optimizations when content is distributed across different repositories in distant network locations.

- Provides support for transferring documents consisting of multiple files, such as XML documents with file references or Microsoft documents with embedded objects, and for creating and updating Documentum virtual documents.
- Maintains a registry of documents on the client and downloads from the repository only content that has changed.
- Provides facilities for opening downloaded content in an editor or viewer.

However, UCF content transfer mode may also be required to work around memory limitations for .NET clients (see Memory limitations associated with MTOM content transfer mode, page 161.

Note: While performing Documentum Foundation Services outbound content transfer operations, the files may be marked as Read-Only on the client file system. Documentum Foundation Services marks a file as Read-Only in the client file system under the following conditions:

- When an object is viewed using the View operation.
- When an object is locked by a user during Checkout operation.
- When the user does not have Version permission on the object during the Checkout operation.

System requirements

DFS provides both Java and .NET UCF integrations. For the Java integration, JRE 1.5.0_12 or later and DFS 6.0SP1 or later are required. For the native .NET integration, .NET Framework 3.5 SP 1 or later and DFS 6.7 or later are required. DFS .NET productivity layers prior to 6.7 are rely on the Java UCF integration and require a JRE to be installed in the client environment.

In DFS 6.7 or later, native .NET UCF client-side components are supported, either as an ActiveX object (for web applications) or as a .NET assembly (for thick clients), and no JRE is required on the client machine. Native DFS UCF .NET integration on the client side will require DFS services version 6.7 or later on the server side.

UCF component packaging

UCF includes both server-side and client-side components. The UCF server components are packaged with the DFS web services as an EAR or WAR file. The client-side components differ, depending on your application development scenario.

If you are developing a thick client that uses the productivity layer, the components are packaged in the DFS client runtime libraries delivered in the SDK. You can set up a project using the usual DFS client dependencies, as described in Configuring .NET consumer project dependencies, page 67 and Configuring Java dependencies for DFS productivity-layer consumers, page 45. No other dependencies are required.

If you are not using the productivity layer, and you are developing a thick client, you will need to reference the UCF client-side libraries in your project, which enables your application to invoke the UcfConnection class. In a .NET project you will need to add a reference to

Emc.Documentum.Fs.Runtime.Ucf.dll. In Java, you should place ucf-connection.jar on your project classpath.

Finally, if you are developing a web application, and need to download the UCF client components to the browser, you will need to develop an applet or an ActiveX object for this purpose. A sample applet and a sample Activex are included in the DFS 6.7 SDK. You will need to package the DFS client runtime dependencies in the applet or ActiveX object. For details see Write the applet code for deploying and launching UCF, page 188, Build and bundle the applet, page 189 (for Java applications), and Tutorial: Using UCF .NET in a .NET client, page 193

Deploying in distributed environments

UCF clients require direct access to UCF server for content transfer. There are deployment models in which the backend DFS server is not visible to the end client, as in the case of a DFS web service-based web application. In this deployment the browser has access to the web application itself, but not to the DFS backend. In this case, it is recommended to use a reverse proxy to forward UCF requests to the backend DFS server. Below is an Apache httpd.conf for this scenario:

```
# ProxyPass
# enables Apache as forwarding proxy

ProxyPass /services/ http://localhost:8888/services/
ProxyPass / http://ui-server:8080/

LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_http_module modules/mod_proxy_http.so
```

UCF is a stateful protocol relying on HTTP sessions to preserve state. DFS requires *more than one trip to the server side* to establish a UCF connection. For this reason, it is required to use sticky session based load balancing so that all requests that are part of the same HTTP session are routed to the same backend node.

Note:

- JSESSIONID is preserved by DFS for session management such as load balancing. The client should not use JSESSIONID cookie for other usages.
- Ensure you do not pass JSESSIONID cookie to UCF server unless this cookie is returned by a previous UCF connection.

The following example demonstrates how to use same HTTP session for different UCF connections

```
// Create an initial UCF connection with cookies, passedInCookies must
not contain JSESSIONID

UcfConnection connection1 = new UcfConnection(new URL(url),
passedInCookies, targetDeploymentId);

// Create a second UCF connection, merge cookies from previous UCF
connection that contains JSESSIONID
// the second UCF connection is routed to the same backend node
through the JSESSIONID

String newCookies = connection1.getCookies() + "my custom cookies";
UcfConnection connection2 = new UcfConnection(new URL(url), newCookies,
targetDeploymentId);
```

UCF failover is not supported as a result. In case of a node failure the whole UCF transfer process, including establishing a new UCF connection must be restarted.

Once an HTTP session is established, DFS will reuse it for the same service instance to avoid any load balancing issues. (More accurately, DFS will reuse the same HTTP session for the same service, provided that the client application does not update the ActivityInfo instance in the ContentTransferProfile instance.) DFS will throw the following exception if the consumer tries to override the existing HTTP JSESSIONID value with a different one:

```
Can not execute DFS call: provided HTTP session id "xxx" overrides
original HTTP session id "xxx". Please use original HTTP session
for DFS calls.
Doing otherwise will cause the call to fail in load-balanced environments.
```

DFS classes specific to UCF

The DFS data model classes related to DFS are:

- ContentTransferProfile
- ContentProfile
- ContentTransferMode
- ActivityInfo
- UcfContent

The *ContentTransferProfile* class provides a means of setting the ContentTransferMode to UCF, which enables UCF processing in the DFS runtime. ContentTransferProfile encapsulates an ActivityInfo instance, which is used to tune the DFS-orchestrated transfer, or to enable client-orchestrated UCF (see also Client-orchestrated UCF, page 177).ContentTransferProfile includes other settings as well related UCF, which determine whether asynchronous or cached content transfer are allowed, and whether Microsoft Office links are processed on the client.

The *ContentProfile* class allows setting a UCF post-transfer action ("dfs:view" or "dfs:edit") to be executed by the UCF client after the transfer is complete.

The *ActivityInfo* class permits a developer to control the UCF connection lifecycle and to provide details for externally initialized UCF connections. Controlling the UCF connection lifecycle by setting the autoCloseConnection flag to false enables an application to reuse UCF connections across service requests. The final service request must set the autoCloseConnection flag true to release associated UCF resources. For an example, see Optimization: controlling UCF connection closure, page 179.

Note: The ActivityInfo passed by the client might be updated by DFS runtime; You will not be able to retrieve the cookies set in the ActivityInfo.

The *UcfContent* class is used explicitly to indicate that no further runtime UCF processing is required on the Content instance. If the files to be transferred are not located on the same machine as the DFS consumer, as it would be in case of a browser integration, the application developer should explicitly provide a UcfContent instance in the DataObject passed to the service operation.

DFS-orchestrated UCF

DFS-orchestrated UCF is the model in which the DFS runtime takes full responsibility of initiating the UCF connection and performing the content transfer between the client and the server. Developers can use any type of DFS Content with this model, except for UcfContent as it implies that the DFS runtime does not participate in the content transfer process. To enable DFS-orchestrated UCF, it is enough to set the ContentTransferMode to UCF in the ServiceContext, as shown in the Java listing below:

```
IServiceContext context = ContextFactory.getInstance().newContext();
ContentTransferProfile profile = new ContentTransferProfile();
profile.setTransferMode(ContentTransferMode.UCF);
context.setProfile(profile);
```

A typical use of DFS-orchestrated UCF would be a thick client invoking the DFS remote web services API.

Client-orchestrated UCF

Client-orchestrated UCF is the model in which the process of establishing a UCF connection is delegated to the client rather than the DFS runtime.

The process of establishing a UCF connection consists of a set of steps that must be taken in a specific order for the procedure to succeed. First of all, a UCF installer must be downloaded from the server side. It will check whether a UCF client is already present in the environment and if it is, whether it needs to be upgraded or not. Before the UCF installer can be executed, it is necessary to confirm its author and whether its integrity has been compromised or not. This is achieved through digitally signing the installer and verifying the file signature on the client side. The downloaded UCF installer is executed only if it considered trusted. Once running, it will install and launch the UCF client and, eventually, request a UCF connection ID from the UCF server once the process is successful.

To encapsulate this complexity, DFS provides the UcfConnection class. This class takes the URL of the UCF server as a constructor argument and allows the developer to obtain a UCF connection ID through a public method call. The provided URL should point to the location of the UCF installer and ucf.installer.config.xml on the remote server. This class is available for both Java and .NET consumers. Both Java and .NET DFS Productivity Layers rely on UcfConnection to establish UCF connections.

A typical use of client-orchestrated UCF would be a browser client, a web application, in which UCF is downloaded from the server, and the client code invokes the UcfConnection class to establish theUCF connection ID.

Browser-based UCF integration

The advantage of browser integration is that the content can be transferred from the client machine directly to the content server, without being stored on any intermediary tier. For this to happen, the browser has to establish the UCF connection from the client machine to the DFS server. Once this is done, the UCF connection details along with the path of the file(s) to be transferred must be provided to the web application which initiates the transfer as a client orchestrated UCF.

In a Java environment, UCF integration is accomplished using an applet. The applet will need "ucf-connection.jar", which is part of the DFS SDK in the classpath. The following code snipped can be used in the applet to establish a UCF connection:

```
UcfConnection c = new UcfConnection(new URL
                  (getParameter("ucf-server")));
uid = c.getUid();
jsessionId = c.getJsessionId();
```

where ucf-server has is a string representing the DFS service context, such as "http://host:port/context-root/module", for example "http://localhost:8888/services/core". The values of "uid" and "jsessionId" must be passed on to the browser and eventually, to the web application initiating the UCF content transfer. One way of passing on these values to the browser is through the JSObject plugin, which allows Java to manipulate objects that are defined in JavaScript.

In a .NET environment the UCF integration can be accomplished using an ActiveX object, which will have to reference and package "Emc.Documentum.FS.Runtime.Ucf.dll", which is part of the SDK. The following method can be defined in the C# based ActiveX component to establish a UCF connection and return its ID:

```
[ComVisible(true)]
public string GetUid(String jsessionId, String url)
{
    UcfConnection c = new UcfConnection(new Uri(url), jsessionId, null);
    return c.GetUcfId();
}
```

As with the Java integration, the UCF connection ID (uid) and "jsessionId" must be passed to the web application initiating the UCF content transfer.

Server-side processing using the productivity layer

The client application can orchestrate UCF content transfer using the established UCF connection. This can be done in a client that uses DFS in local mode, or in a client that uses the DFS web services. In either case, to enable this type of integration the application developer has to provide the HTTP and UCF session IDs to the DFS runtime through an ActivityInfo instance:

```
ContentTransferProfile profile = new ContentTransferProfile();
profile.setTransferMode(ContentTransferMode.UCF);
profile.setActivityInfo(new ActivityInfo(jsessionId, null,
                        ucfId, null));
serviceContext.setProfile(profile);
```

Server-side processing without the productivity layer

Applications that do not use the productivity layer must, in addition to setting the transfer mode and activity info on the service context, provide explicit UcfContent instances in the DataObject:

```
UcfContent content = new UcfContent();
content.setLocalFilePath("path-fo-file-on-the-client-machine");
DataObject object = new DataObject();
object.getContents().add(content);
```

Authentication

UCF does not have any built-in authentication mechanisms. It is controlled from the server side by DFC, which begins the content transfer only after authenticating the user. This leaves the door open for Denial of Service attacks as clients can establish as many UCF connections as they wish.

HTTP-proxy-based SSO solutions like WebSEAL, ClearTrust and SiteMinder address this concern by allowing only authenticated HTTP requests into the protected web object space. Thus, if a UCF server is part of the protected object space, only users authenticated by the SSO proxy would be able to establish a UCF connection.

To establish a secure UCF connection, you must add the SSO cookie to the UcfConnection constructor.

```
UcfConnection connection = UcfConnection(ucfServerUrl, cookieHeader,
targetDeploymentId);
```

Tips and tricks

UCF limitations pertaining to 64-bit JVM

Unified Client Facilities (UCF) requires a 32-bit JVM and is incompatible with a 64-bit JVM. A Java client that uses UCF content transfer mode must run in a 32-bit JVM.

Alternative methods of supplying ActivityInfo and their relative precedence

A client that constructs its own ActivityInfo instance can supply it to the service by directly adding it to a ContentTransferProfile, or by adding it to an instance of UcfContent. The ContentTransferProfile is generally added to the service context, but may also be passed with an OperationOptions instance.

In all cases, if the client-supplied ActivityInfo has properly initialized activityInfo and sessionId settings, and if its closed flag is set to false, and if the ContentTransferMode is set to UCF, the DFS framework will use the client-supplied settings and will not launch the UCF session on the client. (It will assume that the client has taken responsibility for this.)

In the case that an ActivityInfo is supplied in both the ContentTransferProfile and the UcfContent, the framework will use the ActivityInfo that is stored in the ContentTransferProfile.

Optimization: controlling UCF connection closure

The default behavior of the DFS framework is to close an active UCF connection (from the server side) after it has been used by a service operation and to terminate the client UCF process. In some applications this can incur unnecessary overhead. This behavior can be overridden using the ActivityInfo.autoCloseConnection flag. The consumer can set up the ActivityInfo and supply it to the

service using either method described in Alternative methods of supplying ActivityInfo and their relative precedence, page 179. The ActivityInfo should have the following settings:

ActivityInfo field	Supplied value
autoCloseConnection	false
closed	false
activityId	null
sessionId	null
initiatorSessionId	null

The client runtime provides a constructor that permits the consumer to set autoCloseConnection only, and the remaining settings are provided by default. With these settings, the DFS framework will supply standard values for activityId and sessionId, so that content will be transferred between the standard endpoints: the UCF server on the DFS host, and the UCF client on the DFS consumer. The following snippet shows how to set the autoCloseConnection using the Java productivity layer:

```
IServiceContext c = ContextFactory.getInstance().newContext();
c.addIdentity(new RepositoryIdentity("…", "…", "…", ""));
ContentTransferProfile p = new ContentTransferProfile();
p.setTransferMode(ContentTransferMode.UCF);
p.setActivityInfo(new ActivityInfo(false));
c.setProfile(p);
IObjectService s = ServiceFactory.getInstance()
    .getRemoteService(IObjectService.class,
                      c,
                      "core",
                      "http://localhost:8888/services");
DataPackage result = s.get(new ObjectIdentitySet
                          (new ObjectIdentity
                          (new ObjectPath("/Administrator"), "…")),
                                        null);
```

If the consumer sets autoCloseConnection to false, the consumer is responsible for closing the connection. This can be accomplished by setting autoCloseConnection to true before the consumer application's final content transfer using that connection. If the consumer fails to do this, the UCF connection will be left open, and the UCF client process will not be terminated.

This optimization removes the overhead of launching the UCF client multiple times. It is only effective in applications that will perform multiple content transfer operations between the same endpoints. If possible, this overhead can be more effectively avoided by packaging multiple objects with content in the DataPackage passed to the operation.

Note: If high performance content transfer is required for UCF.NET, you must initialize the autoCloseConnection property of ActivityInfo class to FALSE. This setting is not applicable for Java UCF.

Re-use cached ActivityInfo to avoid creating new UCF connections

When using client-orchestrated UCF (in which you populate an ActivityInfo, which is passed to the service or set in a registered service context), to avoid creating new UCF connections on the server, you must cache the ActivityInfo on the client and pass the same instance in all service operation calls.

For example, suppose that you want to optimize a batch import by controlling the AutoCloseConnection flag. In this case you will want to set AutoCloseConnection to false for all transfers except the last, before which you must set AutoCloseConnection to true to close the connection. The following Java sample caches an ActivityInfo and passes it to a private batchimport method, which uses the ActivityInfo in its invocation of the Object service create method.

```
ContextFactory theContextFactory = ContextFactory.getInstance();
IServiceContext theContext = theContextFactory.newContext();
RepositoryIdentity theRepositoryIdent = new RepositoryIdentity();
theRepositoryIdent.setRepositoryName( theRepository );
theRepositoryIdent.setUserName( theUser );
theRepositoryIdent.setPassword(thePassword );
theContext.addIdentity( theRepositoryIdent );

theContext = theContextFactory.register(theContext, "core", theUrl );

theContext.setProfiles( new ArrayList<Profile>() );

ActivityInfo theInfo = new ActivityInfo(false);
for ( int i = 1; i < 3; i++ )
{
    batchImport(theContext, theInfo);
}
theInfo.setAutoCloseConnection(true);
batchImport(theContext, theInfo);
```

The batchimport method does the work of setting up the context and DataObject and invoking the create operation:

```
private static void batchImport(IServiceContext theContext,
                                ActivityInfo theInfo)
{
   try
   {
       OperationOptions theOptions = new OperationOptions();

       ContentTransferProfile theTransferProfile =
                              new ContentTransferProfile();
       theTransferProfile.setTransferMode(ContentTransferMode.UCF);
       theTransferProfile.setActivityInfo( theInfo );
       theOptions.setContentTransferProfile( theTransferProfile );
       ContentProfile theContentProfile = new ContentProfile();
       theContentProfile.setFormatFilter( FormatFilter.NONE );
       theOptions.setContentProfile( theContentProfile );

       theOptions.setPropertyProfile( getPropertyProfile() );

       ServiceFactory theServiceFactory = ServiceFactory.
                                          getInstance();
       IObjectService theObjectService =
               theServiceFactory.getRemoteService(IObjectService.class,
                                                  theContext,
                                                  "core",
                                                  theUrl );
```

```
        DataPackage theDataPackage = new DataPackage();
        for ( int i = 1; i < 2; i++ )
        {
            DataObject theDataObject = new DataObject
                        (new ObjectIdentity(theRepository),"dm_document");
            PropertySet thePropertySet = new PropertySet();
            thePropertySet.set("object_name", "test_" + i);
            theDataObject.setProperties( thePropertySet );
            FileContent theFileContent =
                    new FileContent( "C:\\\\AIS_Test\\\\MyAttachment4.doc",
                                                          "msw8" );
            theFileContent.setRenditionType( RenditionType.PRIMARY );
            theDataObject.getContents().add( theFileContent );
            theDataPackage.addDataObject( theDataObject );

        }
        theObjectService.create(theDataPackage, theOptions);
        System.out.println("File imported");
    }
    catch ( Exception e )
    {
        System.out.println( e.getMessage());
    }
    return;
}
```

If instead you create a new ActivityInfo for each operation, the server will start fresh UCF connections and the optimization will not work.

Opening a transferred document in a viewer/editor

You can specify an action to perform on the client after an operation that transfers content using the setPostTransferAction method of ContentProfile. This feature is available only if the content is transferred using the UCF transfer mode. The setPostTransferAction method takes a String argument, which can have any of the values described in Table 23, page 182.

Table 23. PostTransferAction strings

Value	Description
Null or empty string	Take no action.
dfs:view	Open the file in view mode using the application associated with the file type by the Windows operating system.
dfs:edit	Open the file in edit mode using the application associated with the file type by the Windows operating system.
dfs:edit?app=_EXE_	Open the file for editing in a specified application. To specify the application replace _EXE_ with a fully-qualified path to the application executable; or with just the name of the executable. In the latter case the operating system will need to be able to find the executable; for example, in Windows, the executable must be found on the %PATH% environment variable. Additional parameters can be passed to the application preceded by an ampersand (&).

Resolving ACS URL for UcfContent

By default, UCF relies on ACS to transfer content. On the first request if the ACS URL cannot be resolved by the client machine, the UCF content transfer fails. Further requests will bypass ACS and rely on application server for content transfer. To facilitate UCF content transfer with ACS and BOCS, the ACS URL must be resolvable.

Choosing a Home directory for UcfInstaller

UCF needs UcfLaunchClickOnce.exe to start the UCF client engine on the client machine. The DFS .NET Productivity Layer chooses to download this file in the directories defined by one of the following environment variables and in the following order:

1. %UCF_LAUNCH_CLICK_ONCE_PATH%
2. %USERPROFILE%
3. %HOMEDRIVE%%HOMEPATH%
4. %WINDIR%

If none of the above variables are valid, a UCF exception occurs. You must then set the `%UCF_LAUNCH_CLICK_ONCE_PATH%` variable with a folder path in non-network drive with WRITE permission.

For more information on ClickOnce refer: http://msdn.microsoft.com/en-us/library/t71a733d%28v=vs.80%29.aspx

Alternating to UCF Java from .NET Productivity Layer

By default, the DFS 6.7 .NET Productivity Layer uses UCF .NET to transfer content when UCF content transfer mode is set.

However, you can switch over to UCF Java after you configure the following:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
<appSettings>
<add key="use.native.ucf" value="false"/>
</appSettings>
</configuration>
```

Tutorial: Using UCF in a Java client

The following sections provide a tutorial on a sample Java web application that uses UCF content transfer.

Requirements

UCF is dependent on the availability of a Java 5 or 6 JRE on the client machine to which the UCF jar files are downloaded. It determines the Java location using the JAVA_HOME environment variable.

UCF in a remote DFS Java web application

This section provides instructions for creating a test application that enables UCF transfer, using the topology described under Browser-based UCF integration, page 177.

The test application environment must include the following:

- an end-user machine running a browser, with an available Java Runtime Environment (JRE)
- an Apache application server used as a reverse proxy
- an application server that hosts a web application that includes a minimal user interface and a DFS consumer application
- an application server hosting the DFS web services
- a Content Server and repository

For our tests of this scenario, we deployed both the web application and DFS on Tomcat 6e. The test application shown here also requires the Java Plug-in, documented here: http://java.sun.com/j2se/1.4.2/docs/guide/plugin/developer_guide/contents.html The Java Plug-in is part of the Java Runtime Environment (JRE), which is required on the end-user machine.

The sample application is designed to run as follows:

1. The browser sends a request to a JSP page, which downloads an applet. If the browser is configured to check for RSA certificates, the end user will need to import the RSA certificate before the applet will run. (The signing of the applet with the RSA certificate is discussed in Sign the applet, page 189.)
2. The applet instantiates a UCF connection, gets back a jsessionId and a uid, then sends these back to the JSP page by calling a JavaScript function.
3. In the web application, a servlet uses the jsessionId, uid, and a filename provided by the user to create an ActivityInfo object, which is placed in a ContentTransferProfile in a service context. This enables DFS to perform content transfer using the UCF connection established between the UCF server on the DFS service host and the UCF client on the end-user machine.

The tasks required to build this test application are described in the following sections:

1. Set up the development environment, page 185.
2. Configure the Apache reverse proxy, page 185
3. Code an HTML user interface for serving the applet, page 186
4. Write the applet code for deploying and launching UCF, page 188
5. Build and bundle the applet, page 189
6. Sign the applet, page 189

7. Create a servlet for orchestrating the UCF content transfer, page 190

Set up the development environment

The environment required for the test consists of the following:

- An end-user machine, which includes a browser, and which must have a Java Runtime Environment available in which to run UCF (and the Java Plug-in). The browser should be configured to use a Java 5 JRE for applets (we tested this sample application using JRE 1.5.0_10).
- A proxy set up using the Apache application server (we tested using version 2.2).
- An application server hosting the web application components, including the DFS consumer.
- An application server hosting the DFS services and runtime (which include the required UCF server components). This can be a freestanding DFS installation, or DFS running on a Content Server. The DFS installation must have its dfc.properties configured to point to a connection broker through which the Content Server installation can be accessed.
- A Content Server installation.

To create a test application, each of these hosts must be on a separate port. They do not necessarily have to be on separate physical machines. For purposes of this sample documentation, we assume the following:

- The proxy is at http://localhost:80.
- The web application is at http://localhost:8080.
- The DFS services (and the UCF components, which are included in the DFS ear file) are a freestanding installation at http://localhost:8888/services/core

Configure the Apache reverse proxy

The Apache reverse proxy can be configured by including the following elements in the httpd.conf file:

```
P# ProxyPass
# enables Apache as forwarding proxy

ProxyPass /services/ http://localhost:8888/services/
ProxyPass / http://ui-server:8080/

LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_http_module modules/mod_proxy_http.so
```

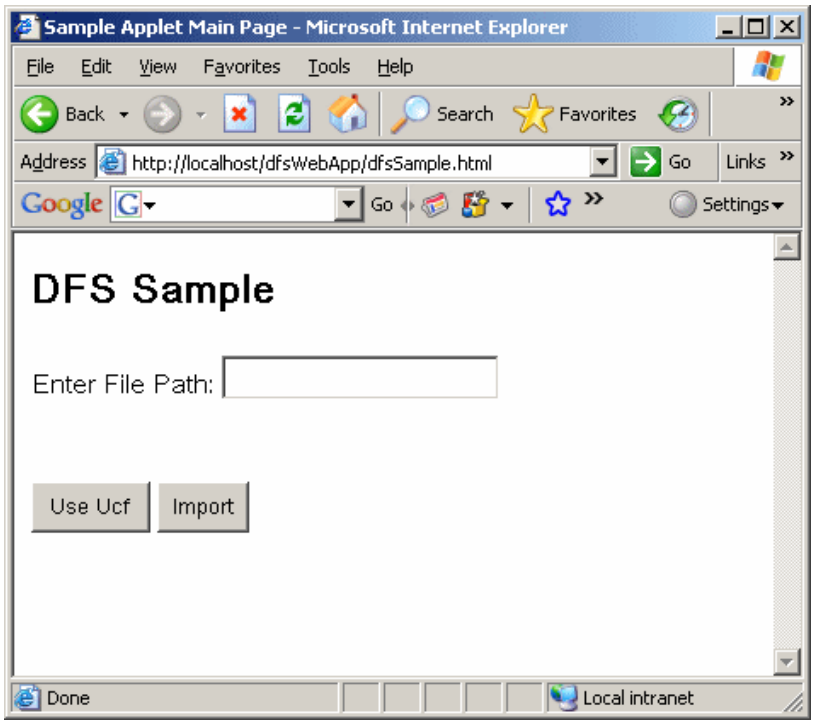
For example, a proxy running on proxy:80 forwards requests as follows:

- http://proxy:80/services/core/runtime/AgentService.rest to http://dfs-server:8888/services/core/runtime/AgentService.rest.
- The default mapping is to the application server that hosts UI and DFS consumer, so it forwards http://proxy:80/dfsWebApp/DfsServiceServlet to http://ui-server:8080/dfsWebApp/DfsServiceServlet.

Code an HTML user interface for serving the applet

The sample HTML presents the user with two buttons and a text box. When the user clicks the **Use Ucf** button, a second popup is launched while the UCF connection is established by the applet. When the applet finishes, the second windows closes and the user can import a file specified by a file path entered in the text box.

Figure 20. User interface for UCF test application



Note: This sample has been implemented with two buttons for demonstration purposes. A button with the sole function of creating the UCF connection would probably not be a useful thing to have in a production application. Make sure not to click this button then close the browser without performing the import: this will leave the UCF client process running.

Example 11-1. HTML for user interface

```
<html>
<head>
 <title>Sample Applet Main Page</title>
 <script type="text/javascript">

  var winPop;

  function OpenWindow()
  {

      var props = "top=0,
                   left=0,
                   toolbar=1,
                   location=1,
                   directories=1,
                   status=1,
                   menubar=1,
```

```
                    scrollbars=0,
                    resizable=0,
                    width=300,
                    height=400";
      winPop = window.open("dfsSample-popup.html", "winPop", props);
  }

  function validate()
  {
      if(document.form1.jsessionId.value == "" ||
                                    document.form1.uid.value=="")
      {
       alert("UCF connection is not ready, please wait");
       return false;
      }
      else if(document.form1.file.value == "")
      {
       alert("Please enter a file path");
       return false;
      }
      else
      {
       return true;
      }

  }


 </script>

</head>

<body>
<h2>DFS Sample</h2>
<form name="form1"
      onSubmit="return validate()"
      method="post"
      action="http://localhost:80/dfsWebApp/DfsServiceServlet">
Enter File Path: <input name="file" type="text" size=20><br>
<input name="jsessionId" type="hidden"><br>
<input name="uid"  type="hidden"><br>

<input type="button" value="Use Ucf" onclick="OpenWindow()">
<input type="submit" value="Import">
</form>
</body>
</html>
```

Note that hidden input fields are provided in the form to store the jsessionId and uid values that will be obtained by the applet when it instantiates the UcfConnection.

Example 11-2. HTML for calling applet (dfsSample-popup.html)

```
<html>
<head>
 <TITLE>Sample Applet PopUp Page</TITLE>
 <script type="text/javascript">

  function setHtmlFormIdsFromApplet()
  {
      if (arguments.length > 0)
      {
       window.opener.document.form1.jsessionId.value = arguments[0];
         window.opener.document.form1.uid.value = arguments[1];
```

```
   }
      window.close();

  }

 </script>

</head>

<body>
<center><h2>Running Applet ........</h2><center>
<center>
       <applet CODE=SampleApplet.class
                CODEBASE=/dfsWebApp
                WIDTH=40
                HEIGHT=100
                ARCHIVE="dfsApplet.jar"><
        /applet>
</center>
</body>
</html>
```

The popup HTML downloads the applet, and also includes a Javascript function for setting values obtained by the applet in dfsSample.html (see HTML for user interface, page 196). The applet will use the Java Plug-in to call this JavaScript function.

Write the applet code for deploying and launching UCF

The applet must perform the following tasks:

1. Instantiates a UcfConnection, passing the constructor the value of the core services URL mapped through the proxy.

   UcfConnection conn = new UcfConnection(new URL("http://localhost:80/services/core"));

2. Get the values for the UCF connection (uid) and http session (jsessionId) and sets these values in the html form by calling the Javascript function defined in the JSP page.

This applet code depends on classes included in ucf-connection.jar (this will be added to the applet in the subsequent step).

Note that this Java code communicates with the Javascript in the JSP using the Java Plug-in (JSObject). For more information on the Java Plug-in, see http://java.sun.com/j2se/1.4.2/docs/guide/plugin/developer_guide/contents.html.

```
import com.emc.documentum.fs.rt.ucf.UcfConnection;

import java.applet.*;
import java.net.URL;

import netscape.javascript.JSObject;

public class SampleApplet extends Applet
{
  public void init ()
  {
    //init UCF
    System.out.println("SampleApplet init.......");
    try
```

```
    {
      UcfConnection conn = new UcfConnection(new URL("http://localhost:80/
                                                      services/core"));
      System.out.println("jsessionId=" + conn.getJsessionId() + ",
        uid=" + conn.getUid());
      JSObject win = JSObject.getWindow(this);
      win.call("setHtmlFormIdsFromApplet", new Object[]
                                           {conn.getJsessionId(),
                                            conn.getUid()});
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }
  public void start ()
  {
  }
}
```

The applet launches a UCF client process on the end-user machine, which establishes a connection to the UCF server, obtaining the jsessionId and the uid for the connection. It uses Java Plug-in JSObject to call the JavaScript function in the HTML popup, which sets the jsessionId and uid values in the user interface HTML form, which will pass them back to the servlet.

Build and bundle the applet

The applet that you construct must contain all classes from the following archives, provided in the SDK:

- ucf-installer.jar
- ucf-connection.jar
- emc-dfs-rt.jar

To create the applet, extract the contents of these two jar files and place them in the same folder with the compiled SampleApplet class, shown in the preceding step. Bundle all of these classes into a new jar file called dfsApplet.jar.

Sign the applet

Applets must run in a secure environment, and therefore must include a signed RSA certificate issued by a certification authority (CA), such as VeriSign or Thawte. The certificate must be imported by the end user before the applet code can be executed. You can obtain a temporary certificate for test purposes from VeriSign, and sign the jar file using the Java jarsigner utility. Detailed instructions regarding this are available at http://java.sun.com/javase/6/docs/technotes/guides/plugin/developer_guide/rsa_signing.html#signing.

Create a servlet for orchestrating the UCF content transfer

The function of the servlet is to perform the following tasks:

1. Receive the jsessionId and uid from the browser and use this data to configure an ActivityInfo, ContentTransferProfile, and ServiceContext such the DFS service will use the UCF connection established between the UCF client running on the end-user machine and the UCF server hosted in the DFS server application.
2. Instantiate the DFS Object service and run a create operation to test content transfer.

Note: This example uses productivity layer support. For suggestions on how to create similar functionality without the productivity layer, see Creating the servlet without the productivity layer, page 192.

Example 11-3. Sample servlet code for orchestrating UCF transfer

```
import com.emc.documentum.fs.datamodel.core.content.ActivityInfo;
import com.emc.documentum.fs.datamodel.core.content.ContentTransferMode;
import com.emc.documentum.fs.datamodel.core.content.Content;
import com.emc.documentum.fs.datamodel.core.content.FileContent;
import com.emc.documentum.fs.datamodel.core.context.RepositoryIdentity;
import com.emc.documentum.fs.datamodel.core.profiles.ContentTransferProfile;
import com.emc.documentum.fs.datamodel.core.DataPackage;
import com.emc.documentum.fs.datamodel.core.DataObject;
import com.emc.documentum.fs.datamodel.core.ObjectIdentity;
import com.emc.documentum.fs.rt.context.IServiceContext;
import com.emc.documentum.fs.rt.context.ContextFactory;
import com.emc.documentum.fs.rt.context.ServiceFactory;
import com.emc.documentum.fs.rt.ServiceInvocationException;
import com.emc.documentum.fs.services.core.client.IObjectService;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;
import java.io.IOException;
import java.io.PrintWriter;


public class DfsServiceServlet extends HttpServlet
{

    public void doPost (HttpServletRequest req,
                        HttpServletResponse res)
            throws ServletException, IOException
    {
        String file = req.getParameter("file");
        res.setContentType("text/plain");
        PrintWriter out = res.getWriter();

        try
        {
            IObjectService service = getObjectService(req);
            DataPackage dp = new DataPackage();
            DataObject vo = new DataObject(new ObjectIdentity(docbase),
              "dm_document");
            vo.getProperties().set("object_name", "testobject");
            int fileExtIdx = file.lastIndexOf(".");

      // Change extension to format accordingly in your test
           Content content = new FileContent(file, file.substring
```

```
                                            (fileExtIdx + 1));
            vo.getContents().add(content);
            dp.addDataObject(vo);

            DataPackage result = service.create(dp, null);

            System.out.println("result: " + result);
            if (result != null)
            {
                out.println("Create success: "
                            + result.getDataObjects().get(0).
                                     getIdentity().getValueAsString());
            }
            else
            {
                out.println("Create failed ");
            }
        }
        catch (Exception ce)
        {
            throw new ServletException(ce);
        }
    }

    public void doGet (HttpServletRequest req,
                       HttpServletResponse res)
           throws ServletException, IOException
    {
        doPost(req, res);
    }

    private IObjectService getObjectService (HttpServletRequest req)
            throws ServiceInvocationException
    {
        String jsessionId = req.getParameter("jsessionId");
        String uid = req.getParameter("uid");

        System.out.println("params:" + jsessionId +"," + uid );

        IServiceContext context = ContextFactory.getInstance().
                                  newContext();
        context.addIdentity(new RepositoryIdentity(docbase,
                            username, password, ""));

        ActivityInfo activity = new ActivityInfo(jsessionId,
                                null, uid, null, true);
        ContentTransferProfile ct = new ContentTransferProfile();
        ct.setTransferMode(ContentTransferMode.UCF);
        ct.setActivityInfo(activity);
        context.setProfile(ct);

        IObjectService  service = ServiceFactory.getInstance().
                                  getRemoteService(
                IObjectService.class, context, "core", serverUrl +
                                                       "/services");

        return service;
    }

    //replace the following with customer's info
    private static String username = "_USERNAME_";
    private static String password = "_PASSWORD_";
    private static String docbase = "_DOCBASE_";
    private static String serverUrl = "http://localhost:8888";
}
```

Note that you will need to provide values for username, password, and docbase fields to enable DFS to connect to your test repository.

In the sample, the getObjectService method does the work of obtaining the jsessionId and the uid from the http request.

```
String jsessionId = req.getParameter("jsessionId");
String uid = req.getParameter("uid");
```

It then constructs an ActivityInfo object, which it adds to a ContentTransferProfile, which in turn is added to the service context.

```
IServiceContext context = ContextFactory.getInstance().newContext();
context.addIdentity(new RepositoryIdentity(docbase, username, password, ""));

ActivityInfo activity = new ActivityInfo(jsessionId, null, uid, true);
ContentTransferProfile ct = new ContentTransferProfile();
ct.setTransferMode(ContentTransferMode.UCF);
ct.setActivityInfo(activity);
context.setProfile(ct);
```

Notice that in addition to the jsessionId and uid, the ActivityInfo is instantiated with two other values. The first, which is passed null, is the initiatorSessionId. This is a DFS internal setting to which the consumer should simply pass null. The second setting, which is pass true, is autoCloseConnection. Setting this to true (which is also the default), causes DFS to close the UCF connection after the service operation that transfers content. For more information on using this setting see Optimization: controlling UCF connection closure, page 179.

Finally, getObjectService instantiates the Object service using the newly created context.

```
IObjectService  service = ServiceFactory.getInstance().
                          getRemoteService(
                IObjectService.class, context, "core", serverUrl +
                                                    "/services");
return service;
```

The key is that the context has been set up to use the UCF connection *to the UCF client running on the end user machine, obtained by the applet* rather than the standard connection to the UCF client machine.

The doPost method finishes by using the service to perform a test transfer of content, using the Object service create method.

Creating the servlet without the productivity layer

To accomplish the same task as the getObjectService method without the productivity layer, you need to generate proxies for the ContextRegistryService and ObjectService using a tool like the JAX-WS reference implementation or Axis2. You can then use these proxies to create the ActivityInfo, ContentTransferProfile, and ServiceContext objects as well as the ObjectService. Because the generated proxies contain only default constructors, you have to use set methods to set values for the specific instance variables instead of passing them as arguments into the constructor. The following code demonstrates how to create the ActivityInfo, ContentTransferProfile, and ServiceContext objects with proxies that are generated by Axis2 1.4:

```
RepositoryIdentity identity = new RepositoryIdentity();
identity.setRepositoryName(docbase);
identity.setUserName(username);
```

```
identity.setPassword(password);
context serviceContext = new ServiceContext();
context.getIdentities().add(identity);
ActivityInfo activity = new ActivityInfo();
activity.setSessionId(jsessionId);
activity.setInitiatorDeploymentId(null);
activity.setActivityId(uid);
activity.setClosed(true);
ContentTransferProfile ct = new ContentTransferProfile();
ct.setTransferMode(ContentTransferMode.UCF);
ct.setActivityInfo(activity);
context.getProfiles().add(ct);
```

You can then instantiate the ObjectService with the ServiceContext factory method. Applications that do not use the productivity layer must, in addition to setting the transfer mode and activity info on the service context, provide explicit UcfContent instances in the DataObject:

```
UcfContent content = new UcfContent();
content.setLocalFilePath("path-fo-file-on-the-client-machine");
DataObject object = new DataObject();
object.getContents().add(content);
```

Tutorial: Using UCF .NET in a .NET client

The following sections provide a tutorial on a sample .NET web application that uses UCF .NET for content transfer.

Requirements

UCF .NET depends on the availability of .NET framework 3.5SP1 on the client machine on which the UCF assembly files are downloaded.

UCF .NET in a remote DFS .NET web application

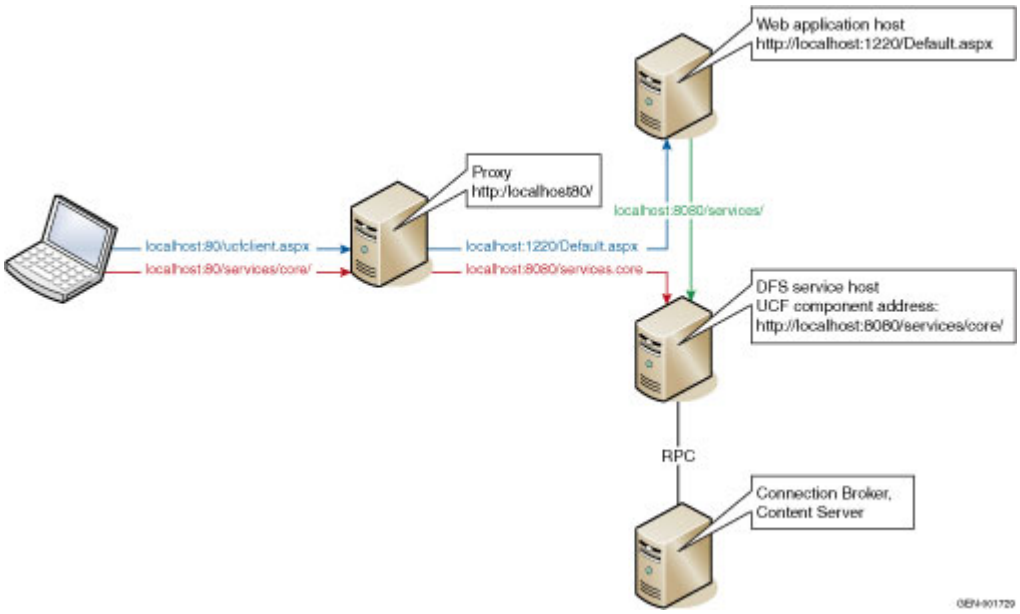
This section provides instructions for creating a test application that enables UCF transfer, using the topology described under Browser-based UCF integration, page 177.

The test application environment must include the following:

- An end-user machine that runs a 32-bit Internet Explorer
- An Apache application server used as a reverse proxy
- A .NET web server hosting a web application that includes a minimal user interface and a DFS consumer application
- An application server hosting the DFS web services
- A Content Server and a repository

For simplicity, we installed the Apache proxy server and application server on the same machine. The following network topology depicts the test application.

Figure 21. Network topology for the test application



The sample application is designed to run as follows:

1. The browser sends a request to an ASP page, which downloads an ActiveX control. Administrator privilege is required to install the ActiveX control on the client machine.
2. The ActiveX control instantiates a UCF connection, gets back a jsessionId and a uid, then sends these back to the ASP page by calling a JavaScript function.

   UCF .NET client will be installed during this phase
3. In the web application, an ASP web control uses the jsessionId, uid, and a filename provided by the user to create an ActivityInfo object, which is placed in a ContentTransferProfile in a service context. This enables DFS to perform content transfer using the UCF connection established between the UCF server on the DFS service host and the UCF client on the end-user machine.

The tasks required to build this test application are described in the following sections:

1. Set up the development environment, page 195
2. Configure the Apache reverse proxy, page 195
3. Code an HTML user interface for serving the ActiveX control, page 195
4. Create an ASP web page using the DFS Productivity Layer, page 197

Set up the development environment

The environment required for the test consists of the following:

- An end-user machine, which includes a 32-bit Internet Explorer, and has .NET framework 3.5SP1 installed (we tested using version 8.0).
- A proxy set up using the Apache application server (we tested using version 2.2).
- A .NET web server hosting the web application components, including the DFS consumer. This can be an IIS web server or Visual Studio Development server.
- An application server hosting the DFS services and runtime (which include the required UCF server components). This can be a freestanding DFS installation, or DFS running on a Content Server. The DFS installation must have its dfc.properties configured to point to a connection broker through which the Content Server installation can be accessed.
- A Content Server installation.

To create a test application, each of these hosts must be on a separate port. They do not necessarily have to be on separate physical machines. For purposes of this sample documentation, we assume the following:

- The proxy is at http://localhost:80.
- The web application is at http://localhost:1220.
- The DFS services (and the UCF components, which are included in the DFS ear file) are a freestanding installation at http://localhost:8080/services/core.

Configure the Apache reverse proxy

The Apache reverse proxy can be configured by including the following elements in the httpd.conf file:

```
# ProxyPass
# enables Apache as forwarding proxy

ProxyPass /services/ http://localhost:8080/services/
ProxyPass / http://localhost:1220/

LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_http_module modules/mod_proxy_http.so
```

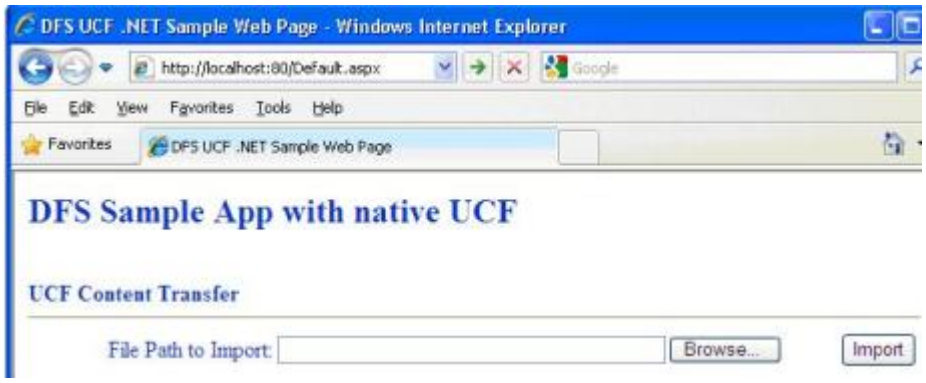
For example, a proxy running on proxy:80 forwards requests as follows:

```
http://proxy:80/services/core/runtime/AgentService.rest to
http://dfs-server:8080/services/core/runtime/AgentService.rest.
```

Code an HTML user interface for serving the ActiveX control

The sample HTML presents the user to import a file with UCF .NET.

Figure 22. User interface for UCF test application



Note: This HTML has been used for testing the ActiveX component within a CAB file provided by DFS SDK.

Javascript is used in the HTML header to launch the UcfLauncherCtrl ActiveX control and place values required by DFS in the form fields.

Example 11-4. HTML for user interface

```
!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
<title>DFS UCF .NET Sample Web Page</title>

 <script language="JavaScript" type="text/javascript">

function startUcf()
        {
            try {
                var ucfClient = document.getElementById("UcfLauncherCtrl");
                ucfClient.init();
                ucfClient.start();
            }
            catch (e) {
                alert("Fail to stat Ucf client: " + e.message);
            }
        }
</script>

</head>
```

The UcfLauncher control in UcfClient.aspx is used to pass parameters for starting the ActiveX object with some startup parameters. It is referenced in the HTML body as follows:

```
<body onload="startUcf()" text="#0033cc">
<object id="UcfLauncherCtrl" classid="CLSID:<%= UcfUtil.
                                      UcfLauncherClassId %>"
           codebase="<%= UcfUtil.UcfLauncherControlUri %>">
<param name="CONTX_PATH" value="<%= UcfUtil.UcfContextPath %>" />
<param name="CLICKONCE_REL_PATH" value="/../" />
<param name="UCF_LAUNCHER_CONFIG" value="<%= UcfUtil.UcfInstallerConfigBase64 %>" />
<param name="UCF_REQ_KEY_VALUE" value="reqKey" />
<param name="PARAM_UCF_LAUNCHER_HEADERS" value="headers" />
<param name="PARAM_UCF_LAUNCH_EXCLUDE_COOKIES" value="excludeCookies=ABCD" />
<param name="PARAM_UCF_LAUNCHER_MODE" value="2" />
```

```
<param name="RQST_ID" value="requestId" />
<param name="JSESSN_ID" value="<%= JSessionIdInput.Value %>" />
</object>
</body>
```

Note: The `UcfLauncher.cab` file serves as a remote resource file in the codebase attribute for this sample.

Although the UcfLauncher.cab file is not packaged in the emc-dfs.ear or emc-dfs.war file, you can download the DFS UcfLauncher.cab file from the the emc-dfs.ear or emc-dfs.war file by following these steps:

- Locate the `UcfLauncher.cab` file in the DFS-SDK under emc-dfs-sdk-6.7\lib\java\ucf\browser.
- Package `UcfLauncher.cab` file as emc-dfs.ear\services-core.war\UcfLauncher.cab

  After you deploy DFS, the CAB file can be downloaded from http://localhost:8080/services/core/UcfLauncher.cab#Version=6,50,0,220.

UCF .NET supports 32-bit and 64-bit browsers. DFS SDK provides two CAB files for use:

- For 32 bit ActiveX, use the UcfLauncher.cab file.
- For 64 bit ActiveX, use the UcfLauncher64.cab file.

You can locate the CAB files, in the DFS SDK, under emc-dfs-sdk-6.7\lib\java\ucf\browser.

Web server, which hosts the DFS consumer, determines which CAB file must be installed on the client, based on the request.

Note: You can implement your own ActiveX control implementation to establish UCF connection. To implement it, you must refer to `Emc.Documentum.FS.Runtime.Ucf.dll` and `UcfInstaller.dll` in your ActiveX control.

Create an ASP web page using the DFS Productivity Layer

The ASP web server page performs the following tasks:

1. Receive the jsessionId and uid from browser and instantiate ActivityInfo, ContentTransferProfile, and ServiceContext.

   DFS service will use the UCF connection established between the UCF client running on the end-user machine and the UCF server hosted in the DFS server application.

2. Instantiate the DFS Object service and run a create operation to test content transfer.

In the Javascript, add a new method to retrieve UCF ID from the ActiveX control.

```
<script language="JavaScript" type="text/javascript">
 function getUcfId() {
  try {
   MainForm.ImportPathInput.value = MainForm.ImportPath.value;
   var ucfClient = document.getElementById("UcfLauncherCtrl");
   MainForm.UcfIdInput.value = ucfClient.getUcfSessionId();
  }
  catch (e) {
   alert("Fail to get Ucf Id: " + e.message);
  }
 }
```

The Import functionality will receive UCF ID and use it for DFS service operation.

```
<table id="transferTable" style="width:100%;">
<tr>
  <td align="right" class="style4">
    File Path to Import:
  </td>

  <td class="style3">
   <input id="ImportPath" type="file" />
  </td>

  <td>
   <asp:Button ID="ImportButton" runat="server"
   onclick="ImportButton_Click" Text="Import"
   OnClientClick="getUcfId()"/>
  </td>
</tr>
</table>


protected void ImportButton_Click(object sender, EventArgs e)
{
 try
 {
  ActivityInfo activityInfo =
   DfsUtils.NewActivityInfo(JSessionIdInput.Value, UcfIdInput.Value);
  IServiceContext sc =
   DfsUtils.NewServiceContext("..", "..", "..", activityInfo);
  IObjectService service =
   DfsUtils.NewObjectService(sc, UcfUtil.ServiceUri);
  DfsDataObject doc =
   DfsUtils.ImportDocument(service, null, ImportPathInput.Value);
 }
 catch (Exception ex)
 {
 }
}

public static ActivityInfo NewActivityInfo(string jsessionId,
                                            string ucfId)
{
 ActivityInfo activityInfo = new ActivityInfo(jsessionId,
                                              null, ucfId, null, true);
 return activityInfo;
}

public static IServiceContext NewServiceContext(string repositoryName,
 string username, string password, ActivityInfo activityInfo)
{
 ContextFactory contextFactory = ContextFactory.Instance;
 IServiceContext context = contextFactory.NewContext();
 RepositoryIdentity repoId = new RepositoryIdentity();
 repoId.RepositoryName = repositoryName;
 repoId.UserName = username;
 repoId.Password = password;
 context.AddIdentity(repoId);
 ContentTransferProfile profile = new ContentTransferProfile();
 profile.TransferMode = ContentTransferMode.UCF;
 profile.ActivityInfo = activityInfo;
 context.SetProfile(profile);
 PropertyProfile propProfile = new PropertyProfile(PropertyFilterMode.
                               ALL);
 context.SetProfile(propProfile);

 return context;
}
```

```
public static IObjectService NewObjectService(IServiceContext serviceContext,
 string url)
{
 IObjectService service = ServiceFactory.Instance.
  GetRemoteService<IObjectService>(serviceContext, "core", url);
 return service;
}

public static DataObject ImportDocument(IObjectService service,
 string repositoryName, string contentFilePath)
{
 string objectName = new FileInfo(contentFilePath).Name;
 DataObject dataObject =
    new DataObject(new ObjectIdentity(repositoryName), "dm_document");
 dataObject.Properties.Set("object_name", objectName);
 dataObject.Contents.Add(new FileContent(contentFilePath, getFormat()));
 DataPackage result = service.Create(new DataPackage(dataObject), null);
 return result.DataObjects[0];
}
```

A sample project is available in the DFS SDK.

Chapter 12

Single Sign-On Using Siteminder and ClearTrust

DFS provides an integration with the Netegrity SiteMinder Policy Server and RSA ClearTrust Server single sign-on plug-ins, which are available with Content Server.

A Kerberos plug-in is also available, but has a different interface, documented in Chapter 13, Using Kerberos Authentication in DFS Clients. The information in this chapter does not apply to Kerberos authentication.

Using the productivity layer client API for SSO integration

The DFS SSO interface provides a means of passing SSO credentials to the DFS service, which in turn passes the credentials through the DFC layer to Content Server. The DFS integration assumes that the DFS client has obtained the SSO credentials, which will either be in the form of a user name and token string, or be contained within an incoming HTTP request. The client provides the credentials for a repository or set of repositories in an SsoIdentity object in the service context. In a local DFS application, the DFS productivity layer runtime will supply the expected SSO credentials to the DFC layer. In a remote DFS application, the client runtime will construct the expected HTTP request with the SSO credentials and send it over the wire to the service.

The productivity layer SSO interface is uniform, whether the client is a .NET remote client, a Java remote client, or a local Java client. In all these cases the client needs to create an instance of the SsoIdentity class and populate it with the SSO credentials. If the SSO credentials are in the form of an incoming HTTP request, the client can instantiate the SsoIdentity using this constructor in Java:

```
SsoIdentity(HttpServletRequest request)
```

Or in .NET:

```
SsoIdentity(HttpRequest request)
```

If the client has credentials in the form of a user name and token string, the client can set the user name and token string in an alternate constructor as shown in the sample below. The SsoIdentity, like other objects of the Identity data type, is set in the service context and used in instantiating the service object:

```
public void callSchemaServiceSso(String token) throws Exception
{
```

```
 SsoIdentity identity = new SsoIdentity();
 identity.setUserName(username);
 identity.setPassword(token);
 identity.setSsoType("dm_rsa");
 IServiceContext serviceContext = ContextFactory.getInstance().
                                  newContext();
 serviceContext.addIdentity(identity);
 ISchemaService service = ServiceFactory.getInstance().
  getRemoteService(ISchemaService.class, serviceContext);
 RepositoryInfo repoInfo = service.getRepositoryInfo(repository, null);
 System.out.println(repoInfo.getName());
}
```

Note that SsoIdentity, like its parent class BasicIdentity, does not encapsulate a repository name. SsoIdentity, like BasicIdentity, will be used to login to any repositories in the service context whose credentials are not specified in a RepositoryIdentity. You can use SsoIdentity in cases where the login is valid for all repositories involved in the operation, or use SsoIdentity as a fallback for a subset of the repositories and supply RepositoryIdentity instances for the remaining repositories. Also note that because SsoIdentity does not contain repository information, the user name and password is authenticated against the designated global registry. If there is no global registry defined, authentication fails.

You can provide a new SSO token with each request to handle SSO tokens that constantly change and whose expiration times are not extended on every request. Note however, that a ServiceContext object should contain only one SsoIdentity, so when you add a new SsoIdentity to the ServiceContext, you should discard the old one.

Clients that do not use the productivity layer

Clients that do not use the productivity layer will need to construct the outgoing HTTP request based on a user name and token or by copying from an incoming HTTP request. The outgoing request must contain (1) a header that will be recognized by the DFS service as the user name and (2) a cookie containing the SSO token. The name of the user name header and the token cookie must match the ones defined in the dfs-sso-config.properties configuration file on the DFS server (see Single sign-on properties, page 203).

Service context registration in SSO applications

In remote DFS applications there may or may not be an SSO proxy between the client and the service. If there is no intervening proxy, you can register the service context, then use the token that is returned in service invocations instead of the SSO credentials. If, however, there is an SSO proxy between the client and the service, you should supply an SSO token with each request.

Note: If you pass in an SsoIdentity as a delta ServiceContext when calling a service, this will override the one in the registered context on the server.

Single sign-on properties

The dfs-sso-config.properties file specifies values that the DFS services use to process HTTP requests and pass expected values to the SSO plug-in via DFC. Note that this configuration file does not apply to Kerberos authentication.

This file is required in the DFS web service application, where you will find it in the lib folder archived in emc-dfs-rt.jar. The client productivity layer runtime will also use this file to determine the header and cookie names to use for the user name and password in the HTTP request to the DFS service. If the file is not found, the DFS client runtime will use default values for the header and cookie name. Therefore, if you want to use special (non-default) names for the header and cookie in a DFS application that uses the client productivity layer, you should copy dfs-sso-config.properties to a directory on your DFS client application's classpath.

Table 24. dfs-sso-config.properties

Property Name	Description
sso.type	The SSO server type, supported values are "dm_rsa" and "dm_netegrity".
user.header.name, user.header.name.<*integer value*>	A list of possible names of HTTP headers in the HTTP request that can potentially contain user names. If more than one header from the list is found in the HTTP request, DFS uses the first header from the list that it finds.
password.cookie.name	The name of the cookie in the HTTP request that contains the SSO ticket.
sso.argument	The SSO server address. Leave this value blank for Netegrity.

Some sample settings for Netegrity Siteminder are shown below:

```
#type of single sign on server
 sso.type = dm_netegrity
 # list of possible names of headers that contain user name.
 # In case there will be more than one header with
 # user name the first found header will be used.
 user.header.name = SM_USER
 #name of the cookie containing the sso ticket
 password.cookie.name = SMSESSION
 #sso argument to specify SSO proxy server
 #so that SSO plugin in content server side will know the SSO server address
    sso.argument =
```

Chapter 13

Using Kerberos Authentication in DFS Clients

Documentum Content Server supports Kerberos authentication since version 6.7, which provides a secure Single-Sign-On (SSO) solution using Windows Integrated Authentication. Content Server supports Kerberos using the Microsoft Active Server Domain Services for Kerberos Key Distribution Center (KDC) services in the following ways:

- In a single domain.
- In two-way trusts between multiple domains in the same forest only; that is, cross-forest trusts are not supported.

Note: The DFS client and server must be in the same domain, whereas Content Server can be in a different domain.

The DFS 6.7 web services can be configured to use server-side JAX-WS handlers that interface with the Content Server Kerberos implementation. In addition, the DFS 6.7 SDK includes new classes that support Kerberos authentication for local Java clients, remote Java clients, and .NET clients. DFS SOAP clients that do not use the support classes in the SDK can authenticate against DFS web services using WS-Security headers that comply with the Kerberos Token Profile 1.1 specification.

The DFS Kerberos API deals specifically with transferring authentication information to the DFS service, using either a remote web service call or a local Java API call. The API does not address obtaining Kerberos tickets from the Kerberos Key Distribution Center (KDC). Because DFS applications are multi-tiered, Kerberos integration is based on delegated authentication. All Kerberos tokens provided to DFS through the web services API must be “forwardable”. The local Java API accepts only Kerberos Ticket Granting Tickets (TGTs).

This chapter focuses specifically on the use of the DFS Kerberos API to integrate DFS-based consumers local or remote DFS services that interact with Content Server instances that are enabled for Kerberos authentication. General information about Kerberos, as well as details regarding obtaining service tickets from a Kerberos Key Distribution Center (KDC) are outside the scope of this documentation. The following documents may be useful in that they address matters pertaining to Kerberos that are not addressed here.

For general information on Kerberos, refer to:

http://web.mit.edu/Kerberos/
http://technet.microsoft.com/en-us/library/bb742516.aspx

For information on the Java GSS API:

http://java.sun.com/products/jndi/tutorial/ldap/security/gssapi.html

http://java.sun.com/j2se/1.5.0/docs/guide/security/jgss/tutorials/index.html

For additional information on Kerberos single sign-on in Java refer to http://java.sun.com/j2se/1.5.0/docs/guide/security/jgss/single-signon.html.

Kerberos authentication in a local DFS web application

A local DFS application is one in which the DFS client and the DFS service run within the same Java virtual machine. This type of consumer requires that you use the DFS client productivity layer. For this type of application, DFS supports Kerberos authentication by providing a BinaryIdentity class that encapsulates a Kerberos credential:

```
/**
 * BinaryIdentity is not XML serializable and will not be sent over the wire.
 */
public class BinaryIdentity extends Identity
    public BinaryIdentity(Object credential,
                          BinaryIdentity.CredentialType credentialType)
```

A BinaryIdentity would typically be populated with a Ticket Granting Ticket and then added to a list of identities in the service context:

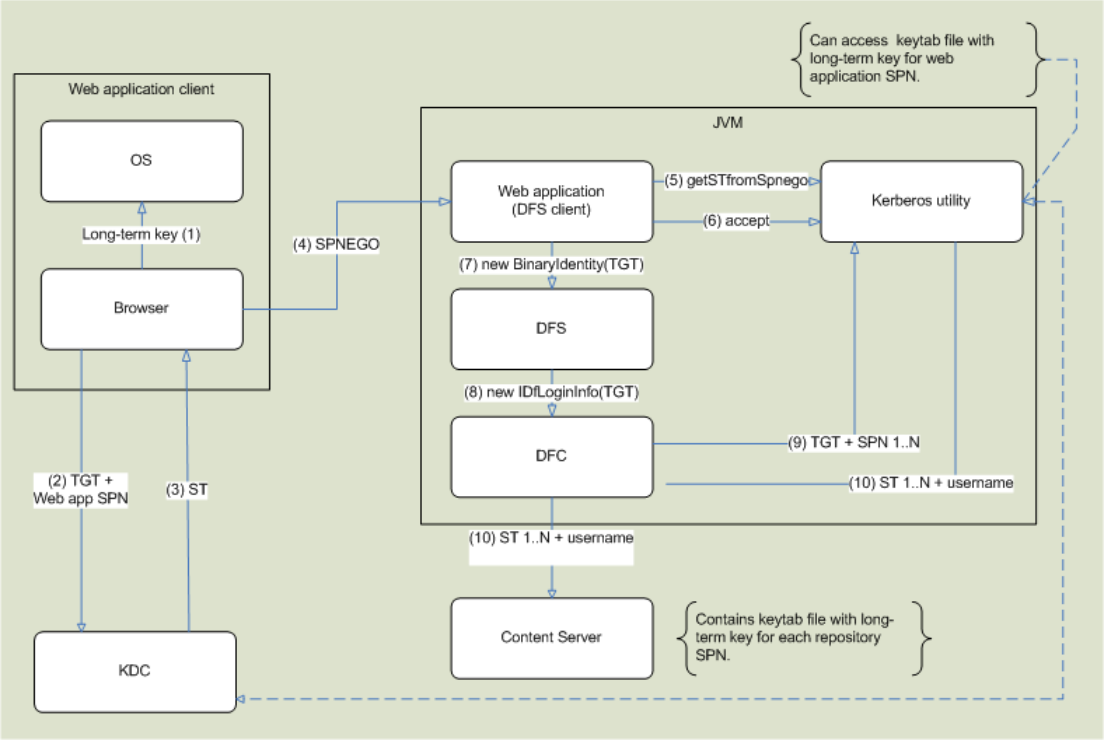
```
IObjectService service = ServiceFactory.getInstance()
  .getLocalService(IObjectService.class, context);

Object tgt = ...;
service.getServiceContext()
       .setIdentities(Arrays
       .asList((Identity)new BinaryIdentity(tgt, BinaryIdentity.
                                  CredentialType.KERBEROS_TGT)));
service.create(...);
```

Support for Kerberos authentication in local DFS applications is provided in a Kerberos utility package, com.emc.documentum.kerberos.utility. To use this utility, add krbutil.jar and jcifs-krb5-1.3.1.jar from the DFS SDK to your project classpath.

The following diagram illustrates a mainstream scenario for using local DFS services and Kerberos authentication in a web application.

Figure 23. Web application using local DFS and Kerberos authentication



In steps 1–4 in the diagram a browser client obtains a service ticket from the KDC and passes it to web application as a SPNEGO token.

Steps 5–7 are the critical steps from the point of view of DFS support:

- In steps 5 the web application calls Kerberos utility static methods to extract the ST from the SPNEGO token, and in step 6 the web application calls the Kerberos utility again to accept the ST and get a Ticket Granting Ticket (TGT) as a result. These steps could be performed with a helper method like the following:

```
public String getDFSTGTfromSPNEGOToken(String SPNEGO)
              throws GSSException
{
  String dfs_st = KerberosUtility.getSTFromSpenegoToken(SPNEGO);
  if (dfs_st != null)
   {
   return KerberosUtility.accept(m_source_spn, dfs_st);
  }
  return null;
 }
```

  Here m_source_spn is a string containing the SPN of the service to be accepted (that is, the SPN of the web application). The result is a java.lang.Object representing the TGT that can be passed to the DFS local API.

- In step 7 the web client instantiates a BinaryIdentity using the result returned by the Kerberos utility and sets the identity in the serviceContext.

  Note: Registration of the service context containing the Kerberos credentials is not currently supported.

```
IObjectService service = ServiceFactory.getInstance()
```

```
    .getLocalService(IObjectService.class, context);

Object tgt = ...;
service.getServiceContext().
    setIdentities(Arrays.asList((Identity)
    new BinaryIdentity(tgt, BinaryIdentity.
                           CredentialType.KERBEROS_TGT)));
```

In steps 8 and on, DFC uses the TGT to obtain STs from the Kerberos utility for every repository involved in the operation. These STs have the same login information as the original ST received from the client, and use Kerberos delegation (provided by the Kerberos utility) to enable Content Server to authenticate the credentials. (These steps are initiated by the DFS runtime and do not require any code in your application.)

Note: When implementing the Kerberos authentication in a multi-domain environment, you must observe the following rules:

- The source SPN accepted in the Kerberos utility cannot end with any realm name.
- The JAAS LoginModule to accept the SPN must be generated by using the Quest library. Add the following line to your DFS local client code before Kerberos handling happens which notifies the Quest library of the Kerberos name servers:

  ```
  System.setProperty("jcsi.kerberos.nameservers", "KDC machine-IP
  address");
  ```

- You cannot pass the local initialized TGT to BinaryIdentity. The TGT must be generated by using the KerberosUtility API.

Kerberos keytab file, JAAS configuration, and Kerberos configuration

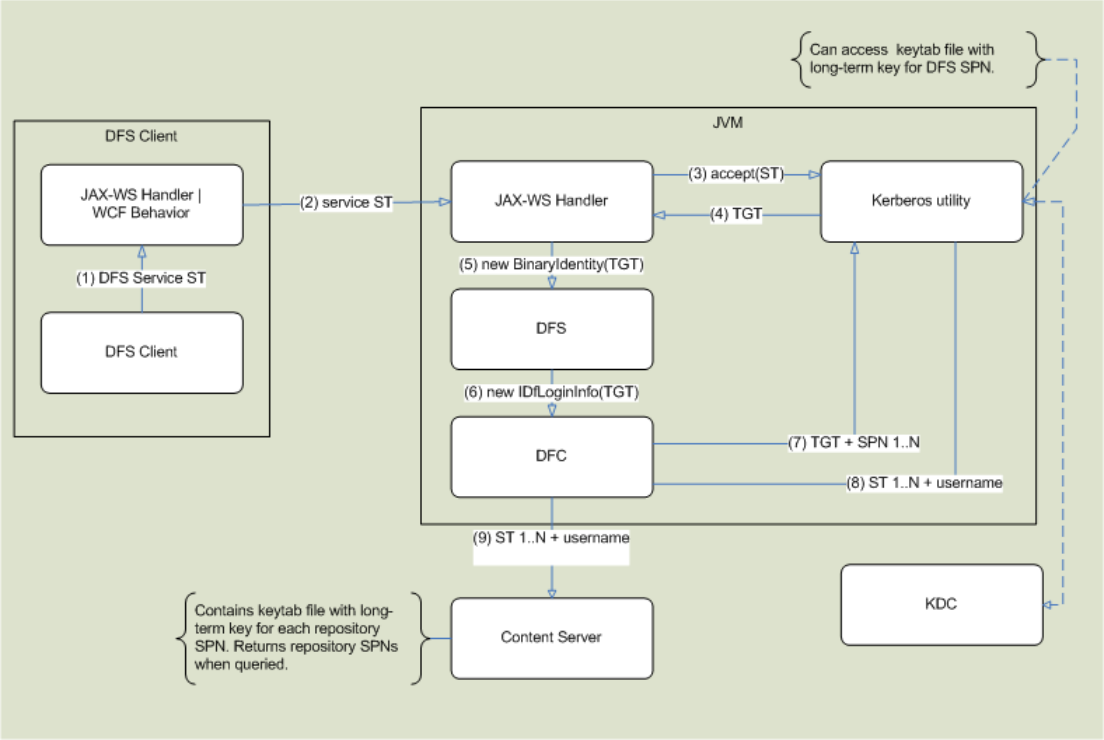
In the application illustrated in the preceding diagram, because the DFS service invocation happens inside a web application that has an SPN defined, the web application's keytab file will have to be present in order for the ST (Service Ticket) to be accepted. Once the ST is accepted, the TGT (Ticket Granting Ticket) from it can be provided to the DFS local Java API in a BinaryIdentity, as described previously.

The location of the keytab file on the server is specified in the JAAS configuration file, which needs to be configured in the application server; the configuration instructions for JAAS vary, depending on the application server you are deploying to. DFC also requires a Kerberos configuration file. For more information, refer to the *Documentum Foundation Services Deployment Guide.*

Kerberos authentication in a remote DFS client

Remote DFS consumer communication between client and server relies on SOAP, and login information is sent over the wire stored in SOAP WS-Security headers. As in the local case, the client knows about the login information it has to send to the server side and has obtained a service ticket for the DFS service from the KDC (these steps are not shown in the diagram below). This login information will be provided as handler-specific input to a JAX-WS handler or WCF behavior.

Figure 24. Kerberos authentication in a remote DFS client application



In Step 1 in the diagram, the DFS client is already assumed to be in possession of the ST obtained from the KDC. If the client is using the DFS SDK libraries, the DFS client sets the ST in a client-side JAX-WS handler or WCF behavior. The JAX-WS handler or WCF behavior takes care of serializing the Kerberos service ticket in the SOAP WS-Security header. On the server, server-side JAX-WS handlers take care of validating the Kerberos service ticket using the Kerberos utility (steps 3 and 4), and passing the ticket to the DFC layer for authentication on the Content Server (steps 5–9).

Note: Due to the Kerberos V5 anti-replay mechanism, each DFS request has to carry a unique service ticket.

From a DFS integration perspective, the main responsibility of the DFS consumer is to provide the ST that it has obtained from the KDC for the DFS service to client-side JAX-WS handlers (Java) or WCF behaviors (.NET). The following sections describe the APIs provided in the DFS SDK for Java and .NET consumers for this purpose. JAX-WS and WCF clients that do not use the productivity layer can make use of the custom JAX-WS SOAP handler or WCF endpoint behavior provided in the DFS SDK. Other types of SOAP clients will need to ensure that the Kerberos ticket is contained in the WS-Security as defined in the Oasis Kerberos Token Profile 1.1 specification. A SOAP sample excerpted from this specification is shown in Kerberos Token 1.1 security header, page 211.

DFS Kerberos remote Java API

To support Kerberos authentication in a Java remote (SOAP) client, the DFS SDK provides a set of support classes in the com.emc.documentum.fs.rt.handlers package (for details regarding these classes refer to the DFS javadocs). The classes enable a client to create an instance of a JAX-WS SOAP handler and add it to the DFS client service object so that it can be invoked by the framework during client-side

SOAP processing, as shown in the following example (You can find more Java client samples from the SDK in the `emc-dfs-sdk-6.7\samples\security\kerberos\DfsJavaKerberosDemo` directory.)

Example 13-1. Java: Invoking a service with Kerberos authentication

```
KerberosTokenHandler handler = new KerberosTokenHandler();
IObjectService service = ServiceFactory
   .getInstance().getRemoteService(..., contextRoot,
                  Arrays.asList((Handler) handler));

byte[] ticket = ...;
handler.setBinarySecurityToken(
   new KerberosBinarySecurityToken(ticket, KerberosValueType.
                                   KERBEROSV5_AP_REQ));
service.create(...)
```

The getRemoteService method is overloaded so that it can pass a list of JAX-WS handlers that the framework will invoke when creating the SOAP message.

Note: A JAX-WS client that does not use the full DFS productivity layer could also use the KerberosTokenHandler to add serialization of the Kerberos token to JAX-WS SOAP processing, by adding it to the handler chain without using the getRemoteService productivity-layer method.

DFS Kerberos remote .NET client API

To support Kerberos authentication in a Windows Communication Framework (WCF) client, the DFS SDK provides a set of support classes in the Emc.Documentum.FS.Runtime.Behaviors namespace. (For details on these classes refer to the CHM documents in the SDK.) The classes enable a client to create an instance of the WCF behavior and add it to the DFS client service object so that it can be invoked by the framework during client-side SOAP processing, as shown in this example:

Example 13-2. C#: Invoking a service with Kerberos authentication

```
KerberosTokenHandler handler = new KerberosTokenHandler();
List<IEndpointBehavior> handlers = new List<IEndpointBehavior>();
handlers.Add(handler);
IObjectService service = ServiceFactory
   .Instance.GetRemoteService<IObjectService>(..., contextRoot,
                                              handlers);

byte[] ticket = ...;
handler.SetBinarySecurityToken(
    new KerberosBinarySecurityToken(ticket, KerberosValueType.
                                    GSS_KERBEROSV5_AP_REQ));
service.create(...);
```

The GetRemoteService method has been overloaded so that it can pass a list of custom behaviors that the framework will invoke when creating the SOAP message.

Note:

- A WCF client that does not use the full DFS productivity layer could also use the KerberosTokenHandler to add serialization of the Kerberos token to WCF SOAP processing, by

adding the custom endpoint behavior without using the getRemoteService productivity-layer method.

- To generate the service ticket for .NET users, the Kerberos delegation level needs to be enabled by setting the Impersonation Level to Delegate (`ImpersonationLevel.Delegate`).

Kerberos Token 1.1 security header

The following shows an example of a Kerberos ticket serialized in the SOAP WS-Security header as described in the Oasis Kerberos Token Profile 1.1 specification. The ticket must be Base64 encoded.

```
<S11:Envelope xmlns:S11="..." xmlns:wsu="...">
    <S11:Header>
        <wsse:Security xmlns:wsse="...">
        <wsse:BinarySecurityToken
EncodingType="http://docs.oasis-open.org/wss/2004/01/⇒
oasis-200401-wss-soap-message178security-1.0#Base64Binary"
ValueType="http://docs.oasis179open.org/wss/⇒
oasis-wss-kerberos-token-profile-1.1#Kerberosv5_AP_REQ"
wsu:Id="MyToken">boIBxDCCAcCgAwIBBaEDAgEOogcD...
        </wsse:BinarySecurityToken>
 ...
 </wsse:Security>
    </S11:Header>
    <S11:Body>
...
    </S11:Body>
</S11:Envelope>
```

Enabling DFS JAX-WS handlers for Kerberos

A remote DFS consumer that uses Kerberos requires that the JAX-WS Kerberos handlers be enabled in the DFS web application. This has to be done during DFS deployment by modifying the authorized-service-handler-chain.xml deployment descriptor and by adding some required jar files to the DFS web application.

For instructions on how to do this, refer to the *Documentum Foundation Services Deployment Guide.*

DFS provides the following default Kerberos server token handler:

com.emc.documentum.fs.rt.handlers.KerberosTokenServerHandler

Although you are allowed to customize a Kerberos server token handler, it is strongly recommended to use the default one to get the best performance. For more information about JAX-WS handlers, see JAX-WS server handlers, page 212.

JAX-WS server handlers

DFS server relies on the following JAX-WS server handler chain to perform identity authentications. A sample can be found from the SDK in `emc-dfs-sdk-6.7\etc\authorized-service-handler-chain.xml`. These handlers are processed in the following order.

- ServerContextHandler

  This handler extracts the following identities in sequence from the SOAP header or the HTTP header:
  - Registered service context token
  - Identities found in service context
  - SSO cookies from the request HTTP header

- KerberosTokenServerHandler

  This handler handles Kerberos tickets from the SOAP header and constructs BinaryIdentity.

- AuthorizationHandler

  This handler Iterates identities found in previous handlers (ServerContextHandler, and/or KerberosTokenServerHandler) and tries to perform authentication for each identities.

  If any identity authentication succeeds, the authentication process ends immediately. If all identities' authentications fail, the authentication process fails.

Do not set multiple credentials to a client-side service context or handlers unless you have to enable multiple authentication schemes. For example, if Kerberos SSO is the only designed authentication scheme, do not set `RepositoryIdentity` to `ServiceContext`.

Other server configuration requirements

In the application illustrated in the preceding diagram, the server-side JAX-WS handlers call the Kerberos utility to accept the ST provided by the client. The Kerberos utility requires that the keytab file that contains the DFS SPN must be available in order to accept the ST.

The location of the keytab file on the server is specified in the JAAS configuration file, which needs to be configured in the application server; the configuration instructions for JAAS vary, depending on the application server you are deploying to.

In addition, a Kerberos configuration file (krb5.ini) is required; this file is either placed in a well-known location or referenced in an application deployment descriptor.

For more detailed information on the Kerberos keytab file, JAAS configuration, and krb5.ini, refer to the *Documentum Foundation Services Deployment Guide.*

Limitations on Kerberos support

The following limitations apply to Kerberos support in this release:

- Due to the multi-tiered nature of DFS applications, Kerberos integration is based on delegated authentication. All Kerberos tokens provided to DFS through the web services API must be "forwardable". The local Java API accepts only Kerberos Ticket Granting Tickets (TGTs).
- DFS services don't accept SPNEGO tokens. (However, the Kerberos utility provided with the DFS SDK includes a method for unwrapping SPNEGO tokens.)
- Registering a ServiceContext with Kerberos credentials is not supported.
- Kerberos-based message level security is not supported.
- Support for .NET integration is limited to remote web services API (in the client productivity layer). There is no Kerberos Utility for .NET.
- Wrapped GSS API Kerberos tokens are not supported.

Chapter 14

Integrating with IBM Tivoli Access Manager for E-business WebSEAL

IBM Tivoli Access Manager for e-business WebSEAL is a high-performance, multi-threaded web server that applies fine-grained security policy to a protected network. WebSEAL incorporates back-end web application server resources into its security policy, and can provide single sign-on (SSO) solutions. WebSEAL acts as a reverse web proxy by receiving HTTP or HTTPS requests from a web browser and delivering content from its own web server or from back-end web application servers. Requests passing through WebSEAL are evaluated by its own authorization service to determine whether the user is authorized to access the requested resource.

EMC Documentum can integrate with WebSEAL, its SSO solution, or any other SSO solution supported by WebSEAL.

The related IBM documentation at http://publib.boulder.ibm.com/infocenter/tivihelp/v2r1/index.jsp provides more information about installing and configuring the WebSEAL server.

In a Documentum/WebSEAL integration the relationship of trust is established among all subsystems within a WebSEAL-protected *web object space.* Once a request is allowed into the object space, subsystems within the space will trust that the request is properly authorized and authenticated. There is no second-pass authentication performed for these requests at any layer of the Documentum stack (that is, DFS, DFC, or Content Server). The Documentum subsystems will have no means to verify whether or not a request has been genuinely authenticated by WebSEAL, so to avoid any security threats, the network must be configured in such a way that the WebSEAL proxy is the only point of access to the Documentum system.

To complete the trusted subsystem pattern, there needs to be a separate relationship of trust established between the DFS/DFC tier and Content Server so that only trusted DFS/DFC instances are granted access to repositories. This is accomplished using *DFC Principal Authentication*, in which DFC uses a trusted principal to log onto Content Server on behalf of a user without authentication. For more information see Configuring a trust relationship between DFS/DFC and Content Server, page 219.

Note: For detailed information on WebSEAL, refer to the WebSEAL Administration Guide.

Client integration

In a WebSEAL deployment all authentication is performed by a WebSEAL, which is trusted by other subsystems within the WebSEAL-protected web object space. The client might provide a

prompt for the login information (in the case of user name/password authentication, for example), or might obtain all the required authentication information transparently for the user (as in the case of Kerberos or smartcard/certificate based authentication). Regardless of the means of obtaining the login information, it is provided to WebSEAL for authentication. Once a successfully authenticated request is allowed through, WebSEAL returns a state cookie named PD-S-SESSION-ID which can be reused by the client to avoid authenticating subsequent requests.

From a DFS perspective there are two client-side integration types: a browser integration for a DFS based web application, and a native web services integration (often a desktop application). Both can follow the HTTP SSO interaction pattern just described.

Browser integration

Most of the time, WebSEAL integrations will not require significant effort on the part of the web application developer to adopt a new authentication mechanism, because WebSEAL is optimized for this type of integration. Browsers can transparently handle HTTP redirects to login pages, or authentication mechanisms based on HTTP Negotiate. This makes for easy integration with WebSEAL as it enables the proxy to display a username/password prompt in the browser window (in the case of redirects), and to negotiate Kerberos or other authentication tokens transparently for both the end user and the web application. An expired session cookie can be gracefully handled by negotiating a new one using forms-based or any other type of authentication.

Web services integration

Web service consumers do not have native support for negotiating a session cookie with the WebSEAL proxy. They normally adhere to a stateless communication pattern, providing all required login information with every request. They can make use of established WebSEAL sessions by providing the session cookie (by default named PD-S-SESSION-ID) obtained from the WebSEAL proxy in service requests. For web service consumers, providing a wrong or expired session cookie will not automatically result in negotiating a new one, but will result in the server throwing an explicit exception, which the web service consumer will need to detect and handle. So for developers of web service consumers, as opposed to browser integrations, adopting a new authentication mechanism will require extra effort in coding the client logic, as well as effort in configuring the WebSEAL proxy.

Note: For information on WebSEAL session cookie names, see the WebSEAL Administration Guide.

Productivity layer consumers

DFS Productivity layer consumers rely on proprietary DFS authentication schemes, as well as standard WS-Security specifications, which WebSEAL does not support. As a result, authenticating to WebSEAL using the DFS productivity layers “out-of-the-box” is not possible. However, it is possible to reuse established WebSEAL sessions by setting the PD-S-SESSION-ID on the outgoing web service request. DFS recommends implementing a custom client-side JAX-WS handler or .NET WCF endpoint behavior for this purpose.

The following productivity-layer API should be used to add a JAX-WS handler to the client-side handler chain:

```
public <T> T getRemoteService(
    Class<T> wsInterface, IServiceContext serviceContext,
            String serviceModule, String contextRoot,
            List<Handler> handlerChain
) throws ServiceInvocationException
```

The following C# API can be used to add a custom WCF endpoint behavior:

```
public T GetRemoteService<T>(
    IServiceContext serviceContext, String serviceModule,
  String contextRoot, List<IEndpointBehavior> behaviors
)
```

The DFS SDK provides a sample JAX-WS client-side SOAP handler, CookieSettingJaxwsHandler.java, and a client-side WCF endpoint behavior, CookieSettingWcfBehavior.cs, which you can use as a templates for your custom client-side handler or behavior.

Web service WSDL-only consumers

DFS web services consumers that do not use the productivity layer integrate with WebSEAL using the same technique as productivity-layer consumers, except that they must use the API provided by their web services framework to add the custom SOAP handler to the handler chain or, in WCF, to add a custom endpoint behavior to the client.

Some WSDL-only consumers might decide to use HTTP Basic Authentication for web services to integrate with WebSEAL. When doing so, they should keep in mind that this kind of integration might not be possible when UCF content transfer is required as UCF connections need an established WebSEAL session to initialize.

The DFS SDK provides a sample JAX-WS client-side SOAP handler, CookieSettingJaxwsHandler.java, and a client-side WCF endpoint behavior, CookieSettingWcfBehavior.cs, which you can use as a templates for your custom client-side handler or behavior.

UCF integration

A UCF connection will not be successfully established unless it is provided a valid WebSEAL session.

In the case of both browser-based and thick-client UCF integrations, the UCF connection has to "piggyback" on an established WebSEAL session by providing a valid PD-S-SESSION-ID cookie to a UcfConnection instance, using the following API:

```
public UcfConnection(URL ucfServerUrl, String targetDeploymentId,
 com.emc.documentum.fs.rt.ucf.Cookie... cookies) throws UcfException
```

DFS-orchestrated UCF content transfer is not supported with WebSEAL authentication, so you must use client-orchestrated UCF to provide the PD-S-SESSION-ID cookie when establishing a UCF connection, as shown in the following listing:

```
IServiceContext context = ContextFactory.getInstance().newContext();
UcfConnection c = new UcfConnection(
```

```
    ucfServerUrl, targetDeploymentId, cookies
);
ActivityInfo info = new ActivityInfo(true);
info.setActivityId(c.getUid());
info.setCookies(c.getCookies());
ContentTransferProfile p = new ContentTransferProfile();
p.setTransferMode(ContentTransferMode.UCF);
p.setActivityInfo(info);
context.setProfile(p);
```

WSDL required by JAX-WS clients

JAX-WS web service consumers require a WSDL, either local or downloaded from a remote server to instantiate (see JAXWS Issue 876). It is technically not possible to download WSDLs from servers in a WebSEAL protected web object space. To work around this issue it is recommended to use any of the following solutions:

- use the JDK's CookieHandler/CookieManager to set a system-wide cookie handler
- exclude the WSDL URLs from the protected web object space
- use a local copy of the service WSDLs

,

Java productivity-layer consumers can set the following DFS runtime property to instruct the DFS runtime to use the local WSDLs shipped with the DFS SDK:

```
dfs.wsdl.location=file:///${emc-dfs-sdk}/etc/wsdl
```

Set this property in the dfs-client.xml client configuration file as shown in the example below:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<DfsClientConfig defaultModuleName="core"
                                 registryProviderModuleName="core">
    <ModuleInfo .../>
    <DfsClientProperty name="dfs.wsdl.location"
        value="file:///C:/shared/emc-dfs-sdk-6.7/etc/wsdl"/>
</DfsClientConfig>
```

WSDL-only consumers have to explicitly provide the path to a local WSDL when instantiating their javax.xml.ws.Service objects.

Note: .NET productivity layer consumers are not affected by this limitation as WCF does not require a WSDL to instantiate a web services consumer.

Registration of service context using ContextFactory is not supported

Productivity-layer consumers will not be able to use ContextFactory to register a ServiceContext due to the fact that it does not provide an API to manipulate HTTP cookies. The recommended workaround for this limitation is to not register the ServiceContext. (Context registration is an optional function.)

UrlContent

There is no API to provide cookies when working with UrlContent in this release. Customers are expected to handle remote content download independently in this case. A UrlContent instance exposes the location (URL) of the content for this purpose.

Configuring a trust relationship between DFS/DFC and Content Server

In order to set up a trusted subsystem environment, you need to configure the trust relationship between the DFC instance that will process the DFS PrincipalIdentity and the Content Server that will allow the DFC instance to log in on behalf of the user configured in the PrincipalIdentity. This is accomplished using DFC principal authentication, in which DFC uses a trusted principal to log onto Content Server. The trusted principal can be defined in either of two ways:

- As a superuser. In this case the superuser account must be defined in the repository, and the user must be configured in the DFS application in a file named trust.properties. You can find a template trust.properties file in the etc folder of the DFS SDK.
- As the DFC instance itself. In this case the DFC instance must be configured as privileged. This can be accomplished using Documentum Administrator, as described in the *Documentum Administrator User Guide.*

In either case the PrincipalIdentity API can be used to extract the user name from login information provided by WebSEAL allowing DFC to log in on behalf of that user. The following sections describe how this is accomplished.

Server-side integration

Part of the simplicity and flexibility of the WebSEAL integration comes from the fact that no matter what authentication mechanism the client uses to negotiate the session cookie (HTTP Basic Authentication, certificate-based authentication, Kerberos tokens, and so on), the subsystems that receive the request via the WebSEAL proxy always gets the same login information. The login information can be in any combination of the following formats:

- HTTP iv-user header
- HTTP iv-creds header
- LTPA cookie

For a trusted subsystem authentication pattern, an iv-user header is convenient because it contains the user name of the authenticated user, so this is recommended for server-side integration. The format of the login information (which can be any of those listed above or a combination), can be specified using a configuration setting on the WebSEAL server.

From the perspective of a DFS application developer, the server-side integration can either deal with:

- a set of applications that all support a trusted subsystem authentication pattern
- or with a set of applications some of which support and some of which do not support explicit authentication.

Uniform trusted subsystem

Applications that all support establishing a relationship of trust with each other can be combined in a *uniform trusted subsystem*. An example of such a subsystem consisting of three components would be WebSEAL, DFS/DFC, and Content Server. Bound together by two relationships of trust, they can work as a single component where authentication is concerned. The first relationship of trust would be between DFS/DFC and WebSEAL, with DFS/DFC trusting the iv-user header provided by WebSEAL. The second relationship of trust is between Content Server and DFS/DFC, Content Server trusting the specific privileged DFC instance packaged with DFS to execute an operation on behalf of the user name extracted from the iv-user HTTP header.

DFS provides the following API to enabled the privileged DFC instance to log the user into the repository when all components in the subsystem support a relationship of trust:

```
import com.emc.documentum.fs.datamodel.core.context.PrincipalIdentity;
IServiceContext context = ContextFactory.getInstance().newContext();
context.addIdentity(new PrincipalIdentity("user"));
```

Mixed trusted and authenticating subsystem

In case when not all components of a system support a trusted subsystem authentication pattern, DFS allows providing separate identities for separate parts of the system. Each subsystem will be responsible for identifying the appropriate identity and using it for authentication purposes if required.

DFS provides the following API to enable log in the user making a request when some, but not all components in the subsystem support a relationship of trust:

```
import com.emc.documentum.fs.datamodel.core.context.PrincipalIdentity;
IServiceContext context = ContextFactory.getInstance().newContext();
context.addIdentity(new PrincipalIdentity("user"));
context.addIdentity(new RepositoryIdentity("repository",
                                           "user", "password"));
```

Web application integration

A client-side browser would normally interact with a web application that resides inside of the WebSEAL web object space. In such applications the browser will handle the WebSEAL authentication transparently for the web application. Once the HTTP request reaches the application it has already been authenticated and authorized by WebSEAL. The responsibility of the web application developer is to extract the iv-user header from the incoming HTTP request (for example using a servlet filter)

and obtain a user name from it. The user name can then be set in a PrincipalIdentity, as shown in the preceding sections, which enables DFC to log in on behalf of the pre-authenticated user.

Note that to accomplish this, the DFC instance in the web application must either be configured as *privileged*, or there must be a superuser account configured in trust.properties on the application class path (see Configuring a trust relationship between DFS/DFC and Content Server, page 219).

Web service integration

A thick client would normally interact directly with a DFS web service that resides inside of the WebSEAL web object space. Once the HTTP request reaches the web service it has already been authenticated and authorized by WebSEAL. The responsibility of the web service developer in this case is to use a server-side JAX-WS handler to extract the iv-user header from the incoming HTTP request and obtain a user name from it. The DFS SDK includes a sample server-side JAX-WS SOAP handler (WebsealIvUserHandler) which you can use as a template for your custom handler.

To configure the DFS service application to add your custom handler to the handler chain, follow this procedure:

To add a custom SOAP handler to the DFS server application

1. Open up the services EAR file and locate APP-INF/classes/authorized-service-handler-chain.xml. If you are deploying a WAR file, locate WEB-INF/classes/authorized-service-handler-chain.xml.
2. Package your custom handler in a jar and add it to APP-INF/lib (in EAR files) or WEB-INF.lib (in WAR files).
3. Insert a descriptor for your custom handler, as shown below, then save the file. Note that the handler should be inserted in the middle of the handler chain, as specified in the comments below:

```
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee">
  <handler-chain>
   <handler>
     <handler-name>Authorization</handler-name>
     <handler-class>com.emc.documentum.fs.rt.impl.
       handler.AuthorizationHandler
  </handler-class>
  </handler>
  <handler-name>YourWebsealIvUserHandler</handler-name>
   <handler-class>com.acme.handler.YourWebsealIvUserHandler
     </handler-class>
  </handler>
        <!-- Any handler using ContextFactory, like
  KerberosTokenServerHandler or AuthorizationHandler must be
  inserted above this comment -->
   <handler>
     <handler-name>Context Local Registry</handler-name>
     <handler-class>com.emc.documentum.fs.rt.impl.handler.
                                    ServerContextHandler
   </handler-class>
   </handler>
   <!-- Any handler modifying DFS SOAP headers must be below this comment -->
 </handler-chain>
</handler-chains>
```

Note that to authenticate on behalf of the user identified in the PrincipalIdentity, the DFC instance packaged with DFS must either be configured as *privileged*, or there must be a superuser account

configured in trust.properties on the application class path (see Configuring a trust relationship between DFS/DFC and Content Server, page 219).

Preserving JSESSIONID cookie name

In WebSEAL junction creation, the -j option will modify the value of the path attribute of a Set-Cookie header to give identical cookies for different back-end applications.

The -j junction option provides an additional feature to modify the cookie name by prepending a special string:

```
AMWEBJCT!<jct-name>!
```

For example, if a cookie named JSESSIONID arrives across a junction called /jctA, the cookie name is changed to :

```
AMWEBJCT!jctA!JSESSIONID
```

However, as a front-end application of the WebSEAL proxy server, the DFS client depends on the JSESSIONID cookie for its operations. Therefore, the JSESSIONID cookie-renaming behavior should be disabled. There are two options for accomplishing this:

- Preserve the names of all cookies.

  Prevent renaming of non-domain cookies across a specific -j junction by configuring that junction with the -n option.

- Preserve the names of specified cookies.

  The name entry in the [preserve-cookie-names] stanza of the WebSEAL configuration file allows you to list the specific cookie names that are not to be renamed by WebSEAL. For example:

  ```
  [preserve-cookie-names]
  name = JSESSIONID
  ```

For further details, refer to "Handling cookies from servers across multiple -j junctions" in http://publib.boulder.ibm.com/infocenter/tivihelp/v2r1/index.jsp?topic=/com.ibm.itame.doc/am611_webseal_admin620.htm.

Chapter 15

Comparing DFS and DFC

This chapter provides a general orientation for users of DFC who are considering creating DFS client applications. It compares some common DFC interfaces and patterns to their functional counterparts in DFS. This document covers the following topics:

Fundamental differences

DFS is a service-oriented API and an abstraction layer over DFC. DFS is simpler to use than DFC and will allow development of client applications in less time and with less code. It also greatly increases the interoperability of the Documentum platform and related technologies by providing WSDL interface to SOAP clients generally, as well as client libraries for both Java and .NET. However, because it exposes a data model and service API that are significantly different from DFC, it does require some reorientation for developers who are used to DFC.

When programming in DFS, some of the central and familiar concepts from DFC are no longer a part of the model.

Session managers and sessions are not part of the DFS abstraction for DFS consumers. However, DFC sessions are used by DFS services that interact with the DFC layer. The DFS consumer sets up identities (repository names and user credentials) in a *service context*, which is use to instantiate service proxies, and with that information DFS services take care of all the details of getting and disposing of sessions.

DFS does not have (at the exposed level of the API) an object type corresponding to a SysObject. Instead it provides a generic DataObject class that can represent any persistent object, and which is associated with a repository object type using a property that holds the repository type name (for example “dm_document”). Unlike DFC, DFS does not generally model the repository type system (that is, provide classes that map to and represent repository types). Any repository type can be represented by a DataObject, although some more specialized classes can also represent repository types (for example an Acl or a Lifecycle).

In DFS, we've chosen to call the methods exposed by services *operations*, in part because this is what they are called in the WSDLs that represent the web service APIs. Don't confuse the term with DFC operations—in DFS the term is used generically for any method exposed by the service.

DFS services generally speaking expose a just a few service operations (the TaskManagement service is a notable exception). The operations generally have simple signatures. For example the Object service update operation has this signature:

```
DataPackage update(DataPackage dataPackage, OperationOptions options)
```

However, this "simple" operation provides a tremendous amount of power and flexibility. It's just that the complexity has moved from the number of methods and the complexity of the method signature to the objects passed in the operation. The operation makes a lot of decisions based on the composition of the objects in the DataPackage and relationships among those objects, and on profiles and properties provided in the operationOptions parameter or set in the service context—these settings are used to modify the default assumptions made by the service operation. The client spends most of its effort working with local objects, rather than in conversation with the service API.

Login and session management

The following sections compare login and session management in DFC and DFS. Generally speaking, session management is explicitly handled by a DFC client using the IdfSessionManager and IdfSession interfaces. DFS provides a higher level of abstraction (the notion of service context), and in terms of the interface presented to DFS consumers, handles session management behind the scenes. However, if you are developing DFS services that use DFC, you do need to get and release managed sessions.

DFC: Session managers and sessions

This section describes sessions and session managers, and provides examples of how to instantiate a session in DFC.

Understanding Sessions

To do any work in a repository, you must first get a session on the repository. A session (IDfSession) maintains a connection to a repository, and gives access to objects in the repository for a specific logical user whose credentials are authenticated before establishing the connection. The IDfSession interface provides a large number of methods for examining and modifying the session itself, the repository, and its objects, as well as for using transactions (refer to IDfSession in the Javadocs for a complete reference).

Understanding Session Managers

A session manager (IDfSessionManager) manages sessions for a single user on one or more repositories. You create a session manager using the DfClient.newSessionManager factory method.

The session manager serves as a factory for generating new IDfSession objects using the IDfSessionManager.newSession method. Immediately after using the session to do work in the repository, the application should release the session using the IDfSessionManager.release() method in a *finally* clause. The session initially remains available to be reclaimed by session manager instance that released it, and subsequently will be placed in a connection pool where it can be shared.

Getting a Session Manager

To get a session manager, encapsulate a set of user credentials in an IDfLoginInfo object and pass this with the repository name to the IDfSessionManager.setIdentity method. In simple cases, where the session manager will be limited to providing sessions for a single repository, or where the login credentials for the user is the same in all repositories, you can set a single identity to IDfLoginInfo.ALL_DOCBASES (= *). This causes the session manager to map any repository name for which there is no specific identity defined to a default set of login credentials.

```
/**
 * Creates a simplest-case IDfSessionManager
 * The user in this case is assumed to have the same login
 * credentials in any available repository
 */
public static IDfSessionManager getSessionManager
  (String userName, String password) throws Exception
{
  // create a client object using a factory method in DfClientX

  DfClientX clientx = new DfClientX();
  IDfClient client = clientx.getLocalClient();

  // call a factory method to create the session manager

  IDfSessionManager sessionMgr = client.newSessionManager();

  // create an IDfLoginInfo object and set its fields
  IDfLoginInfo loginInfo = clientx.getLoginInfo();
  loginInfo.setUser(userName);
  loginInfo.setPassword(password);

  // set single identity for all docbases
  sessionMgr.setIdentity(IDfSessionManager.ALL_DOCBASES, loginInfo);
  return sessionMgr;
}
```

If the session manager has multiple identities, you can add these lazily, as sessions are requested. The following method adds an identity to a session manager, stored in the session manager referred to by the Java instance variable sessionMgr. If there is already an identity set for the repository name, setIdentity will throw a DfServiceException. To allow your method to overwrite existing identities, you can check for the identity (using hasIdentity) and clear it (using clearIdentity) before calling setIdentity.

```
public void addIdentity
  (String repository, String userName, String password)
   throws DfServiceException
{
  // create an IDfLoginInfo object and set its fields

  IDfLoginInfo loginInfo = this.clientx.getLoginInfo();
  loginInfo.setUser(userName);
  loginInfo.setPassword(password);
```

```
  if (sessionMgr.hasIdentity(repository))
  {
    sessionMgr.clearIdentity(repository);
  }
  sessionMgr.setIdentity(repository, loginInfo);
}
```

Note that setIdentity does not validate the repository name nor authenticate the user credentials. This normally is not done until the application requests a session using the getSession or newSession method; however, you can authenticate the credentials stored in the identity without requesting a session using the IDfSessionManager.authenticate method. The authenticate method, like getSession and newSession, uses an identity stored in the session manager object, and throws an exception if the user does not have access to the requested repository.

DFC sessions in DFS services

In DFS sessions are handled by the service layer and are not exposed in the DFS client API. DFS services, however, do and must use managed sessions in their interactions with the DFC layer. A DFS service that uses DFC absolutely must get its instance of the DFC session manager (that is, an instance of the IDfSessionManager interface) through the DFS layer, using the getSessionManager static method of the DfcSessionManager class. This turns over much of the complexity of dealing with session managers, identities, and sessions to the DFS framework. DFS maintains a cache of session managers that are associated by a token with a service context kept in thread-local storage. DfcSessionManager.getSessionManager retrieves a session manager from the cache based on the token stored in the serviceContext, and takes care of the details of populating the session manager with identities stored in the service context. The service context itself is created based on data passed in SOAP headers from remote clients, or on data passed by a local client during service instantiation.

From the viewpoint of the custom DFS service, the essential thing is to get the session manager using DfcSessionManager.getSessionManager, then invoke the session manager to get a session on a repository. To get a session, the service needs to pass a string identifying the repository to the IDfSessionManager.getSession method, so generally a service will need to receive the repository name from the caller in one of its parameters. Once the service method has the session, it can invoke DFC methods on the session within a try clause and catch any DfException thrown by DFC. In the catch clause it should wrap the exception in a custom DFS exception (see Creating a custom exception, page 131), or in a generic ServiceException, so that the DFS framework can handle the exception appropriately and serialize it for remote consumers. The session must be released in a finally clause to prevent session leakage. This general pattern is shown in the listing below.

```
import com.emc.documentum.fs.rt.context.DfcSessionManager;
...

public void myServiceMethod(DataObject dataObject) throws ServiceException
{
    IDfSessionManager manager = null;
    IDfSession session = null;
    try
    {
        manager = DfcSessionManager.getSessionManager();
        session = manager.getSession(dataObject.getIdentity().getRepositoryName());
        // do DFC stuff with DFC session
    }
    catch (DfException e)
    {
        throw new ServiceException("E_EXCEPTION_STRING", e, dataObject.getIdentity());
    }
    finally
    {
        if (manager != null && session != null)
        {
            manager.release(session);
        }
    }
}
```

If your DFS application does not include custom services, or if your custom services do not use DFC, then you need not be too concerned about programmatic management of sessions. However, it's desirable to understand what DFS is doing with sessions because some related aspects of the runtime behavior are configurable using DFS and DFC runtime properties. As stated above, DFS maintains a cache of session managers. This cache is cleaned up at regular intervals (by default every 20 minutes), and the cached session managers expire at regular intervals (by default every 60 minutes). The two intervals can be modified in dfs-runtime.properties by changing dfs.crs.perform_cleanup_every_x_minutes and dfs.crs.cache_expiration_after_x_minutes. Once the session is obtained, it is managed by the DFC layer, so configuration settings that influence runtime behavior in regard to sessions, such as whether the sessions are pooled and how quickly their connections time out, are in dfc.properties (and named dfc.session.*). These settings are documented in the dfcfull.properties file, and DFC session management in general is discussed in the *Documentum Foundation Classes Development Guide.*

Note that for each request from a service consumer, DFS will use only one IDfSessionManager instance. All underlying DFC sessions are managed (and may be cached, depending on whether session pooling is enabled) by this instance. If there are multiple simultaneous DFS requests, there should theoretically be an equivalent number of active DFC sessions. However, the number of concurrent sessions may be limited by configuration settings in dfc.properties, or by external limits imposed by the OS or network on the number of available TCP/IP connections.

Creating objects and setting attributes

This section compares techniques for creating objects and setting attributes in DFC and DFS.

Creating objects and setting attributes in DFC

This section describes the process for creating objects using DFC interfaces.

Creating a document object

Using DFC, you can create an empty document object, then populate it with values you provide at runtime.

Example 15-1. The TutorialMakeDocument class

```
package com.emc.tutorial;

import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;

public class TutorialMakeDocument
{
   public TutorialMakeDocument()
   {
   }

   public Boolean makeDocument(
      IDfSessionManager sessionManager,
      String repositoryName,
      String documentName,
      String documentType,
      String sourcePath,
      String parentName)
   {
      IDfSession mySession = null;
      try
         {

// Instantiate a session using the session manager provided
            mySession = sessionManager.getSession(repositoryName);

// Instantiate a new empty document object. The DM_DOCUMENT
// variable is a static variable set at the end of this class
// definition.
            IDfSysObject newDoc =
               (IDfSysObject) mySession.newObject(DM_DOCUMENT);

// Populate the object based on the values provided.
            newDoc.setObjectName(documentName);
            newDoc.setContentType(documentType);
            newDoc.setFile(sourcePath);
            newDoc.link(parentName);

// Save the document object.
            newDoc.save();
            return true;
         }
      catch (Exception ex)
         {
            ex.printStackTrace();
            return false;
         }
```

```
      finally
         {
// Always, always release the session when you're finished.
            sessionManager.release(mySession);
         }
   }
   public static final String DM_DOCUMENT = "dm_document";
}
```

Creating a folder object

Using DFC, you can create a folder object by instantiating a new folder object, then setting its name and parent.

Example 15-2. The TutorialMakeFolder class

```
package com.emc.tutorial;

import com.documentum.fc.client.IDfFolder;
import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;

public class TutorialMakeFolder
{
   public TutorialMakeFolder()
   {
   }

   public Boolean makeFolder(
      IDfSessionManager sessionManager,
      String repositoryName,
      String folderName,
      String parentName
   )
   {
      IDfSession mySession = null;
      try
         {
// Use the session manager provided to get a new session from
// the repository.
            mySession = sessionManager.getSession(repositoryName);

// Instantiate a new folder object.
            IDfSysObject newFolder =
               (IDfFolder) mySession.newObject(DM_FOLDER);

// Try to instantiate a folder based on the name and parent.
            IDfFolder aFolder =
               mySession.getFolderByPath(parentName + "/" + folderName);

// If the folder doesn't already exist, set its name and parent,
// then save the folder.
            if (aFolder == null)
               {
                  newFolder.setObjectName(folderName);
                  newFolder.link(parentName);
                  newFolder.save();
                  return true;
               }
```

```
// Otherwise, there's nothing to do.
            else
               {
                  return false;
               }
         }
      catch (Exception ex)
         {
            ex.printStackTrace();
            return false;
         }
      finally
         {

// Always, always release the session when you're finished.
            sessionManager.release(mySession);
         }
   }
   public static final String DM_FOLDER = "dm_folder";
}
```

Setting attributes on an object

You can set attributes on an object directly by type. Most often, you will have a specific control that will set a specific data type. Alternatively, this example queries for the data type of the attribute name the user supplies, then uses a switch statement to set the value accordingly.

Example 15-3. The TutorialSetAttributeByName class

```
package com.emc.tutorial;

import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;
import com.documentum.fc.client.IDfType;
import com.documentum.fc.common.DfId;
import com.documentum.fc.common.DfTime;
import com.documentum.fc.common.IDfId;
import com.documentum.fc.common.IDfTime;

public class TutorialSetAttributeByName
{
   public TutorialSetAttributeByName()
   {
   }

   public String setAttributeByName(
      IDfSessionManager sessionManager,
      String repositoryName,
      String objectIdString,
      String attributeName,
      String attributeValue)
   {
      IDfSession mySession = null;
      try
         {

// Instantiate a session using the sessionManager provided
```

```
            mySession = sessionManager.getSession(repositoryName);

// Instantiate an ID object based on the ID string.
            IDfId idObj =
               mySession.getIdByQualification(
                  "dm_sysobject where r_object_id='" + objectIdString + "'"
               );

// Instantiate the system object using the ID object.
            IDfSysObject sysObj = (IDfSysObject) mySession.getObject(idObj);

// Query the object to get the correct data type for the attribute.
            int attributeDatatype = sysObj.getAttrDataType(attributeName);
            StringBuffer results = new StringBuffer("");

// Capture the current value.
            results.append("Previous value: " +
                           sysObj.getValue(attributeName).toString());


// Use a switch statement to set the value using the correct
// data type.
            switch (attributeDatatype)
               {
                  case IDfType.DF_BOOLEAN:
                     if (attributeValue.equals("F") |
                         attributeValue.equals("f") |
                         attributeValue.equals("0") |
                         attributeValue.equals("false") |
                         attributeValue.equals("FALSE"))
                        sysObj.setBoolean(attributeName, false);
                     if (attributeValue.equals("T") |
                         attributeValue.equals("t") |
                         attributeValue.equals("1") |
                         attributeValue.equals("true") |
                         attributeValue.equals("TRUE"))
                        sysObj.setBoolean(attributeName, true);
                     break;

                  case IDfType.DF_INTEGER:
                     sysObj.setInt(attributeName,
                                   Integer.parseInt(attributeValue));
                     break;

                  case IDfType.DF_STRING:
                     sysObj.setString(attributeName, attributeValue);
                     break;

                     // This case should not arise - no user-settable IDs
                  case IDfType.DF_ID:
                     IDfId newId = new DfId(attributeValue);
                     sysObj.setId(attributeName, newId);
                     break;

                  case IDfType.DF_TIME:
                     DfTime newTime =
                        new DfTime(attributeValue, IDfTime.DF_TIME_PATTERN2);
                     sysObj.setTime(attributeName, newTime);
                     break;

                  case IDfType.DF_UNDEFINED:
                     sysObj.setString(attributeName, attributeValue);
                     break;

               }
```

```
// Use the fetch() method to verify that the object has not been
// modified.
            if (sysObj.fetch(null))
               {
                  results = new StringBuffer("Object is no longer current.");
               }
            else
               {

// If the object is current, save the object with the new attribute
// value.
                  sysObj.save();
                  results.append("\nNew value: " + attributeValue);
               }
            return results.toString();
         }
      catch (Exception ex)
         {
            ex.printStackTrace();
            return "Set attribute command failed.";
         }
      finally
         {

// Always, always release the session.
            sessionManager.release(mySession);
         }
   }
}
```

Creating objects and setting properties in DFS

To create an object in DFS you construct a complete representation of the object locally, then pass the representation to the Object service create operation. The representation includes a PropertySet, in which you can set attributes of the repository object, using name/value pairs:

```
PropertySet properties = dataObject.getProperties();
properties.set("object_name", "MyImage");
```

Note: In the following example and other examples in this document, it is assumed that the service object (proxy) has already been instantiated and is stored as an instance variable. For a more linear example that uses a local variable for the service object, see Querying the repository in DFS, page 240.

Working with properties this way, you deal more directly with the Content Server metadata model than working with encapsulated data in DFC classes that represent repository types.

```
public DataPackage createWithContentDefaultContext(String filePath)
    throws ServiceException
{
    File testFile = new File(filePath);

    if (!testFile.exists())
    {
        throw new RuntimeException("Test file: " +
                                   testFile.toString() +
                                   " does not exist");
```

```
    }

    ObjectIdentity objIdentity = new ObjectIdentity(defaultRepositoryName);
    DataObject dataObject = new DataObject(objIdentity, "dm_document");
    PropertySet properties = dataObject.getProperties();
    properties.set("object_name", "MyImage");
    properties.set("title", "MyImage");
    properties.set("a_content_type", "gif");
    dataObject.getContents().add(new FileContent(testFile.getAbsolutePath(),
                              "gif"));

    OperationOptions operationOptions = null;
    return objectService.create(new DataPackage(dataObject),
                                operationOptions);
}
```

You can also create relationship between objects (such as the relationship between an object and a containing folder or cabinet, or virtual document relationships), so that you actually pass in a data graph to the operation, which determines how to handle the data based on whether the objects already exist in the repository. For example, the following creates a new (contentless) document and links it to an existing folder.

```
public DataObject createAndLinkToFolder(String folderPath)
{
    // create a contentless document to link into folder
    String objectName = "linkedDocument" +
                         System.currentTimeMillis();
    String repositoryName = defaultRepositoryName;
    ObjectIdentity sampleObjId =
                               new ObjectIdentity(repositoryName);
    DataObject sampleDataObject =
                                new DataObject(sampleObjId, "dm_document");
    sampleDataObject.getProperties().set("object_name", objectName);

    // add the folder to link to as a ReferenceRelationship
    ObjectPath objectPath = new ObjectPath(folderPath);
    ObjectIdentity<ObjectPath> sampleFolderIdentity =
        new ObjectIdentity<ObjectPath>(objectPath, defaultRepositoryName);
    ReferenceRelationship sampleFolderRelationship =
                                  new ReferenceRelationship();
    sampleFolderRelationship.setName(Relationship.RELATIONSHIP_FOLDER);
    sampleFolderRelationship.setTarget(sampleFolderIdentity);
    sampleFolderRelationship.setTargetRole(Relationship.ROLE_PARENT);
    sampleDataObject.getRelationships().add(sampleFolderRelationship);

    // create a new document linked into parent folder
    try
    {
        OperationOptions operationOptions = null;
        DataPackage dataPackage = new DataPackage(sampleDataObject);
        objectService.create(dataPackage, operationOptions);
    }
    catch (ServiceException e)
    {
        throw new RuntimeException(e);
    }

    return sampleDataObject;
}
```

Versioning

This section compares techniques for checkin and checkout of object in DFC and DFS.

DFC: Checkout and Checkin operations

When working with metadata, it is often most convenient to work directly with the document using DFC methods. When working with document content and files as a whole, it is usually most convenient to manipulate the documents using standard operation interfaces. Two of the most common are the Checkout and Checkin operations.

The Checkout operation

The execute method of an IDfCheckoutOperation object checks out the documents defined for the operation. The checkout operation:

- Locks the documents
- Copies the documents to your local disk
- Always creates registry entries to enable DFC to manage the files it creates on the file system

Example 15-4. TutorialCheckout.java

```
package com.emc.tutorial;

import com.documentum.com.DfClientX;
import com.documentum.com.IDfClientX;
import com.documentum.fc.client.IDfDocument;
import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;
import com.documentum.fc.common.DfId;
import com.documentum.fc.common.IDfId;
import com.documentum.operations.IDfCheckoutNode;
import com.documentum.operations.IDfCheckoutOperation;

public class TutorialCheckOut
{
   public TutorialCheckOut()
   {
   }

   public String checkoutExample
   (
      IDfSessionManager sessionManager,
      String repositoryName,
      String docId
   )
   {
      StringBuffer result = new StringBuffer("");
      IDfSession mySession = null;

      try
         {
```

```
            // Instantiate a session using the session
                                    manager provided.
            mySession = sessionManager.getSession(repositoryName);

            // Get the object ID based on the object ID string.
            IDfId idObj =
               mySession.getIdByQualification(
                  "dm_sysobject where r_object_id='" + docId + "'"
               );

            // Instantiate an object from the ID.
            IDfSysObject sysObj = (IDfSysObject) mySession.
                                        getObject(idObj);

            // Instantiate a client.
            IDfClientX clientx = new DfClientX();

            // Use the factory method to create a checkout
            // operation object.
            IDfCheckoutOperation coOp = clientx.getCheckoutOperation();

            // Set the location where the local copy of the
            //   checked out file is stored.
            coOp.setDestinationDirectory("C:\\");

            // Get the document instance using the document ID.
            IDfDocument doc =
               (IDfDocument) mySession.getObject(new DfId(docId));

            // Create the checkout node by adding the document to
            //  the checkout operation.
            IDfCheckoutNode coNode = (IDfCheckoutNode) coOp.add(doc);

            // Verify that the node exists.
            if (coNode == null)
               {
                  result.append("coNode is null");
               }

            // Execute the checkout operation. Return the result.
            if (coOp.execute())
               {
                  result.append("Successfully checked out file ID:
                                " + docId);
               }
            else
               {
                  result.append("Checkout failed.");
               }
            return result.toString();
         }
      catch (Exception ex)
         {
            ex.printStackTrace();
            return "Exception hs been thrown: " + ex;
         }
      finally
      {
         sessionManager.release(mySession);
      }
   }
}
```

Special considerations for checkout operations

If any node corresponds to a document that is already checked out, the system does not check it out again. DFC does not treat this as an error. If you cancel the checkout, however, DFC cancels the checkout of the previously checked out node as well.

DFC applies XML processing to XML documents. If necessary, it modifies the resulting files to ensure that it has enough information to check in the documents properly.

The Checkin operation

The execute method of an IDfCheckinOperation object checks documents into the repository. It creates new objects as required, transfers the content to the repository, and removes local files if appropriate. It checks in existing objects that any of the nodes refer to (for example, through XML links).

Check in a document as the next major version (for example, version 1.2 would become version 2.0). The default increment is NEXT_MINOR (for example, version 1.2 would become version 1.3).

Example 15-5. The TutorialCheckIn class

```
package com.emc.tutorial;

import com.documentum.com.DfClientX;
import com.documentum.com.IDfClientX;
import com.documentum.fc.client.IDfDocument;
import com.documentum.fc.client.IDfSession;
import com.documentum.fc.client.IDfSessionManager;
import com.documentum.fc.client.IDfSysObject;
import com.documentum.fc.common.DfId;
import com.documentum.fc.common.IDfId;
import com.documentum.operations.IDfCheckinNode;
import com.documentum.operations.IDfCheckinOperation;

public class TutorialCheckIn
{
   public TutorialCheckIn()
   {
   }

   public String checkinExample(
      IDfSessionManager sessionManager,
      String repositoryName,
      String docId
   )
   {
      IDfSession mySession = null;
      try
         {
            mySession = sessionManager.getSession(repositoryName);

// Get the object ID based on the object ID string.
            IDfId idObj =
               mySession.getIdByQualification(
                  "dm_sysobject where r_object_id='" + docId + "'"
               );

// Instantiate an object from the ID.
```

```
            IDfSysObject sysObj = (IDfSysObject) mySession.getObject(
                                                 idObj);

// Instantiate a client.
            IDfClientX clientx = new DfClientX();

// Use the factory method to create an IDfCheckinOperation instance.
            IDfCheckinOperation cio = clientx.getCheckinOperation();

// Set the version increment. In this case, the next major version
// ( version + 1)
            cio.setCheckinVersion(IDfCheckinOperation.NEXT_MAJOR);

// When updating to the next major version, you need to explicitly
// set the version label for the new object to "CURRENT".
            cio.setVersionLabels("CURRENT");

// Create a document object that represents the document being
// checked in.
            IDfDocument doc =
               (IDfDocument) mySession.getObject(new DfId(docId));

// Create a checkin node, adding it to the checkin operation.
            IDfCheckinNode node = (IDfCheckinNode) cio.add(doc);

// Execute the checkin operation and return the result.
            if (!cio.execute())
               {
                  return "Checkin failed.";
               }

// After the item is created, you can get it immediately using the
// getNewObjectId method.

            IDfId newId = node.getNewObjectId();
            return "Checkin succeeded -  new object ID is: " + newId;
         }
      catch (Exception ex)
         {
            ex.printStackTrace();
            return "Checkin failed.";
         }
      finally
      {
         sessionManager.release(mySession);
      }
   }
}
```

Special considerations for checkin operations

The following are considerations when you are creating a custom checkin operation.

Setting up the operation

To check in a document, you pass an object of type IDfSysObject or IDfVirtualDocument, *not the file on the local file system*, to the operation's add method. In the local client file registry, DFC records the path and filename of the local file that represents the content of an object. If you move or rename the file, DFC loses track of it and reports an error when you try to check it in.

Setting the content file, as in IDfCheckinNode.setFilePath, overrides DFC's saved information.

If you specify a document that is not checked out, DFC does not check it in. DFC does not treat this as an error.

You can specify checkin version, symbolic label, or alternate content file, and you can direct DFC to preserve the local file.

If between checkout and checkin you remove a link between documents, DFC adds the orphaned document to the checkin operation as a root node, but the relationship between the documents no longer exists in the repository.

Processing the checked in documents

Executing a checkin operation normally results in the creation of new objects in the repository. If opCheckin is the IDfCheckinOperation object, you can obtain a complete list of the new objects by calling

```
IDfList list = opCheckin.getNewObjects();
```

The list contains the object IDs of the newly created SysObjects.

In addition, the IDfCheckinNode objects associated with the operation are still available after you execute the operation. You can use their methods to find out many other facts about the new SysObjects associated with those nodes.

DFS: VersionControl service

Checkin and checkout of objects, and related functions, are managed in DFS using the VersionControl service. The checkout operation, for example, checks out all of the objects identified in an ObjectIdentitySet and on success returns a DataPackage containing representations of the checked out objects.

Note: In this example and other examples in this document, it is assumed that the service object (proxy) has already been instantiated and is stored as an instance variable. For a more linear example that uses a local variable for the service object, see Querying the repository in DFS, page 240.

```
public DataPackage checkout(ObjectIdentity objIdentity)
     throws ServiceException
{
    ObjectIdentitySet objIdSet = new ObjectIdentitySet();
    objIdSet.getIdentities().add(objIdentity);

    OperationOptions operationOptions = null;
    DataPackage resultDp;
    try
    {
```

```
        resultDp = versionControlService.checkout(objIdSet,
                                        operationOptions);
    }
    catch (Exception e)
    {
        e.printStackTrace();
        throw new RuntimeException(e);
    }
    System.out.println("Checkout successful");

    List<VersionInfo> vInfo = versionControlService.getVersionInfo
                              (objIdSet);
    VersionInfo versionInfo = vInfo.get(0);

    System.out.println("Printing version info for " + versionInfo.
                        getIdentity());
    System.out.println("isCurrent is " + versionInfo.isCurrent());
    System.out.println("Version is " + versionInfo.getVersion());

    System.out.println("Symbolic labels are: ");
    for (String label : versionInfo.getSymbolicLabels())
    {
        System.out.println(label);
    }

    versionControlService.cancelCheckout(objIdSet);
    System.out.println("Checkout cancelled");
    return resultDp;
}
```

The DataPackage can contain content (as controlled by a ContentProfile in OperationOptions or in the service context). The following specifies in OperationOptions that content of any format will be transferred to the client.

```
ContentProfile contentProfile = new ContentProfile();
contentProfile.setFormatFilter(FormatFilter.ANY);
OperationOptions operationOptions = new OperationOptions();
operationOptions.setContentProfile(contentProfile);
operationOptions.setProfile(contentProfile);
```

Querying the repository

This section compares techniques for querying the repository in DFC and DFS.

Querying the repository in DFC

Creating and executing a query in DFC is a straightforward paradigm. You instantiate a blank query object, set its DQL arguments, then execute the query and capture the results in a collection object.

Example 15-6. Abstract example of executing a query in DFC

```
public class OwnerNameQuery
{
  private IDfCollection getIdFromOwnerName(
      IDfSessionManager sessionManager,
      String repositoryName,
```

```
      String ownerName
   )
   {
      IDfSession mySession = null;
      try
         {
            mySession = sessionManager.getSession(repositoryName);
            IDfQuery query = new DfClientX().getQuery();
            query.setDQL("select r_object_id from dm_document " +
              "where owner_name=" + ownerName);
            IDfCollection co = query.execute(session,
                              IDfQuery.DF_READ_QUERY );
    return co;
  }
}
```

Querying the repository in DFS

In DFS you can use the Query service to execute either a passthrough query using a DQL string literal, or a structured query. The following class shows how to do this in linear fashion, so that you can see the complete sequence events involved in setting up the service context, instantiating the service, setting up the objects that represent the query, and invoking the service.

```
package com.emc.documentum.fs.doc.samples.client;

import java.util.List;

import com.emc.documentum.fs.datamodel.core.CacheStrategyType;
import com.emc.documentum.fs.datamodel.core.DataObject;
import com.emc.documentum.fs.datamodel.core.DataPackage;
import com.emc.documentum.fs.datamodel.core.OperationOptions;
import com.emc.documentum.fs.datamodel.core.content.ContentTransferMode;
import com.emc.documentum.fs.datamodel.core.context.RepositoryIdentity;
import com.emc.documentum.fs.datamodel.core.profiles.ContentTransferProfile;
import com.emc.documentum.fs.datamodel.core.properties.PropertySet;
import com.emc.documentum.fs.datamodel.core.query.PassthroughQuery;
import com.emc.documentum.fs.datamodel.core.query.QueryExecution;
import com.emc.documentum.fs.datamodel.core.query.QueryResult;
import com.emc.documentum.fs.rt.ServiceException;
import com.emc.documentum.fs.rt.context.ContextFactory;
import com.emc.documentum.fs.rt.context.IServiceContext;
import com.emc.documentum.fs.rt.context.ServiceFactory;
import com.emc.documentum.fs.services.core.client.IQueryService;

/**
 * This class demonstrates how to code a typical request to a DFS core service
 * (in this case QueryService). The code goes through the steps of creating a
 * ServiceContext, which contains repository and credential information, and
 * calling the service with the profile.
 *
 * This sample assumes that you have a working installation
 * of DFS that points to a working Content Server.
 *
 */
public class QueryServiceTest
{
    /*********************************************************
    * You must supply valid values for the following fields: */

    /* The repository that you want to run the query on */
```

```
private String repository =  "techpubs";

/* The username to login to the repository */
private String userName =  "dmadmin";

/* The password for the username */
private String password =  "D3v3l0p3r";

/* The address where the DFS services are located */
private String host = "http://127.0.0.1:8888/services";

/*********************************************************/
/* The module name for the DFS core services */
private static String moduleName = "core";
private IServiceContext serviceContext;

public QueryServiceTest()
{
}

public void setContext()
{
    ContextFactory contextFactory = ContextFactory.getInstance();
    serviceContext = contextFactory.newContext();
    RepositoryIdentity repoId = new RepositoryIdentity();
    repoId.setRepositoryName(repository);
    repoId.setUserName(userName);
    repoId.setPassword(password);
    serviceContext.addIdentity(repoId);
}

public void callQueryService()
{
    try
    {
        ServiceFactory serviceFactory = ServiceFactory.getInstance();
        IQueryService querySvc =
         serviceFactory.getRemoteService(IQueryService.class,
                                         serviceContext,
                                    moduleName,
                                         host);

        PassthroughQuery query = new PassthroughQuery();
        query.setQueryString("select r_object_id, "
                           + "object_name from dm_cabinet");
        query.addRepository(repository);
        QueryExecution queryEx = new QueryExecution();
        queryEx.setCacheStrategyType(CacheStrategyType.
                                    DEFAULT_CACHE_STRATEGY);

       OperationOptions operationOptions = null;

        // this is where data gets sent over the wire
        QueryResult queryResult = querySvc.execute(query,
                                                   queryEx,
                                                   operationOptions);
        System.out.println("QueryId == " + query.getQueryString());
        System.out.println("CacheStrategyType ==
                           " + queryEx.getCacheStrategyType());
        DataPackage resultDp = queryResult.getDataPackage();
        List<DataObject> dataObjects = resultDp.getDataObjects();
        System.out.println("Total objects returned is: "
                           + dataObjects.size());
        for (DataObject dObj : dataObjects)
        {
```

```
                PropertySet docProperties = dObj.getProperties();
                String objectId = dObj.getIdentity().getValueAsString();
                String docName = docProperties.get("object_name")
                                                .getValueAsString();
                System.out.println("Document " + objectId + " name is "
                                                        + docName);
            }
          }
        catch (ServiceException e)
          {
           e.printStackTrace();
          }
    }

    public static void main(String[] args)
    {
        QueryServiceTest t = new QueryServiceTest();
        t.setContext();
        t.callQueryService();
    }
}
```

Starting a workflow

DFC provides a rich interface into Workflow functionality. DFS as of release 6 SP1 has a much more limited interface which supports fetching information about workflow templates and metadata and starting a workflow.

Starting a workflow in DFC

For detailed information about the workflow interface in DFC, refer to the DFC Javadocs, which contain inline sample code to illustrate many of the workflow-related methods. The following illustrates use of the IDfWorkflow.execute method to start a workflow.

```
// Setup all params for sendToDistributionList() here...
 IDfId wfId = sess.sendToDistributionList(userList,
                                          groupList,
                                          "Please review",
                                          objList,
                                          5,
                                          false);
 IDfWorkflow wfObj = (IDfWorkflow)sess.getObject(wfId);
 IDfWorkflow wfObj2 = (IDfWorkflow)sess.newObject("dm_workflow");
 wfObj2.setProcessId(wfObj.getProcessId());
 wfObj2.setSupervisorName("someUser")
 wfObj2.save();
 wfObj2.execute();
```

Starting a workflow using the DFS Workflow service

To start a workflow you can use the Workflow service operations. There are some dependencies among these operations, the general procedure is:

1. Use the getProcessTemplates to get a DataPackage populated with DataObject instances that represent business process templates.
2. From this DataPackage, extract the identity of a process that you want to start.
3. Pass this ObjectIdentity to the getProcessInfo operation to get an object representing business process metadata.
4. Modify the ProcessInfo data as required and pass it to the startProcess operation to start the process.

The following sample starts at step 3, with the processId obtained from the data returned by getProcessTemplates.

Note: In the following example and other examples in this document, it is assumed that the service object (proxy) has already been instantiated and is stored as an instance variable. For a more linear example that uses a local variable for the service object, see Querying the repository in DFS, page 240.

```
public void startProcess(String processId,
                         String processName,
                         String supervisor,
                         ObjectId wfAttachment,
                         List<ObjectId> docIds,
                         String noteText,
                         String userName,
                         String groupName,
                         String queueName) throws Exception
{

    // get the template ProcessInfo
    ObjectId objId = new ObjectId(processId);
    ProcessInfo info = workflowService
        .getProcessInfo(new ObjectIdentity<ObjectId>(objId,
                        defaultRepositoryName));

    // set specific info for this workflow
    info.setSupervisor(supervisor);
    info.setProcessInstanceName(processName + new Date());

    // workflow attachment
    info.addWorkflowAttachment("dm_sysobject", wfAttachment);

    // packages
    List<ProcessPackageInfo> pkgList = info.getPackages();
    for (ProcessPackageInfo pkg : pkgList)
    {
        pkg.addDocuments(docIds);
        pkg.addNote("note for " + pkg.getPackageName() + " "
                    + noteText, true);
    }

    // alias
    if (info.isAliasAssignmentRequired())
    {
        List<ProcessAliasAssignmentInfo> aliasList
            = info.getAliasAssignments();
        for (ProcessAliasAssignmentInfo aliasInfo : aliasList)
```

```
        {
            String aliasName = aliasInfo.getAliasName();
            String aliasDescription = aliasInfo.getAliasDescription();
            int category = aliasInfo.getAliasCategory();
            if (category == 1) // User
            {
                aliasInfo.setAliasValue(userName);
            }
            else if (category == 2 || category == 3) // group,
                     user or group
            {
                aliasInfo.setAliasValue(groupName);
            }

            System.out.println("Set alias: "
                               + aliasName
                               + ", description: "
                               + aliasDescription
                               +  ", category: "
                               + category
                               + " to "
                               + aliasInfo.getAliasValue());
        }
    }

    // Performer.
    if (info.isPerformerAssignmentRequired())
    {
        List<ProcessPerformerAssignmentInfo> perfList
            = info.getPerformerAssignments();
        for (ProcessPerformerAssignmentInfo perfInfo : perfList)
        {
            int category = perfInfo.getCategory();
            int perfType = perfInfo.getPerformerType();
            String name = "";
            List<String> nameList = new ArrayList<String>();
            if (category == 0) // User
            {
                name = userName;
            }
            else if (category == 1 || category == 2) //
                     Group, user or group
            {
                name = groupName;
            }
            else if (category == 4)     // work queue
            {
                name = queueName;
            }
            nameList.add(name);
            perfInfo.setPerformers(nameList);

           System.out.println("Set performer perfType: " + perfType +
                              ", category: " + category + " to " + name);
        }
    }

  ObjectIdentity wf = workflowService.startProcess(info);
  System.out.println("started workflow: " + wf.getValueAsString());
```

```
}
```

Index

T

U

V

W

X