Leveraging Web Services to Integrate with EMC Documentum in an SOA

Abstract

This white paper examines features of the EMC® Documentum® 6 platform that enable its industry-leading enterprise content management capabilities to participate in a service oriented architecture (SOA) and communicate with other systems via web services. The primary enabling feature, EMC Documentum Foundation Services (DFS), is examined in technical detail, and sample code fragments that illustrate how to use DFS are provided. The EMC Documentum Process Suite is also examined as a prototypical orchestrator of SOA services. After reading this paper, you should have a good understanding of how to access Documentum services in an SOA using web services.

February 2008
# Table of Contents

Executive summary ................................................................................................................................. 4
Introduction ............................................................................................................................................... 4
    Audience ................................................................................................................................................ 4
Web services and SOA ................................................................................................................................. 4
    SOA ......................................................................................................................................................... 5
    SOA and business process management ................................................................................................ 5
    Web services .......................................................................................................................................... 5
    Web services in an SOA .......................................................................................................................... 6
Documentum Foundation Services ............................................................................................................ 7
    Object service ......................................................................................................................................... 7
    Version control service .......................................................................................................................... 8
    Schema service ...................................................................................................................................... 8
    Query service ......................................................................................................................................... 8
    Search service ....................................................................................................................................... 8
    Workflow service .................................................................................................................................. 9
Using DFS services ................................................................................................................................. 9
    SOAP ...................................................................................................................................................... 9
    Proxies ................................................................................................................................................... 9
    Client library ........................................................................................................................................ 9
    Orchestration tools ............................................................................................................................ 10
The DFS data model .................................................................................................................................. 10
    DataPackage ........................................................................................................................................ 10
    DataObject .......................................................................................................................................... 10
    Content ............................................................................................................................................... 10
    ObjectIdentity ..................................................................................................................................... 11
    Property .............................................................................................................................................. 11
    Permissions ......................................................................................................................................... 11
    Relationships ...................................................................................................................................... 11
DFS example using the Java Client Library .............................................................................................. 12
    Step 1: Create and populate the ServiceContext .................................................................................. 12
    Step 2: Instantiate a DFS Service ........................................................................................................ 12
    Step 3: Query the repository for objects .............................................................................................. 13
    Step 4: Examine the results .................................................................................................................. 13
BPM and SOA: A DFS example using Process Builder ............................................................................ 13
    Process data ......................................................................................................................................... 14
    Process template .................................................................................................................................. 14
    Data mapping ....................................................................................................................................... 16
    Defining business logic ....................................................................................................................... 19
Conclusion ................................................................................................................................................. 21
Appendix: DFS Java client sample code .................................................................................................. 21
Executive summary

COM. EJB. .NET. Enterprise Application Integration. CORBA. Message Oriented Middleware. Enterprise Service Bus. Besides having been elevated (or is it lowered?) to buzzword status, what do these terms have in common? At one time or another, each seemingly held the promise of an enterprise utopia in which business applications could openly and simply communicate with one another. However, as back-end system implementation technologies grew more diverse and the sheer quantities of information expanded in exponential leaps and bounds, each of these technologies has taken its turn of falling short of the lofty goal of enterprise interoperability.

As business applications have become ever more componentized and distributed, new architectures and technologies have arisen that are less proprietary and more flexible than their brethren of yore. One of these newer architecture-technology pairings is the services oriented architecture (SOA)-web services duet. While SOA does not necessarily imply web services and vice-versa, the two terms are often used in tandem, with web services providing the platform- and location-agnostic messaging technology and SOA governing the topology and providing guidelines for service behaviors.

Whether the combination of SOA and web services can deliver on the vision of open interoperability remains to be seen. One thing that is certain, however, is that SOA and web services are experiencing a heyday of immense proportion. Over the last few years, web services have become practically ubiquitous. From personal websites to enterprise portals to mammoth-scale back-end systems, everything seems to be declaring web services the transport and messaging mechanism of choice. Similarly, there is great momentum behind the push toward service orientation, and most major enterprise software vendors now tout SOA-friendly services.

As evidenced by the Service Based Object (SBO) support in the Business Object Framework (BOF), EMC® Documentum® has long been oriented toward delivering content services. With the introduction of Documentum Foundation Services (DFS), however, the Documentum platform has taken a big step forward in realizing the vision of service-oriented enterprise content management. Introduced in Documentum 6, DFS provides a flexible, robust, and complete set of enterprise content management (ECM) services that are readily consumable in an SOA. Accessible as web services, these enterprise content services (ECS) can be easily utilized by applications across an enterprise, thereby providing interoperability with the core functionality of the Documentum platform. In the sections that follow, we’ll delve into how DFS works and how you can leverage it to integrate with Documentum in an enterprise SOA environment.

Introduction

This white paper examines features of the EMC Documentum 6 platform that enable its industry-leading enterprise content management capabilities to participate in an SOA and communicate with other systems via web services. The primary enabling feature, EMC Documentum Foundation Services (DFS), is examined in technical detail, and sample code fragments that illustrate how to use DFS are provided. The EMC Documentum Process Suite is also examined as a prototypical orchestrator of SOA services. After reading this paper, you should have a good understanding of how to access Documentum services in an SOA using web services.

Audience

This white paper is intended for EMC Documentum customers, developers, system engineers, partners and anyone who is interested in learning more about how to leverage EMC Documentum Foundation Services in a service-oriented environment.

Web services and SOA

The terms “web services” and “SOA” are often used interchangeably even though they each belong to a different concept. Before getting into the details of DFS and how it can be used to communicate with
Documentum, it is important to have a solid understanding of what an SOA is, what web services are, and how the two can be interrelated.

**SOA**

At a basic level, an SOA is simply an architectural paradigm that:
- Promotes highly componentized packaging of software functionality into modules known as services
- Groups software functions along the lines of discrete business processes
- Publishes these business process-scoped services so that they can be readily consumed by applications participating in the SOA

In an SOA environment, elaborate business applications can be efficiently synthesized by chaining together these published services. This method for building up complex business applications yields a number of benefits to the enterprise:
- Development effort is reduced, as each business process need only be implemented once. Additionally, SOA frameworks can be leveraged that enable a developer to easily chain multiple services together with little if any additional coding.
- Applications can be rapidly developed because software designers and developers do not have to learn the specifics of lower-level proprietary technologies. Instead, they need only be proficient with the technology used to communicate within the SOA (for example, web services).
- Behavioral and semantic consistency is ensured, as applications all leverage the same set of service implementations. Similarly, if a service’s behavior needs to change, the change need only be implemented (and tested) in one location.
- By definition, SOA promotes development of loosely coupled systems, which inherently improves maintainability.

SOA is often compared to monolithic architectures that internally encapsulate business process logic and also to remote procedure architectures in which applications communicate with one another through direct API calls. It should be evident that neither of these alternative architectures is as flexible or maintainable as an SOA.

**SOA and business process management**

Because of the flexibility that SOA provides applications, it is highly complementary to business process management (BPM) technologies. BPM is focused on orchestrating, automating, and optimizing processes so that organizations can achieve new levels of efficiency, productivity, and agility in their operations. BPM allows processes to be defined using reusable components that encapsulate the various human and systems-based activities that make up business processes.

These components are connected through a process model, which also includes business rules that define process flow and behavior based on the runtime context of each process instance. BPM typically leverages SOA to allow business processes to interact with systems, applications, and data sources, which are spread across - and may even exist outside - the enterprise. This allows for rapid configuration and deployment of processes, as well as providing ease in maintaining and refining how the process interacts with disparate information systems over time. Best of all, BPM systems typically provide graphical tools for modeling and implementing business process. This enables systems-based interactions to be easily configured from a services library without requiring any programming expertise. An example of using BPM in an SOA environment is provided later in this paper.

**Web services**

Whereas SOA is an architectural concept, web services is a technology. More precisely, web services is a category of technologies. According to the W3C consortium that defines many web specifications, a web service is simply “a programmatic interface that is used for application to application communication over
Leveraging Web Services to Integrate with EMC Documentum in an SOA

More specifically, a web service is a software system that follows a set of standards to provide interoperable functionality across a variety of platforms. Although the term “web services” technically refers to a category of technologies, in common use the term typically implies a stack of standard protocols that are used for system-to-system communication. These protocols include:

- A transport protocol defines the method for transporting messages across the network. HTTP is arguably the most commonly used transport protocol today. Other transport protocols include FTP and SMTP.
- A messaging protocol defines the encoding used to describe messages in a common XML format. SOAP is the usual messaging protocol, but others such as XML-RPC and REST also exist.
- A description protocol describes the public interface to a web service. WSDL is typically used for this.
- A discovery protocol provides a means for enabling applications to discover the service on the network. UDDI is commonly used for service discovery.

Figure 1. Typical use of web services protocols

The transport, messaging, and description protocols are necessary for all web services; they form the core set of standards that define the rules for interacting with the web service. Discovery protocols are a bit different. Whereas the other protocols are critical to the definition and runtime utilization of all web services, discovery protocols are used to publish web services that need to be readily found by consumers that might not otherwise know how to access the services. Discovery protocols are especially important in an SOA.

It is important to note that using web services does not necessarily imply compliance to SOA. Indeed, because web services are easy to use for cross-platform interoperability, they are quite frequently used as a common language for purpose-built integrations that cross technology boundaries (for example, a .NET system communicating with a J2EE system). For a web service to effectively participate in an SOA, it must be designed with that purpose in mind.

**Web services in an SOA**

Because of their immense popularity (which, no doubt, stems largely from their composition of established, well-understood technologies such as HTTP and XML), web services are often used as an enabling technology in SOA environments. Web services are innately loosely coupled, which is a principal design

1 [http://www.w3.org/2002/ws/](http://www.w3.org/2002/ws/)
requirement for an SOA. Outside of this, however, web services need to be carefully architected to be effective participants in an SOA. Some key design principles include:

- **Loose coupling.** As previously mentioned, web services are inherently loosely coupled.

- **Appropriate service granularity.** A service typically encapsulates a number of lower-level operations. How many operations should be combined into a single service? While there are no hard and fast rules, a good guideline is that a single service should be able to support a set of interrelated business processes. For example, the DFS ObjectService provides a set of operations that enable clients to perform object lifecycle operations (such as create, move, delete, and so on) on repository objects.

- **Appropriate operation granularity.** The amount of functionality performed by any operation in a given service needs to be carefully chosen to maximize the usefulness of the service in an SOA. An operation that does not contain enough functionality is preferred rather than exposing a low-level API in that it does not promote consistent behavior, and an operation that contains too much functionality will be too specialized, therefore reducing its ability to be utilized and reused by many consumers.

- **Implementation hiding.** Each service and operation should hide implementation details from its interface (WSDL, for web services). This further increases decoupling and enhances flexibility.

Web services that are designed to these guidelines will prove most effective in an SOA.

**Documentum Foundation Services**

Documentum Foundation Services (DFS) comprises the core of the Documentum 6 Enterprise Content Services (ECS), an SOA-compliant services offering. Fundamentally, DFS is a set of out-of-the-box tools and services designed from the ground up to enable Documentum core functionality to be leveraged in an SOA. DFS exposes the Documentum industry-leading content management functionality as standards-compliant web services.

The following sections provide an overview of DFS services.

**Object service**

The object service provides a set of basic operations on repository objects. These services are responsible for all typical persistent operations on an object such as creating, getting, updating, and deleting repository objects. Additionally, all object services can operate on multiple objects enabling clients to optimize service usage by minimizing the number of service interactions. For example, an object service can create users and associate them with groups in a single transaction.

The object service operations include:

- **create** creates an object (or objects) in the repository and optionally links into a specific location.
- **createPath** creates a folder structure in the repository.
- **get** retrieves the specified version of a set of objects from the repository.
- **update** updates properties, content, and relationships of a set of repository objects, and saves the objects. If the objects to be updated do not exist, they are automatically created.
- **delete** deletes a set of objects from the repository.
- **copy** copies a set of objects from one location to another, either within a single repository or from one repository to another. During this operation, the service can optionally make modifications to the objects being copied.
- **move** moves a set of objects from one location to another location, optionally updating the objects as they are moved.
- **validate** validates a set of objects against the repositories’ data dictionary rules, testing if they represent valid repository objects with valid repository properties.
Version control service

The version control service provides operations that enable access of and changes to specific versions of objects within the repository.

The version control service provides the following operations:

- `getCheckoutInfo` provides checkout information about a set of objects, specifically whether the objects are checked out and the name of the user who has checked them out.
- `checkout` checks out the specified version of a set of repository objects.
- `checkin` checks in a set of repository objects.
- `cancelCheckout` cancels the checkout of a set of repository objects.
- `deleteVersion` deletes a specific version of a repository object. If the deleted object is the current version, the previous version in the version tree is made the current version.
- `deleteAllVersions` deletes all versions of a repository object.
- `getCurrent` exports the current version of a repository object (including any object content).
- `getVersionInfo` provides information about the version of a repository object.

Schema service

The schema service provides a mechanism for retrieving information about repository schemas, which are the formal definitions of the repository metadata. It exposes a generalized view into the data dictionary, including information regarding types, properties, and relationships.

The schema service includes the following operations:

- `getSchemaInfo` retrieves schema information for the default schema of the specified repository.
- `getRepositoryInfo` retrieves schema information about a repository, including a list of repository schemas.
- `getTypeInfo` retrieves information about a repository type.
- `getPropertyInfo` retrieves information about a repository property.
- `getDynamicAssistValues` retrieves information about value assistance for a specified repository property.

Query service

The query service is the primary mechanism for retrieving information from a repository. The query service is general purpose and uses execution semantics quite similar to those used to query a relational database.

The query service provides one operation:

- `execute` executes a query against the repository and returns the result.

Search service

The search service provides full-text and structured search capabilities and can be used to search for content spread over multiple Documentum repositories and even external sources.

The search service provides the following operations:
• `getRepositoryList` provides a list of managed and external repositories that are available to the service for searching.

• `execute` executes a query against data in the repository and returns the result.

**Workflow service**

The workflow service includes operations for interrogating the repository for workflow processes and an operation for initiating workflow processes.

The workflow service provides the following operations:

• `getProcessTemplates` returns the list of process templates installed in the repository.

• `getProcessInfo` retrieves detailed information about a specific process template. This operation can be called before starting a workflow to return a data structure that lists all of the values required to start the workflow.

• `startProcess` initiates a workflow.

**Using DFS services**

There are a number of ways for clients to consume DFS services. A DFS service’s primary interface is its WSDL. The WSDL defines the contract for the service and provides flexibility for clients to interact with the service using a variety of mechanisms, some programmatic, and some that do not require any programming.

**SOAP**

One way to invoke a DFS service operation is to communicate directly using SOAP messages that comply to the service’s WSDL definition. Perhaps not the most developer-friendly method of invocation, this is the most open way of invoking a DFS operation and is typically the basis for tools that orchestrate DFS services without requiring programming.

**Proxies**

Proxies, also known as “client bindings,” are machine-generated code modules that encapsulate the mechanics of creating and sending SOAP requests and receiving and parsing SOAP responses. The facades created by proxies greatly simplify the invocation of WSDL-exporting services such as those in DFS.

To create a proxy, a developer typically runs a language-specific utility that digests a WSDL and outputs the proxy code. Such utilities exist for many languages including Java, .NET, and Python. Oftentimes, IDEs provide point-and-click functionality for generating proxies as well.

The steps required to use a proxy to access DFS services are usually quite simple, although the specifics depend on the client’s implementation language and the proxy generation utility that is used. Typically, the proxy provides an API that is native to the client environment that encapsulates the call to the service.

**Client library**

DFS includes a rich client library that provides enhanced support for Java- and .NET-based clients. This library provides functionality very similar to a set of Java proxies but adds a number of performance and convenience improvements. Features of the Java Client Library include:

• Convenience methods and constructors.

• Transparent handling of exceptions passed in SOAP messages returned by the service. This enables the client to display the stack trace of the exception as it would an exception thrown by an application running in the local Java virtual machine.
Leveraging Web Services to Integrate with EMC Documentum in an SOA

- Transparent invocation of UCF and handling of UCF content transfer.
- Simplified security.
- Location transparency. Available in the Java Client Library, this feature enables clients to execute the services either remotely via web services or locally within the client’s Java virtual machine.

The client libraries provide a more traditional object-oriented perspective to DFS, which can help improve productivity with veteran Documentum developers because of its familiarity to the DFC model of Documentum development.

**Orchestration tools**

Orchestration tools are applications that enable non-programmers to model complex business processes by chaining together multiple services in an SOA. The Documentum platform provides one such tool within the Documentum Process Suite: Process Builder. Process Builder provides an intuitive, interactive user interface that makes the task of creating complex business processes as easy as drawing a flow chart. An example of using Process Builder to orchestrate DFS services is shown in the section “BPM and SOA: A DFS example using Process Builder.”

**The DFS data model**

DFS employs a data model that comprises the object model for data passed to and returned from DFS services. The DFS data model is an important component of the overall DFS architecture, as its rich design helps promote simplicity across the services.

An overview of the DFS data model follows.

**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 graphs and is used to communicate with Object service operations such as create, get, and update. Object service operations process the DataObject graphs in the DataPackage sequentially.

**DataObject**

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

A DataObject instance can potentially be large and complex. Much of the work in DFS service consumers is dedicated to constructing the DataObject instances. A DataObject contains comprehensive information about the repository object that it represents including its identity, properties, content, and relationships to other repository objects. In addition, DataObject instances may contain settings that instruct the services about how the client wishes the DataObject to be processed.

DataObject instances are consistently passed to and returned by services in simple collections defined by the DataPackage class. The use of these collections permits processing of multiple DataObject instances in a single service interaction.

**Content**

The content for a DataObject is represented by the Content class and its subtypes. The Content subtypes support multiple types of input to services and multiple content transfer options. A given Content object can represent a complete document, a single page in a document, or a set of pages in a document. A
DataObject contains a list of zero or more of these Content instances, allowing representation of both primary content and content renditions.

The ContentProfile object 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.

**ObjectIdentity**

The ObjectIdentity class uniquely identifies a repository object. It contains both the repository name and a unique identifier for the object within that repository. This identifier can take one of several forms: an object ID, an object path, or a query expression that selects a single object.

**Property**

A Property object represents a property or attribute of an object. A Property object can represent both single-valued and repeating repository attributes. Each DataObject can optionally contain a set of Property objects within a PropertySet. Additionally, a Property can be optionally marked as Transient. Transient properties are custom Property objects that are not persistent properties of repository objects. This is a powerful concept that allows Property objects to be used to transfer information beyond that which is directly bound to the attributes on a repository object.

Similar to a ContentProfile, a PropertyProfile defines filters that limit the properties that are returned with an object by a service call. This allows for optimization of a service by returning only those properties that your application requires.

**Permissions**

A DataObject contains a list of Permission objects. Together, these represent the permissions available to the user defined in the service context associated with the given repository object. The Permission list can serve to provide the calling client with read access to the current user's permissions on a repository object.

The PermissionProfile enables the client to set filters that control the contents of Permission lists in DataObjects returned by service calls. By default, the platform services will return an empty Permission list. The client must explicitly request in a PermissionProfile for the permissions be returned.

**Relationships**

The Relationship class defines relationships between repository objects. Relationships allow the client to construct a single DataObject that specifies all of its relations to other objects (both existing and new) and to get, update, or create the entire set of objects and their relationships in a single service interaction. This enables scenarios such as creating a new folder and linking a set of new documents into it to be performed in a single service call.

The RelationshipProfile class enables a client optimization mechanism that provides fine control over the size and complexity of the DataObject instances returned by DFS services. As a performance optimization, the default behavior of DFS services is to return DataObjects without any Relationship instances. To alter this behavior, a client needs to provide a RelationshipProfile that specifies the types of Relationship instances to return with the DataObject.
DFS example using the Java Client Library

The following code example shows the end-to-end process of connecting and authenticating with DFS, and then retrieving a set of objects (in the form of DataPackage instances) from the repository and displaying each object’s properties.

This code queries the repository for all documents that have an object name that starts with “DFS” and then outputs their subject and creation date attribute values.

**Step 1: Create and populate the ServiceContext**

The first part of the example program creates an instance of a ServiceContext object and populates it with the name and credentials of the target repository. The ServiceContext is the client object that stores the service’s stateful information such as repository credentials. ServiceContext objects are created via a ContextFactory:

```java
// Get a ContextFactory so that we can create a ServiceContext
ContextFactory contextFactory = ContextFactory.getInstance();

// Use the ContextFactory to create a ServiceContext for a service
IServiceContext serviceContext = contextFactory.newContext();
```

Once the ServiceContext instance has been created, it can be populated with credentials that will be used to authenticate to the repository. The Java Client Library provides the RepositoryIdentity object for storing repository credentials. A ServiceContext may be associated with multiple RepositoryIdentity objects, but only one per repository.

```java
// Create a RepositoryIdentity to store the repository credentials on the ServiceContext
RepositoryIdentity repoId = new RepositoryIdentity();

// Populate the repository credentials
repoId.setRepositoryName(repositoryName);
repoId.setUserName(userName);
repoId.setPassword(userPassword);
```

Now that the RepositoryIdentity has been populated, it is added to the ServiceContext:

```java
// Add the populated RepositoryIdentity to the ServiceContext
serviceContext.addIdentity(repoId);
```

**Step 2: Instantiate a DFS Service**

Now that the ServiceContext has been populated, it can be passed to the ServiceFactory for creation of a service instance. For this example, we will instantiate an ObjectService.

```java
// Get an ObjectService from the ServiceFactory
IObjectService objectService = ServiceFactory.getInstance().getRemoteService(IObjectService.class,                     // Class of the service type to instantiate
serviceContext,                                               // Current service context
"core",                                                        // Service module
"http://d6server:9080/services");                          // Context
```
Step 3: Query the repository for objects

Now that the ObjectService instance is instantiated, it can be used to query the repository for objects. Querying the repository requires a Qualification object that encapsulates the query criteria. In this case, a DQL qualification is used to find documents whose object name starts with “DFS”:

```java
// Build an object Qualification using the criteria expressed in DQL syntax
Qualification qualification = new Qualification("dm_document where object_name like 'DFS%'");
```

In order to pass the Qualification to the ObjectService, it must first be associated with an ObjectIdentity instance:

```java
// Create an ObjectIdentity with our DQL qualification and repository name
ObjectIdentity targetObjectIdentity = new ObjectIdentity(qualification, repositoryName);
// Create an ObjectIdentitySet that contains the ObjectIdentity
ObjectIdentitySet objIdSet = new ObjectIdentitySet(targetObjectIdentity);
```

Next, create an OperationOptions object to pass options to the services call:

```java
// Create an empty OperationOptions object.
// This object encapsulates options, profiles, and filters that control the
// behavior of the service. An empty instance causes default values to be used.
OperationOptions opts = new OperationOptions();
```

Finally, the ObjectService.get() call is invoked:

```java
// Execute ObjectService.get(). This will return a populated DataPackage.
DataPackage dataPackage = objectService.get(objIdSet, opts);
```

Step 4: Examine the results

The DataPackage returned by the call to ObjectService.get() must be iterated through to access the contained DataObject instances. Each DataObject contains a PropertySet that is used to access individual property values:

```java
// Iterate through the DataObject objects in the result DataPackage
Iterator iterator = dataPackage.getDataObjects().iterator();
while(iterator.hasNext()){
    DataObject thisDataObject = (DataObject) iterator.next();
    // Write out some properties for this DataObject
    PropertySet properties = thisDataObject.getProperties();
    System.out.println("Object Name: " + properties.get("object_name").getValueAsString());
    System.out.println("Creation Date: " + properties.get("r_creation_date").getValueAsString());
}
```

“Appendix: DFS Java client sample code” provides a complete code listing for this example.

BPM and SOA: A DFS example using Process Builder

The following example shows how Documentum Process Suite’s Process Builder application can be used to orchestrate DFS services without requiring any code to be written. Documentum Process Suite is EMC’s BPM offering.
This example illustrates service orchestration by configuring and coordinating services provided by three systems: DFS services from Documentum, services from a purchasing system, and services from an accounts payable system. These services are sequenced to create an end-to-end composite purchasing process. When a new invoice arrives, it is validated against a purchasing system. If the purchasing system answers that the invoice is valid then the invoice gets paid. Otherwise, a BPM subprocess is initiated to review the process.

**Process data**

An essential step when implementing a business process is to define the data used in the process. Figure 2 is a snapshot of the Process Builder screen that allows us to define this data.

![Process Variables](image)

**Figure 2. Define the data**

**Process template**

Process Builder includes many activity templates for automated tasks and integration points. Orchestration of services is accomplished by dragging process activities from a palette of available processes onto the process canvas and then configuring the flow of execution. The resulting process model manages how data
flows through each of the individual services. This model is essentially the blueprint to the overall business process. The process model Figure 3 illustrates the order of operations for the example.

**Figure 3. Process model**

By dragging the Dynamic Web Service activity template from the palette onto the canvas we can configure the invocation of a service. The next step is to identify and configure the individual services that will be orchestrated within the process from the Web Services Descriptor Language (WSDL) file. The Activity Inspector allows the designer to locate the WSDL for each service.

Figure 4 is a snapshot of the Activity Inspector being used to connect to the QueryService DFS Service.

**Figure 4. Activity Inspector connecting to the QueryService DFS Service**
In Figure 4, the Activity Inspector has been pointed to the WSDL of the QueryService. Upon resolving the address to the WSDL, the Activity Inspector prompts for the Port Type and the Operation to use. These map directly to the underlying web services operation that will be invoked.

**Data mapping**

Next, we need to create data mappings that allow data to flow between the services in the system. For each web service call we map the process data to inputs and outputs for the web service. The data mappings are easily configured using the Activity Inspector.

In Figure 5, the docbase.name variable from the process execution data is mapped to the repositories[0] input on the QueryService service.

![Figure 5. Creating a data mapping](image)

In this example, we are watching for a new invoice document in a specific path of the Documentum repository. In this case we will simply input the query by using the data mapping function to map a fixed string into the @queryString input:
The mapping function allows for the combination of inputs from many sources (process execution, data payloads, user input, and others) to form inputs.

Once all of the input parameters have been mapped, the service response needs to be captured by mapping it back into the process data. In Figure 7 the ObjectId output from the QueryService is mapped to the Invoice.r_object_id parameter in the Process Data.
Figure 7. Mapping the service response

In Figure 8, a number value corresponding to the `amt_due` portion of the input query is mapped to the `invoice_amt` variable in Process Data.

Figure 8. Mapping a number value
It is possible to conditionally map data depending on whether the input data meets certain criteria. In this example, the `po_num` property is mapped to the `po_num` Process Data variable only if the input name equals the literal “`po_num`.” First, the node is made conditional, as shown in Figure 9.

![Figure 9. Making a node conditional](image1.png)

Figure 9. Making a node conditional

Then, the condition is created using the Conditional Node Builder dialog box.

![Figure 10. Creating the condition](image2.png)

Figure 10. Creating the condition

**Defining business logic**

Once data mapping is complete, the final step to creating a composite business process is to define the business logic that will route data through the system. In this example, we examine the Valid Invoice? decision point in the Process Template.
This business logic can be defined without any programming whatsoever. To construct the logic, the user simply opens up the Activity Inspector on the decision node and enters the transition criteria, selecting the desired options within the editor.

![Activity Inspector](image)

Figure 11. Entering the transition criteria
Once each of the nodes in the Process Template has been appropriately configured, the process definition is complete. The process is then ready for public consumption — no programming required!

**Conclusion**

SOA. Web services. DFS. What do these terms have in common? These are the architecture and technologies that Documentum 6 leverages to share its functionality with business processes across an enterprise. Technology agnostic, location insensitive, and intuitive to use, DFS provides the robust, modular, and flexible foundation that enables the Documentum platform to contribute its industry-leading content management services to the enterprise SOA environment.

When coupled with the Documentum Process Suite, DFS becomes an efficient enabler of business agility. Together, Documentum’s SOA and BPM offerings allow business solutions to be constructed with minimum effort and with no programming required.

**Appendix: DFS Java client sample code**

This code example shows the basics of programmatically invoking a DFS service call using the Java Client Library.

```java
package com.acme.dfs.sampleclient;

import java.util.Iterator;
import com.emc.documentum.fs.datamodel.core.*;
import com.emc.documentum.fs.rt.ServiceInvocationException;
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.IObjectService;

public class SimpleClient {
    public static void main(String[] args) throws ServiceInvocationException {
        // Set important variables
        String repositoryName = "d6_test1";
        String userName = "admin";
        String userPassword = "password";

        try {
            // Get a ContextFactory so that we can
            // create a ServiceContext
            System.out.println("Creating ContextFactory");
            ContextFactory contextFactory = ContextFactory.getInstance();

            // Use the ContextFactory to create
            // a ServiceContext for a service
            System.out.println("Creating ServiceContext");
            IServiceContext serviceContext = contextFactory.newContext();

            // Create a RepositoryIdentity to store the
            // repository credentials on the ServiceContext
            System.out.println("Creating RepositoryIdentity");
            RepositoryIdentity repoId = new RepositoryIdentity();

            // Populate the repository credentials
            System.out.println("Populating RepositoryIdentity");
            repoId.setRepositoryName(repositoryName);
            repoId.setUserName(userName);
            repoId.setPassword(userPassword);

            // Add the populated RepositoryIdentity to the ServiceContext
            System.out.println("Adding RepositoryIdentity to ServiceContext");
        }
    }
}
```
serviceContext.addIdentity(repoId);

// Get an ObjectService from the ServiceFactory
IObjectService objectService = ServiceFactory.getInstance().getRemoteService(IObjectService.class, serviceContext, "core", "http://d6server:9080/services");

// Query the repository for objects matching our criteria
// Build an object Qualification using the criteria expressed in DQL syntax
Qualification qualification = new Qualification("dm_document where object_name like 'DFS%'");

// Create an ObjectIdentity with our DQL qualification and repository name
ObjectIdentity targetObjectIdentity = new ObjectIdentity(qualification, repositoryName);

// Create an ObjectIdentitySet that contains the ObjectIdentity
ObjectIdentitySet objIdSet = new ObjectIdentitySet(targetObjectIdentity);

// Create an empty OperationOptions object
OperationOptions opts = new OperationOptions();

// Execute ObjectService.get() which hopefully returns a populated DataPackage
DataPackage dataPackage = objectService.get(objIdSet, opts);

// Iterate through the DataObject objects in the result DataPackage
Iterator iterator = dataPackage.getDataObjects().iterator();
while (iterator.hasNext()) {
    System.out.println("######## GETTING OBJECT PROPERTIES ########");
    DataObject thisDataObject = (DataObject) iterator.next();
    // Write out some properties for this DataObject
    PropertySet properties = thisDataObject.getProperties();
    System.out.println("Object Name: " + properties.get("object_name").getValueAsString());
    System.out.println("Creation Date: " + properties.get("r_creation_date").getValueAsString());
}

} catch (Exception e) {
    e.printStackTrace();
    throw new RuntimeException(e);
}