THE UNIVERSITY OF HULL

RepoMMan Project

D-D15 System Documentation

Simon Lamb

September 2007

The RepoMMan Project

Project Director:	Ian Dolphin, Head of e-Strategy, University of Hull	(i.dolphin@hull.ac.uk)
Project Manager:	Richard Green	(r.green@hull.ac.uk)
Technical Lead:	Robert Sherratt	(r.sherratt@hull.ac.uk)
Repository Domain Specialist:	Chris Awre	(c.awre@hull.ac.uk)
Lead software developer:	Simon Lamb	(s.lamb@hull.ac.uk)

The Repository Metadata and Management Project (RepoMMan) at the University of Hull is funded by the JISC Digital Repositories Programme. The project is being carried out by the University's e-Services Integration Group (e-SIG) within Academic Services.

Contents

Introduction 4

Purpose of document 4

Overview 4

Methodologies, tools and techniques 6

Systems architecture 7

Elements of the RepoMMan system 8

BPEL Processes 8

The RepoMMan bespoke services 19

Developing and deploying Web Services 19
The RepoMMan services 25

Testing

Appendix 1: FOXML for an example file object 42

Appendix 2: FOXML for an example collection object 44

Introduction

The RepoMMan Project was established to provide a user interface to a Fedora digital repository which would allow a user to interact with it as an integral part of his or her workflow.

The brief of the JISC-funded project required the use of a BPEL engine to orchestrate Web Services in order to provide the required functionality through a browser.

Purpose of document

Intended audience

This system document's intended audience is software developers, software engineers, system administrators and anyone with an interest in the architectural and technological background of the RepoMMan tool.

Purpose

The purpose of this document is give an overview of the RepoMMan tool and the architecture upon which it is built. The RepoMMan architecture is made up of several layers. These layers include the Adobe Flex Presentation layer, BPEL Processes, and the RepoMMan and Fedora Web Services that they call. The document will also cover the Java servlets that have been written to provide functionality that services could not provide.

The document assumes a certain familiarity on the part of the reader with BPEL, with digital repositories in general, and with the Fedora Commons repository software in particular. The RepoMMan website[1] provides a wide range of documents that may help with this, as does the Fedora Commons website[2] and the Active Endpoints Inc website.[3]

Overview

The purpose of the RepoMMan tool is to allow users to interact with a Fedora-based digital repository in order to use it as part of their workflow in producing materials that might later be moved to the University of Hull institutional repository. The specification for the tool required that it allow for management of digital assets (perhaps texts, images, datasets etc), including versioning and safe storage in a 'private' repository space, and that it should provide flexible access to them. Flexible access meant providing easy 'any time, anywhere' download of the material from the repository using only a browser and also the provision of limited multi-user access sufficient to support collaborative authoring.

The project produced an interface which covered all but the last requirement. Development of a 'sharing' facility required functionality in the underlying Fedora

[1] At: http://www.hull.ac.uk/esig/repomman
[2] At: http://www.fedora-commons.org/
[3] At: http://www.active-endpoints.com/

software which was not available when the project closed; it will be added into the tool at a later stage.

The top level of the tool, as delivered, looks like this:

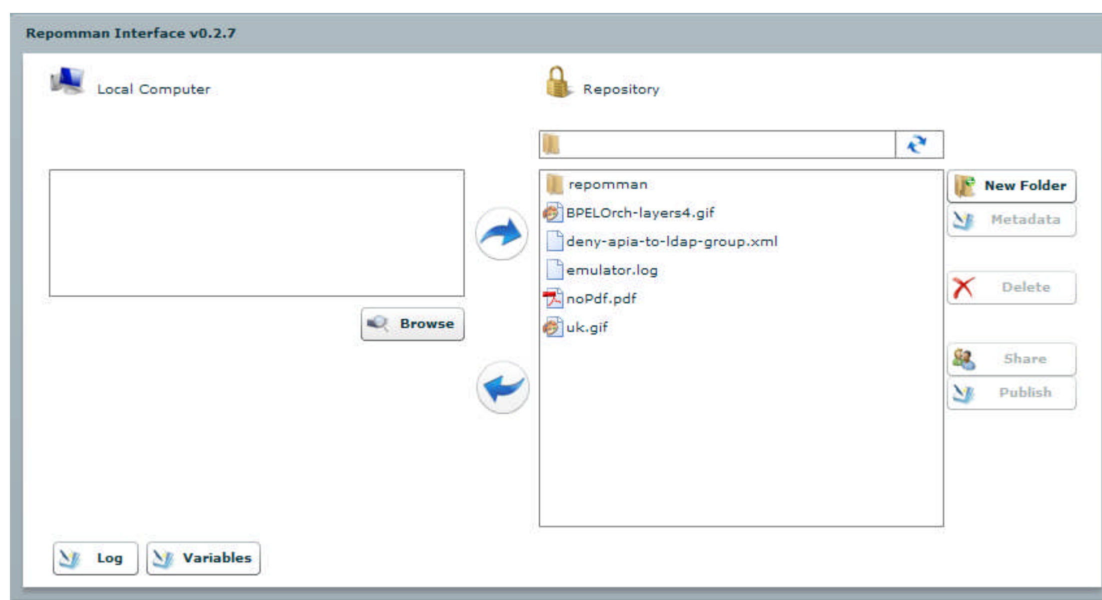

Figure 1: The RepoMMan user interface

The interface deliberately mimics an FTP tool with which many users will be familiar. The left-hand side of the screen provides a browse facility onto the user's local computer. The right-hand side of the screen provides a window onto the user's private repository space. The contents of the repository is presented in the form of a conventional Windows file layout although the reality on the remote system is much different. The 'folders' on the user's screen represent Fedora collections whilst the 'files' represent Fedora objects which may contain a number of versions of their binary content, or digital payload, as well as metadata etc. The user need not be aware of these distinctions.

A number of buttons on the interface provide functionality:

- for selecting (by browsing) a file on the user's computer
- for upload and download of files between the user's computer and the repository
- for creating a new folder (collection) in the repository
- for adding metadata to a repository object
- for deleting a single version of an objects binary content
- for deleting folder or an entire object (all versions)

Two further buttons provide access to a session log and to a variables page to assist potential technical support.

The screen has two non-functional buttons as placeholders. One of these will provide access to 'sharing' facilities whilst the second will allow the user to 'publish' an object into the receiving queue for the University's main repository. As noted above, sharing was put on hold pending a new version of the Fedora software to support it whist publishing was not part of the JISC project agreement although the functionality will shortly be added for use within the University.

The interface is intended to be used within a secure environment, for instance the University portal or an institutional VLE or VRE which will deal with authentication and pass the information invisibly to the tool.

Methodologies, tools and techniques

The following is a list of the tools and libraries used for development of the RepoMMan tool.

Development environments

- Eclipse 3.2 For Java developers
- ActiveBPEL Designer 4.0
- ActiveBPEL Engine 3.0
- Adobe Flex Builder 2.0

Testing environments

- SoapUI 1.7.6 SOA and Web services testing application

Libraries

- Apache Tomcat 5.0.28
- Apache Axis 1.4
- Castor 1.0.5

Systems Architecture

This section will describe the system architecture beneath the RepoMMan tool. This will include detailing the layers within the system and what they comprise.

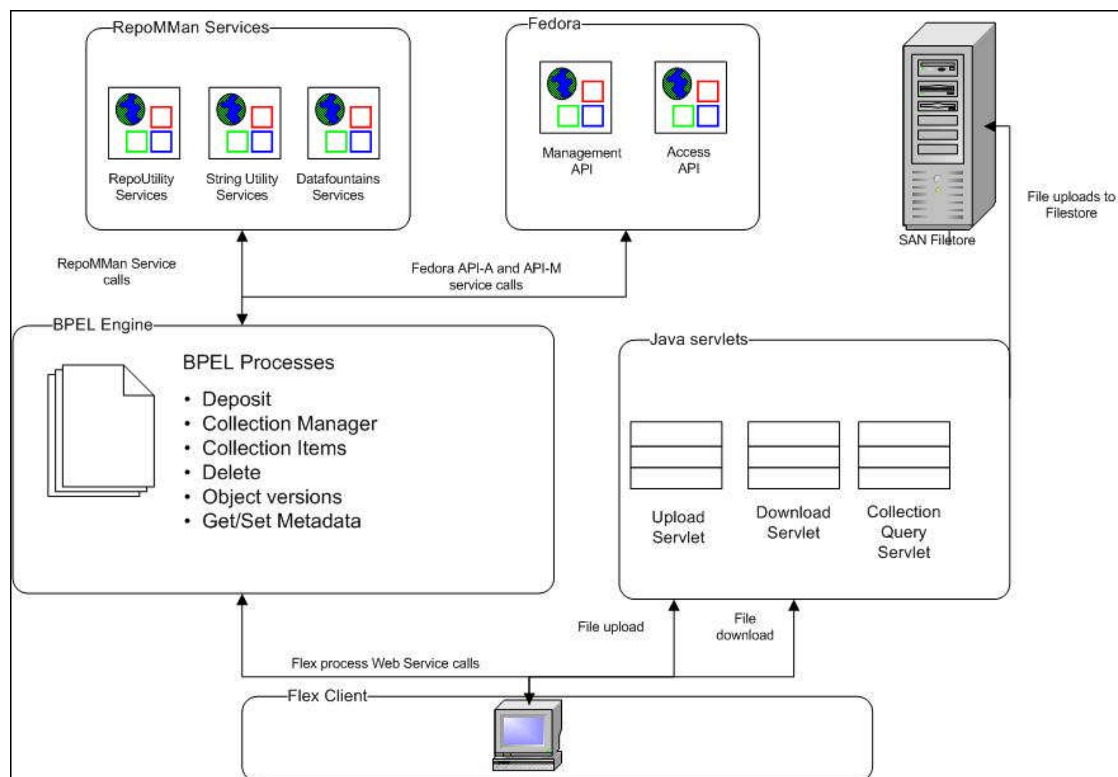

Figure 2: The RepoMMan architecture

The diagram presents an overview of the full architecture of the RepoMMan tool. 'Adobe Flex' provides the rich, and intuitive client for the tool. Flex itself communicates to the server-side via calls to the 'Web service' exposed BPEL processes and the Java servlets via HTTP requests and responses.

The bulk of the RepoMMan tool logic is built into the BPEL processes, these themselves call upon specific RepoMMan Web Services in order to undertake tasks that are specific to RepoMMan needs. Because BPEL orchestrates and consumes functionality via Web Services, all the tools that had to be integrated or developed had to be implemented in such a way that they could be exposed as Web Services.

Fedora itself exposes its Management and Access API's as Web Services, which means that calls to Fedora fit into the BPEL process architecture easily. This ready availability of Web Services to address repository functionality was a major factor in choosing the Fedora software as the basis for RepoMMan's work.

The Java servlets provide the 'File/large data' transport that, at the time of development, did not easily fit into the services approach. It was decided that rather then encoding files and transporting the binary along with service calls, FTP would provide a more reliable and secure option. As the diagram shows, at the University of Hull the filestore for binary content (as opposed to the XML of the repository digital objects themselves) is provided by the University storage area network (SAN) which allows FTP and HTTP (required for Fedora) access. It

is worth noting that a new version of the Active Endpoints BPEL engine used by RepoMMan will apparently support straightforward movement of files; this being the case, the RepoMMan tool will almost certainly be reworked to manage the transport of binary content using this new BPEL process.

Elements of the RepoMMan system

This section will provide an overview of the constituent entities within the RepoMMan system.

BPEL Processes

The BPEL processes layer is the hub for all RepoMMan functionality. There are several processes that serve specific purposes with regards to the functionality of the RepoMMan tool:

- Collection Manager
- Collection Items
- Deposit
- File versions
- Delete
- Get Metadata
- Set Metadata

Collection Manager

The vision of the RepoMMan tool was to provide users with a view of their repository space that is almost identical to a 'Windows Explorer' tree. This suggests a need to be able to mimic such a structure within the repository. Whilst the concept of 'repository object' maps easily onto 'file', there is no close equivalent in a Fedora repository to the concept of a 'folder'. However the repository allows 'collections' of objects to be created and this facility can be used to provide pseudo-folders. Thus the 'folders' that the user sees in the RepoMMan tool are, in practice, collections of Fedora objects. Functionally, this is not a distinction that the user need understand.

Fortunately within Fedora, creating collections providing a collection disseminator and collection RDF query to manipulate them is relatively trivial.

A collection object with the necessary collection disseminators and query is created to act as the folder within the tool. Fedora provides two methods of managing collections. An explicit collection is one where the collection object itself contains a list of all its member objects, whereas an implicit collection is one where each member of the collection individually asserts its relationship with the collection object.

For the RepoMMan tool, it was deemed the best solution to have every object declare itself as a member of a collection i.e. to use the implicit method. This means that every time a new object is deposited into the repository, the collection object itself does not have to be edited to take into account the new member.

With the need for using collections as an integral part of the RepoMMan tool, a Collection Manager process was designed in order to manage the creation of collections (and thus pseudo-folders). It was decided that the each user of the tool would have a root collection, which maps to their root folder within the tool. From there on, the user can create folders and sub-folders which in turn would create sub-collections using the same Collection Manager process.

The Collection Manager process role is to undertake the management of creating and retrieving a user's root collection. It is used at all times when a new collection is needed, including when a user creates a new folder in their repository space.

What the process does...

The purpose of this process is to service a request for a root collection (i.e. the initial directory for a user's repository space) - this can be done by either returning details of the root collection that already exists or by creating a new one. The process will also service the creation of new 'user created collections' which appear to the user as folders and sub-folders within their repository workspace.

The 'ReceiveCreateCollectionRequest' takes the following the variables from the client:-

- userID
- rootCollection
- collectionPID
- collectionName

The rootCollection variable determines whether the collection being requested is a ROOT or a personal collection. If rootCollection is set to True, then the service will do a search using the Fedora Access operation 'findObjects' to see if a root collection exists for the userID specified. If a rootCollection already exists for the user, then the PID will be returned and the CollectionResponse reply will be done.

When a rootCollection does not already exist or an additional personal (sub-) folder is being requested, a new collection will be built together with the components needed to manipulate it.

To do this a new Fedora collection object needs to be built in FOXML1.0 (Fedora Object XML). The ingest process involves calling the RepoMMan service 'BuildCollectionObject' operation which outputs Fedora compliant FOXML1.0 for a new object, based on 'Filename' (not used in this instance), 'ObjectPid', 'mimeType', 'userID' and 'content location' (again, not used for a collection. The object pid (permanent identifier) is derived from a call to the Fedora's getNextPid operation. There are some extra inputs including the collectionName, rootCollection boolean (the web service will build slightly different objects depending if they are a root or personal collection) and the MemberQuery script location which needs to be included in the collection datastream. (This small script is in fact an RDF query which is run against the repository to identify the members of an implicit collection.) The service will return a FOXML1.0 collection object as a string.

The process then only has to convert the given FedoraObject into Base64Binary, and this can then be ingested into the Repository, using Fedoras API-M Ingest operation.

The PID of the collection created will then be returned back in the Collection manager Reply activity.

Collection Items

When an existing user opens the RepoMMan tool and the PID of the appropriate root collection has been established using the collection manager, the MemberQuery script mentioned above must be used to return a list of the collections and files within the root collection.

In order standardise this task, a BPEL process 'Collection Items' was designed which would take as an input the ObjectPID of any collection (not just the root), and return, as an output, XML which contains a list of the files and/or collections within it.

What the process does...

The Collection Items process on the face of it is relatively straightforward, it does consume several services however the main benefit is that it returns a list of sub-items with enough information about each so that the client can display them intelligently.

The 'RecieveGetCollectionItemsReq' takes in five input variables:-

- CollectionObjectPID
- FedoraIP
- FedoraPort
- FedoraUsername
- FedoraPassword

The 'ReplyGetCollectionItemsResp' contains the following information about each of the sub-items in the collection:-

- ItemObjectPID
- ItemIsCollection
- ItemLabel
- ItemLastModified
- MimeType

The process will first call the 'GetCollectionObjects' service, this is a simple but important RepoMMan service that will return a list of ObjectPIDs that represent the sub-items of the CollectionObjectPID provided to it. It may seem that this is all we need to display a list of the sub-items, however it is much more valuable to the client that more information about each object is returned in the resulting XML.

The next stage in the Process goes into a 'For Each loop', this is where more information is extracted from each of the objects in the returned collection list. The 'InvokeGetObjectProfile', calls the Fedora access service operation which returns the ObjectLabel, ObjectType, ModifiedDate and many more.

An 'If' statement test is then completed to determine whether the object being processed is a Collection object itself. This is simply done by testing for the 'c' at the end of the objectPID, the isCollection boolean is set depending if it does or not. The addition of the letter 'c' to collection PIDs was a device adopted by the

project to make them easy to identify, as here. Collection objects are actually very similar in structure to objects with binary content and this step provides a simple means to distinguish them.

The next stage aims to retrieve the mimeType for the non-collection objects. If the object is not a collection, the Fedora operation 'GetDatastream' is called. This is called against the 'file' datastream in the object, which contains the information about the binary content, sometimes called the data payload, referenced by the object, one such piece of information is the mimeType.

As the process goes through the 'For each loop' testing the collection members the output information is set using the assign activity. Once all of the objects have passed through the 'For each loop', the 'ReplyGetCollectionItemsResp' activity is called. At this point the list of Collection items will not only contain the ObjectPID, but the other information as detailed above in an XML form.

Deposit

Deposit is obviously one of the most fundamental parts of providing a Repository system to a user. This process is responsible for either creating and ingesting a new object, if a user is depositing a new file, or updating an existing object with a new version of its binary content if a previous version already exists.

What the process does...

The process will either create and ingest a new object together with the necessary URL reference to its binary content or, if it is a new file version, perform an 'update datastream' action on the appropriate existing object within Fedora.

Before this happens the filename is derived from the URL supplied, using some bespoke RepoMMan services. Fedora's findObject operation is invoked in order to find if the file already exists within the repository, and depending on this result a search will be done to establish whether any files found are within a different collection or not. Depending on this either an ingest or an update is performed.

The ingest process involves the calling of the RepoMMan service 'BuildObject' operation which outputs Fedora compliant FOXML1.0 for a new object, based on 'Filename', 'ObjectPid', 'mimeType', 'userID' and 'content location'. The object pid is derived from a call to the Fedora's getNextPid function.

If an update is required a call will be made to the Fedora management service 'ModifyDatastreamByReference', this allows for the addition of a new content location in the object identifying a new version of the binary content. The location is a URL pointing to the file of content on the University SAN.

Once the process is complete, the deposit process returns either 'true' or 'false' for the NewObject type (this was an ingest or update), and the object PID.

File versions

The RepoMMan tool presents the user with a view of all the files (objects) and folders (collections) they have created and deposited into the repository. When the user double-clicks on a particular filename, the various versions stored on the

repository are displayed to user. This allows them to manage versions of their current work, and revisit previous versions if required.

Versions are handled in Fedora automatically; when a particular datastream in a Fedora object is updated it retains the previous datastream content if set up for versioning. RepoMMan had to deal with the situation that all versions of a file uploaded by a user were likely to have the same filename. This was solved by invisibly (to the user) adding a date and time stamp to the end of the filename. So for instance, when the 'file' datastream (the reference to the binary content) gets updated with the new contentURL for the new version of the file in the file-store, the previous reference(s) also remain stored.

The aim of the File Versions process is give the RepoMMan client enough information to display the relevant details about each version (date and time, mimeType, etc) and allow the user to download or delete the said version. The Flex client uses the Fedora fileLocation URL to serve the user when they want to access a particular version of the file. Fedora can use URL access to the binary content of a particular object in the following format:-

http://ServerAddress/fedora/get/OBJECT_PID/DATASTREAM_ID

In the case of the RepoMMan tool, all data payload datastreams use the ID 'file'. Fedora allows access to previous versions of the datastream by adding the DateTime to the end of the URL, for example a RepoMMan version:-

http://serverAddress/fedora/get/hullwf:469/file/2007-09-14T14:11:01.422Z

The File Version process checks a particular object for versions and constructs the URLs of the versioned binary content. Together with the mime type of the files, this information is passed back to the RepoMMan client. Keep clear the distinction between the URL passed to Fedora to retrieve a version of the binary content for an object (as in the last example above) and the URL internal to the Fedora object which retrieves a particular file of content from the SAN.

What the process does...

The 'RecieveGetFileVersionsReq' takes in four inputs variables:-

- ObjectPID
- ObjectLabel
- fedoraIP
- fedoraPort

The 'ReplyGetFileVersionsResp' returns the following information about each of the file versions that exist:-

- fileLocation
- VersionDateTime
- mimeType

The File Versions process's first activity is to assign the variables required to invoke the Fedora Management operation 'GetDatastreamHistory', this operation will return xml describing all the versions that exist for a particular datastream. In this case, we are returning all the versions of the 'file' (data payload) datastream for the desired object. The 'GetDatastreamHistory' operation returns

quite verbose data regarding each of the datastream versions that exist, including ID, VersionID, MimeType, CreateDate, location etc..

The File Versions process then goes into a 'While' loop, this will loop based on the number of versions returned in 'GetDatastreamHistory'. On the first Loop (determined by the counter being set to 0), the process will branch to set the 'GetFileVersionsResp' to the first of the file versions (i.e. the current version). Because the current version of the datastream doesn't require an date/time appended, the assign in this command will not add one onto the returned 'fileLocation'. For subsequent loops, the process will follow into the second branch where the DateTime of the version will be appended to the 'fileLocation' URL.

Once the output XML has been created for all the file versions that exist for the object, the process will continue to the 'ReplyGetFileVersionsResp' activity, at this point the XML identifying all the file versions is returned to the client.

Delete

The RepoMMan tool required functionality for users to delete the files, collections and file versions they have created in their personal space within the repository.

In the first mode, deleting an entire file object, the delete will purge the object from the repository and delete the physical file from the Repository file-store. The Delete BPEL process takes an input of ObjectPID; it purges the object from Fedora and deletes the binary content that it references from the data-store. Similarly for a Collection Object it will need to delete the collection object from Fedora but with the added complication that the sub-objects and sub-collections and so forth need to be deleted as well. When it is just a specific version of a file that is being deleted from an object, then the object itself doesn't need to be purged, rather the reference to the version must be deleted from the object and the associated file of binary content removed from the file store.

What the process does…

The DeleteObject BPEL deals with three specific circumstances: deleting a collection object, deleting an entire file object, and deleting a single file version from an object. The first is a simple operation requiring that the object be purged from the repository. The second operation requires that all the relevant files on the filestore also be identified and deleted. Deletion of a version requires modification of the 'file' datastream within an object to remove the particular reference and deletion of the binary content from the file store. Deletion of a tree requires recursive calls of the first two.

The ReceiveDeleteObjectReq activity takes in the following list of input variables:-

- ObjectPID
- ObjectLabel
- deleteType
- userID
- fedoraIPAddress
- fedoraPort
- fedoraUsername
- fedoraPassword
- ftpIpAddress

- ftpPort
- ftpDirectory
- ftpUsername
- ftpPassword
- VersionDateTime

The ReplyDeleteObjectResp outputs the following variable:-

- ObjectDeleted

The process first assigns all of the default variables, this includes FTP and Fedora settings which will be used throughout the process.

Deletion types

Collection object

If the current object is a collection, determined by the 'c' at the end of the objectPID, the ObjectPID and other default variables are assigned to the Fedora management operation 'PurgeObject' and then it is invoked. At this point the collection object is deleted from the repository - no other operations are required.

File object

If the current object in the loop is a file object there is a bit more involved in comparison with deleting a collection object from the repository. The first operation called is 'GetDatastreamHistory' this is from the Management API in Fedora, the purpose of this is to retrieve all the versions of the 'file' datastream, so that we know which physical files need to be deleted from store. Once we know how many file versions there are and their naming details, the Fedora object itself is purged using the 'PurgeObject' operation.

The process then goes into a 'while' loop, which loops on the number of file versions there are. For each file version returned in 'GetDatastreamHistory', the process will first invoke the RepoMMan String utility service 'getFilenameFromURL'; this operation simply takes in the URL location of the file, and returns just the filename itself. Using the filename, as well as the user's username and other default variables the process can then invoke the RepoMMan operation 'DeleteFile' to delete the file from the file-store. Once all the file versions have been deleted, the process will loop around again to delete the next sub-item if it exists.

File version delete

When we want to a delete a file version, we are only deleting a specific version of the binary content referenced within the object's 'file' datastream. So in this mode, the object itself does not need purging however we still need to delete the file from the remote file-store and the particular content version from the object's datastream.

The first operation to be invoked in the 'fileVersion' branch is 'GetDatastream' from the Fedora management API. The Input variables are the ObjectPID, the datastreamID (always 'file') and date/time. The operation returns a large amount of data about the datastream, including

mimeType, label, contentURL and much more. However the important part is the contentURL as it gives us the Filename details stored on the remote file-store. The next activity in the sequence is the Fedora management operation 'PurgeDatastream', this allows us to delete a whole datastream (not what is required here), or a specific version within it by entering date/time ranges. (The date/time of the version to be deleted will be submitted to the Delete process by the RepoMMan client when it is invoked.) The RepoMMan StringUtility service 'GetFilenameFromURL' is invoked to return the filename from the 'contentURL' variable derived from the file datastream.

Once the filename of the version to be deleted is derived, the file itself can be deleted from the remote file-store. At this point the RepoMMan service 'DeleteFile' is invoked using the default FTP settings, the user's username and the filename.

The last remaining activity in the process in all cases of 'delete' is the 'ReplyDeleteObjectResp' which informs the client if the Delete activity be it a 'Collection object', 'singular object' or 'file version' has been successful or not.

Recursive deletion

If the item identified by the user for deletion in the RepoMMan client is a folder (collection) it is not appropriate simply to purge it from the repository: it may be the parent item in a tree of content.

In this case the first activity that gets invoked is the 'GetAllCollectionObjectsChildren' service. In essence this takes a Collection ObjectPID and returns a list of all the sub items, including their sub items. Once the BPEL process knows all the ObjectPID's for the sub-items it can then loop through them from the lowest level of the structure to delete them from the Repository. Eventually the parent collection item is itself deleted. The deletion process uses the mechanisms for deleting file and collection objects outlined above.

Metadata

The RepoMMan tool includes the functionality to add metadata around a deposited object in the tool. One of the original aims for the tool was to automate as much of the metadata generation process as possible. There are two strands to this metadata generation, the first provides metadata about the author of an object by taking information from variables within the user's working context (usually the University portal or a VLE/VRE), the second uses a third-party tool to generate descriptive metadata from textual material that forms the binary content of an object. This work was undertaken at a 'proof of concept' level and will be somewhat altered and improved before the tool is deployed for general use in the University.

Here is not the place to consider the automatic generation of metadata, simply to note that the RepoMMan team identified a tool from the Data Fountains suite,[4] the iVia metadata tool, to use for this process.

The RepoMMan tool currently uses the Fedora 'DC' datastream to store all the metadata against the object; as noted above, this is proof of concept work and

[4] See: http://dfnsdl.ucr.edu/

will need adaptation and expansion. Other forms of metadata will eventually be used and the DC datastream populated by mapping this richer metadata down.

The Metadata part of the RepoMMan tool uses two different BPEL services 'Get Metadata' and 'Set Metadata'.

Get Metadata

The Get Metadata process is invoked when a user clicks the 'Metadata' button within the tool.

If it is the first time that the metadata function has been invoked, depending on the type of the file, metadata will be automatically generated. In all cases contextual metadata will be generated from the user context, additionally if file type is a PDF, Word document, or a text file the iVia metadata tool will be invoked in order to generate descriptive metadata.

If, however, metadata has already been generated for the object, then the Get Metadata will retrieve the metadata already stored against the object. This is so that a user can retrieve generated metadata, edit it to suit their needs and then save it.

What the process does…

When Metadata against a particular object is requested, the 'Get Metadata' process has two stages to determine the metadata returned. The first part of the process, retrieves the 'DC' metadata datastream from the Fedora object. If this already contains metadata created by or automatically for the user that metadata is returned, if no such metadata exists the process continues to the second stage of the process where it invokes the metadata generation tool.

The 'RecieveGetMetadataReq' takes in the following input variables:-

- ObjectPID
- UserID
- FedoraIP
- FedoraPort
- FedoraUsername
- FedoraPassword

The 'ReplyGetMetadataResp' outputs the following variables:-

- title
- version
- author
- publisher
- contributor
- citation
- language
- copyright
- format
- abstract

- keywords
- date
- type
- accessRights
- identifier
- source
- relation
- coverage

The process will first assign these variables for the invoke activity 'GetDublinCoreMetadata'. This RepoMMan service will execute a HTTP request on the DC datastream of the object, returning its metadata. Once the current metadata of an object is known, the process tests using an 'If' activity, to see whether it has been previously edited or is the 'default' form. The default is all of the metadata fields being empty apart from date, format and identifier which are set by Fedora on the deposit of a new object. If the Metadata has been previously edited, the process will invoke the 'AssignMetadataRespVariable' activity, at this point all the DC metadata is assigned to the reply variable 'ReplyGetMetadataResp', this is then sent back to the client.

However if it is found that the metadata has not been edited previously, then the process will continue to the 'metadata generation' part of the process. The first activity in this branch, is 'AssignGetDatastreamVariables', this assigns the 'ObjectPID' and the DsID to 'file', to gather the detailed information about the file datastream using the Fedora service 'getDatastream'. Once the 'InvokeGetDatastream' activity has completed, details including 'label, 'location, and importantly 'mimeType' are returned. The 'mimeType' determines whether metadata generation can be completed or not: the iVia tool can only currently generate metadata for Word documents, PDFs and text HTML files. However it is possible that different metadata generation tools can easily be integrated into the process and therefore added as a new branch.

The next 'If' statement determines which branch the process follows. It will compare the mimeType to 'application/msword', 'application/pdf' and 'text/html', if it is one of these it will follow this path. Otherwise the process will follow the 'else' path were default generated metadata is assigned to the 'ReplyGetMetadataResp' variable and returned to the client.

When the data payload is a Word document, PDF or Html text file then it will follow the Generate metadata path. 'AssignGenerateMetadataVariable' is the first activity to be called in this branch, this simply assigns the URL location of the file which acts as the input of the RepoMMan 'GenerateMetadata' service. The GenerateMetadata service will then invoke libiViaMetadata component of the DataFountains in order to gather useful metadata by analysing the file content. Once the metadata has been generated by the tool into the input file, the 'AssignGeneratedMetadataVariables' activity is called and the process will then return it to the client via the 'ReplyGetMetadataResp' activity.

Set Metadata

Once the user has clicked on 'Metadata' within the tool, the 'GetMetadata' process is invoked and generated or previously edited metadata is returned for the object. The user can then choose to edit metadata within the tool and then update it save it for future use. When the user clicks 'Update' the BPEL process 'SetMetadata' is called, and the Object DC metadata datastream is updated to reflect the changes.

What the process does…

The 'RecieveSetMetadataRequest' takes in the following input variables:-

- objectPID
- fedoraIP
- fedoraPort
- fedoraUsername
- fedoraPassword

The Metadata:-

- title
- version
- author
- publisher
- contributor
- citation
- language
- copyright
- format
- abstract
- keywords
- type
- accessRights
- identifier
- source
- relation
- coverage

The 'ReplySetMetadataResponse' outputs the following variable:-

- modifiedDate

The SetMetadata process is very straightforward as it only has one execution path, and that is to simply update the Fedora object with the supplied Dublin Core metadata. Once the process has received all the required variables through the 'RecieveSetMetadataRequest' activity, 'AssignGetDublinCoreXMLVariables' assigns all of the Metadata variables inputted, to the 'GetDublinCoreXML' service call. This RepoMMan service, simply just takes the edited metadata, and returns it as Dublin Core in formatted valid XML.

When Datastream contents are modified, the Datastream (in this case the Dublin Core Metadata) needs to be encoded in base64binary. The

RepoMMan StringUtility 'Encoder' undertakes this process, returning a String representation of the base64Binary, so after the Dublin Core XML is generated, the XML is assigned to the 'InvokeEncoder' input variable and invoked.

Once the edited Metadata is in Dublin Core XML format and encoded in base64binary the Fedora object itself can be updated to the reflect the changes. The Fedora management service 'ModifyDatastreamByValue' is called. This 'ModifyDatastreamByValue' operation requires the input of ObjectPID, dsID (which is 'DC' in the case of the metadata), dsLabel, dsContent (the encoded Dublin Core) and some checksum types which can be left to default. The ModifyDatastreamByValue service returns a 'modifiedDate' of the change, which in this case is assigned to the 'ReplySetMetadataResponse' activity and returned the client.

At this stage the Metadata changes made by the user have been reflected with updates to the object's Dublin Core datastream within Fedora.

The RepoMMan bespoke services

Although much of the functionality required for the RepoMMan tool is encapsulated within the Fedora repository software, it was necessary to provide somewhat more in order that the RepoMMan team could provide a user-friendly tool.

Building around the BPEL framework and the philosophy of Service Orientated Architectures (SOA), the bespoke functionality required to 'fill the gaps' was to be developed and deployed as Web Services. This meant that the RepoMMan services could easily be orchestrated alongside Fedora's own and any other services in the future.

Although many of the RepoMMan-developed services have a specific management or utility role, they do provide an essential part of the orchestrations described earlier in this document.

This first part of this section aims to give a background description of how the RepoMMan team developed the Web Services and deployed them for use within BPEL orchestrated processes. The second part will give a concise description of what each of the services does, and its role within the RepoMMan tool.

Developing and deploying Web Services

The RepoMMan team, choose Java for developing the backend to the services and indeed the development of Web Services. Java 2 Enterprise Edition (J2EE) itself provides many of the libraries required to develop and deploy Web Services, these include Java API for XML Web Services (JAX-WS) - for developing the Web Service and client software, Java Architecture for XML Binding (JAXB) - for providing support for binding XML data to Java beans, and many more libraries to support Web Service technologies. However the Apache Software Foundation themselves have produced an Open Source Web Service framework 'Axis' which the RepoMMan team chose to use.

Apache Axis provides the libraries for generating Web Services and the Java code for deploying Web Services on its Integrated SOAP Server. Axis provides support for both RPC/Encoded and Document/Literal Web Services. It can also generate Plain Old Java Objects (POJO's) for any XML schemas that are bound to a WSDL (Figure 3). However RepoMMan chose to use Castor, a more specialised tool for undertaking this code generation.

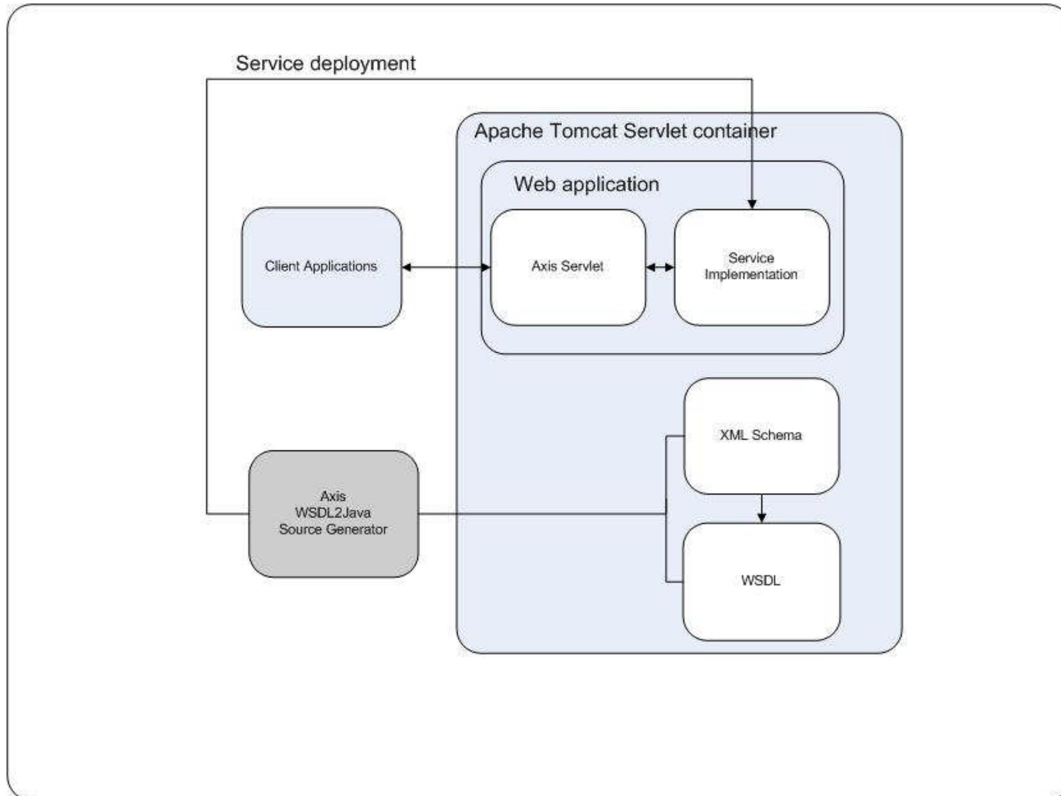

Figure 3: Overview of the Apache Axis code generation and Soap Server

Castor is an open source data binding library for Java, which allows XML schemas to be parsed into POJOs. With Castor being a standalone XML data binding library, it produces much cleaner and more human readable Java Objects that can be used as Beans in a J2EE solution. Another benefit is that the Castor-generated objects can be integrated in to the Web Services deployed on the Axis SOAP server.

In the initial stages of the project, the RepoMMan team developed test services using self deploying 'Java Web Service' files (or *.jws files). JWS files are simply, Java classes with simple 'public String getOutput (String inputVariable)' methods that Axis can generate simple WSDL from. These type of WSDL services are suitable for use in a test environment, however once multiple variables need to be passed as inputs and outputs to the service it starts to get messy.

Ideally the Document/Literal wrapped WSDL Web Services approach is the best choice for enterprise systems, due to its compliance with WS-I standards and its more strict validation through XML schemas. The RepoMMan approach to writing WSDL and Web Services has been based on the methods documented by IBM[5] and used by many Java WS professionals. Added to this, are Fedora's choice to update all its Management and Access Services to the new 'best practice' Document/Literal style and the excellent support for it in the BPEL specifications.

[5] http://www.ibm.com/developerworks/webservices/library/ws-castor/

The main difference to the automatic generation of WSDL by Axis, is the need to write the XML Schema and WSDL by hand. This may sound a laborious task, however it makes sense as it allows you to define the XML data model yourself and not leave it to generic code generation. This is also made easier by XML editors that typically have schema and WSDL templates.

The following is a brief example of the stages involved in writing the WSDL and XML schema for the DeleteFile operation for use in the RepoMMan tool.

The first task when developing a RepoMMan service was to define the data model needed, to encapsulate the input and output messages to operation. Figure 4 shows an example from the 'RepoUtils' XML Schema and the data from the Delete operation.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
targetNamespace="http://repository.hull.ac.uk/xsd/RepoUtils/2007/01/RepoUtils.xsd"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:ns="http://repository.hull.ac.uk/xsd/RepoUtils/2007/01/RepoUtils.xsd"
elementFormDefault="qualified" attributeFormDefault="qualified">
    <xs:element name="deleteFileInput">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="filename" type="xs:string" />
                <xs:element name="userID" type="xs:string" />
                <xs:element name="collectionQuery" type="xs:boolean"/>
                <xs:element name="objectPID" type="xs:string" />
                <xs:element name="ftpDirectory" type="xs:string"/>
                <xs:element name="ipAddress" type="xs:string"/>
                <xs:element name="ftpUsername" type="xs:string"/>
                <xs:element name="ftpPassword" type="xs:string" />
              </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="deleteFileOutput">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="fileDeleted"  type="xs:boolean"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
     <xs:element name="deleteFile">
        <xs:complexType>
            <xs:sequence>
                    <xs:element ref="ns:deleteFileInput"/>
               </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="deleteFileResponse">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="ns:deleteFileOutput"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Figure 4: An example from the 'RepoUtils' XML schema

Once the data model within the XML Schema xsd has been written, the WSDL can be written to use the data defined. As mentioned above, the RepoMMan team decided that the WSDL should be written in Document/Literal style; Figure 5 an

example of the DeleteFile operation and relevant WSDL using the data defined in the RepoUtils XML Schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
targetNamespace="http://repository.hull.ac.uk/wsdl/RepoUtils/2007/01/RepoUtils.wsdl"
xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
 xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:types="http://repository.hull.ac.uk/xsd/RepoUtils/2007/01/RepoUtils.xsd"
 xmlns:tns="http://repository.hull.ac.uk/wsdl/RepoUtils/2007/01/RepoUtils.wsdl">
      <wsdl:types>
              <xs:schema elementFormDefault="qualified"
targetnamespace="http://repository.hull.ac.uk/wsdl/RepoUtils/2007/01/importtypes"
attributeFormDefault="qualified">
               <xs:import
namespace="http://repository.hull.ac.uk/xsd/RepoUtils/2007/01/RepoUtils.xsd"
schemaLocation="RepoUtils.xsd" />
            </xs:schema>
    </wsdl:types>
    <wsdl:message name="deleteFileReq">
        <wsdl:part name="parameters" element="types:deleteFile"/>
    </wsdl:message>
    <wsdl:message name="deleteFileResp">
        <wsdl:part name="parameters" element="types:deleteFileResponse"/>
     </wsdl:message>
     <wsdl:portType name="RepoUtilsPortType">
           <wsdl:operation name="deleteFile">
             <wsdl:input message="tns:deleteFileReq" />
             <wsdl:output message="tns:deleteFileResp" />
         </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="RepoUtilsSOAPBinding" type="tns:RepoUtilsPortType">
        <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
           <wsdl:operation name="deleteFile">
            <soap:operation soapAction="" style="document"/>
            <wsdl:input>
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="RepoUtilsService">
        <wsdl:port name="RepoUtilsSOAPPort" binding="tns:RepoUtilsSOAPBinding">
            <soap:address location="http://localhost:8080/axis/services/RepoUtilsSOAPPort" />
        </wsdl:port>
    </wsdl:service>
 </wsdl:definitions>
```

Figure 5: An example of the DeleteFile operation and relevant WSDL

Once the RepoMMan Document/Literal wrapped WSDL and XML schema has been written, the next stage was to generate the Java stubs that undertake the processing between the Web Services and the business layer of code which contains the management and utility functionality. The Axis generated Java stubs provide the skeleton code to link the Web Service calls with the business layer. Apache Axis provides the functionality to do this using the WSDL2Java utility. Figure 6 shows how this would be invoked (note the NStoPKg options which are used to build the source as specific packages):

```
java org.apache.axis.wsdl.WSDL2Java -s RepoUtils.wsdl
--NStoPkg
http://repository.hull.ac.uk/wsdl/RepoUtils/2007/01/RepoUtils.wsdl=uk.ac.hu
ll.repomman.webservices
--NStoPkg
http://repository.hull.ac.uk/xsd/RepoUtils/2007/01/RepoUtils.xsd=uk.ac.hull
.repomman.webservices -o "src"
```

Figure 6: Using the WSDL2Java utility

This generated all of the code required for deploying the RepoMMan Web Service, including the data bindings of the objects representing the XML Schema data (Figure 7); this was to be replaced with the Castor generated code. However the client and server stub code and service binding data for the Axis server were used for deployment.

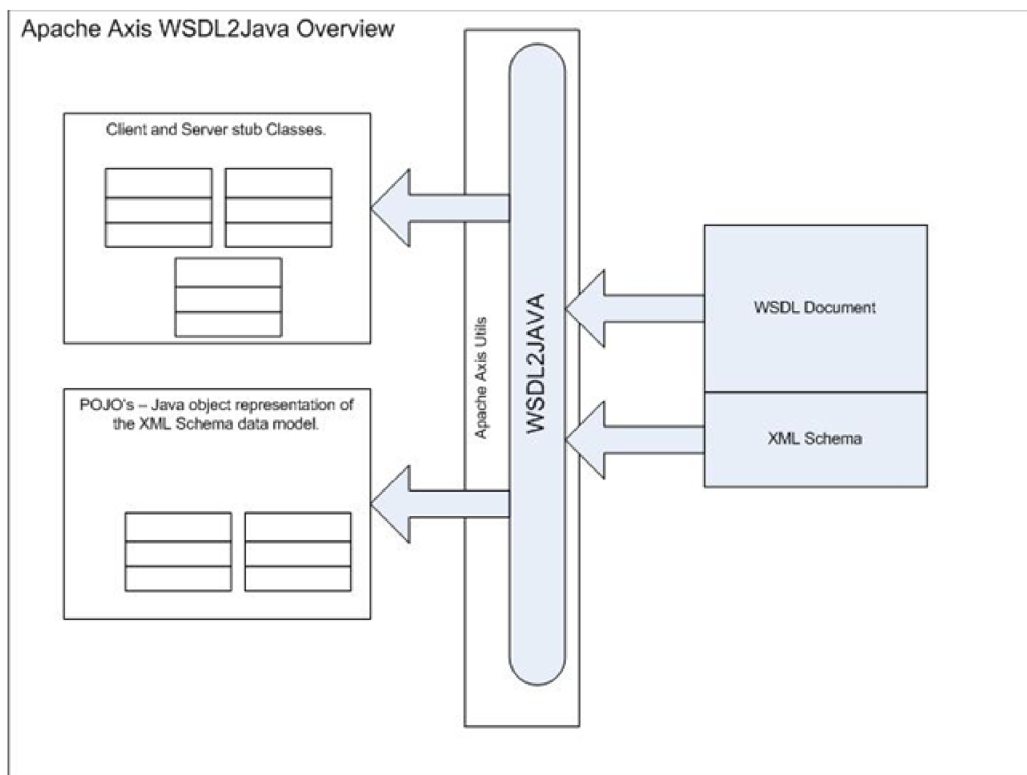

Figure 7 – Apache Axis WSDL2Java tool

The Castor utility was then executed from a command-line, with a pointer to the XML Schema file that needs processing, this created the Java object representations of the Data model.

```
java org.exolab.castor.builder.SourceGenerator - i "RepoUtils.xsd" -package
uk.ac.hull.repomman.webservices -nomarshall -dest "src" -f
```

Figure 8: Command-line, with a pointer to the XML Schema file that needs processing

The last change to make the RepoMMan Web services work with the Castor generated Java stubs was to change the Serializers and Deserializers in the 'Web Service Deployment Descriptor' (WSDD) file. The default lines being:-

```
org.apache.axis.encoding.ser.BeanSerializerFactory
org.apache.axis.encoding.ser.BeanDeserializerFactory
```

Changed to:-

```
org.apache.axis.encoding.ser.castor.CastorSerializerFactory
org.apache.axis.encoding.ser.castor.CastorDeserializerFactory
```

Once the RepoMMan WSDL, XML Schema and Java code had been generated, the code that links the Web Service to the Business logic at the back end had to be written. This stage in the process is relatively easy, the WSDL equivalent Java methods are contained in Axis generated *SOAPBindingImpl.java. The example for the 'DeleteFile' is shown in figure 9.

```
public uk.ac.hull.repomman.webservices.DeleteFileOutput
deleteFile(uk.ac.hull.repomman.webservices.DeleteFileInput deleteFileInput) throws
java.rmi.RemoteException {

                    //Creates instance of remoteFileUtilty for delete functionality;
                    RemoteFileUtility remoteFileUtility = new
RemoteFileUtility(deleteFileInput.getIpAddress(),
          deleteFileInput.getFtpUsername(), deleteFileInput.getFtpPassword());
                    uk.ac.hull.repomman.webservices.DeleteFileOutput deleteFileOutput =
new uk.ac.hull.repomman.webservices.DeleteFileOutput();

                    boolean collectionQuery =  deleteFileInput.getCollectionQuery();
                    String userID = deleteFileInput.getUserID().trim();
                    String ftpDirectory = deleteFileInput.getFtpDirectory().trim();
                    String filename = deleteFileInput.getFilename().trim();
                    String pathname;
…..

..
.

          return deleteFileOutput;

    }
```

Figure 9: Code tlinking the Web Service to the Business logic at the back end

As can be seen from figure 9, the delete file method takes the uk.ac.hull.repomman.webservice.DeleteFileInput object generated by Castor as an input. This object relates directly to the XML schema written in the first stage of creating the Web Service. Obviously the 'deleteFileInput' element in the XML Schema contained IpAddress, FtpUsername, FtpPassword etc… These variables are accessible In the Java by executing the automatically generated 'Getter' and 'Setter' methods. The code above shows these 'Getter' methods being invoked to get the IpAddress, Username and Password.

Once this has been compiled along with the other Axis and Castor generated classes, the package can be placed in the axis/WEB-INF/classes directory and the server restarted. On invoking the web service with the correct input parameters, it should execute the code and delete the file specified.

This was the approach used for writing all of all the RepoMMan Services for use in the BPEL orchestration.

The RepoMMan Services

As discussed earlier in this document, the RepoMMan team were required to develop and deploy some bespoke services in order to provide the functionality required from the tool.

Two services were written, one to provide Management and the other String Utilities. The Management services account for the operations that provide the major functionality for the RepoMMan tool, whilst the String Utilities provide functions that were sometimes only trivial but were not present in the BPEL tool itself.

The services were written using the methodology described above and deployed as Document/Literal style WSDL's. These were then invoked in the relevant parts of the BPEL processes, as described earlier in this document.

The name for the two services are:

- RepoUtils
- StringUtils

The next section aims to give a overview of the operations contained in the services, giving a detailed description of input and outputs and how they work.

RepoUtils

The RepoUtils service provides the operations required to undertake some of tasks that were required for the RepoMMan tool. These utilities generally fulfil a management role as they include such processes as building objects, returning lists of collection members, deleting etc.

buildFedoraObject

The 'buildFedoraObject' operation takes in the following Input Variables:-

- objectPID
- filename
- collectionPID
- contentLocation
- mimeType
- userID

This service will simply build a 'generic' RepoMMan Fedora object which will always contain the following datastreams:

> **File Datastream** – This is an 'Externally Referenced' Datastream, this means that the binary content itself is not contained within the object, but is referenced by a URL. This datastream is the basis for the 'file' reference, as the RepoMMan tool FTP's the binary content associated with the object to the external HTTP accessible filestore. The datastream has an essentially standard format in a basic object with only the URL reference being changed.

RELS-EXT Datastream – This datastream is used to register an object's relationship or association with another, as in a 'collection'. Again this datastream does not deviate much between objects, only in the relationship it defines.

Fedora Internal DC Datastream – The Fedora internal DC datastream is used to provide 'find' functionality within Fedora, using the Fedora service FindObjects. The Fedora default is a small subset of DC elements but for use in the RepoMMan implementation this datastream has containers for all the Dublin Core elements possible. The extra elements are used with the metadata functions of the RepoMMan tool described later.

The Object also contains some simple properties that are also defined when it is being built. These include:-

Label – A generic label to give the object: in the RepoMMan tool the label is used for the original filename of the content being deposited, for example 'SystemDoc.doc'.

OwnerId – The OwnerId is a relatively new feature in Fedora which allows one to specify an owner for the object. In terms of the RepoMMan tool this fits in quite well, as users can 'own' their objects in their private space until they make them public. Within the RepoMMan tool this is set to the UserID when the object is built. The Fedora OwnerId provides the basis on which to create XACML security policies which limit access to the object.

The 'buildFedoraObject' operation will always build a Fedora object that contains the above elements. The input parameters for the operation, listed below, provide the values specific to a particular object.

objectPID

Fedora allows objects to be ingested without an objectPID (a unique persistent identifier) being defined; in the absence of such a value a PID is generated by Fedora on the fly and inserted into the primary entry. However the RepoMMan tool depends on the use of collections to give users a 'file and folder' style of interface which, internally, requires the use of implicit collections. A member of an implicit collection uses its own PID when declaring its relationship with the collection object in the RELS-EXT datastream (see example at figure 10). Clearly, in order to do this the PID must be known beforehand.

As described in the 'Deposit' BPEL process description, a call to the Fedora access service 'getNextPID' will return a valid PID to use, and the service is simply called before invoking 'buildFedoraObject'.

The ObjectPID is also added to the primary Object PID properties within the XML (see the sample object at Appendix 1).

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rel="info:fedora/fedora-system:def/relations-external#">
  <rdf:Description rdf:about="info:fedora/hullwf:440">
    <rel:isMemberOf rdf:resource="info:fedora/hullwf:308c"/>
  </rdf:Description>
</rdf:RDF>
```

Figure 10: An example RELS-EXT datastream

filename

The Filename variable is used in multiple places in the RepoMMan Fedora object, including the Object 'Label', and internal DC metadata. Storing the filename in this way means that it can be displayed back to the user when they list their deposited objects.

collectionPID

The RepoMMan tool requires the functionality for users to create folders and upload files to those folders much like they would in their normal operating system. The way this is achieved in Fedora is through 'collections'. Collection objects are shown in the RepoMMan interface as a 'Folder'. 'Files' within the folder are created in Fedora using objects which are members of that collection. Because all file objects deposited into the RepoMMan tool are in a collection of some sort (if only the user's root collection), the CollectionPID is fundamental. Figure 10 above shows the XML from the Fedora 'file' object with PID 'hullwf:440' which is a member of a collection object with a PID 'hullwf:308c'. As noted elsewhere, the 'c' on the end of the collection PID is a RepoMMan addition to make the identification of collection objects straightforward.

contentLocation

As described above, the 'file' datastream is used to identify the binary content, or data payload, which in the RepoMMan implementation is held externally to the Fedora server. Because prior to invoking 'buildFedoraObject', the data-payload itself is uploaded to the externally accessible file store, the URL is known and therefore can be passed through to the service to be referenced in the object. See Figure 11 for an example of how it is used.

mimeType

The mimeType of the file being deposited into the repository is derived when the file is uploaded to the external filestore. This is then passed into 'buildFedoraObject' in order to be used in the 'file' datastream (see Figure 11), as Fedora requires that a mimeType is stored alongside the content reference.

userID

As described above the userID is fundamental to the Object with regards to the Private area on the RepoMMan tool. This is because it allows Fedora

to search by userID, allowing to quickly retrieve what a user has deposited. The userID is stored in the object properties as the 'ownerID'.

```
<foxml:datastream CONTROL_GROUP="E" ID="file" STATE="A" VERSIONABLE="true">
    <foxml:datastreamVersion CREATED="2007-10-10T10:00:33.517Z" ID="file.0"
LABEL="system-documentation.doc"
      MIMETYPE="application/msword" SIZE="0">
      <foxml:contentDigest DIGEST="none" TYPE="DISABLED"/>
      <foxml:contentLocation REF="http://www.google.com/Systemsdoc.doc" TYPE="URL"/>
    </foxml:datastreamVersion>
  </foxml:datastream>
```

Figure 11: Part of an object's FOXML showing contentLocation and MIMETYPE in use

Once the 'buildFedoraObject' service has completed building the object based on the inputted values it will return the 'ObjectXML' as a string to the 'Deposit' BPEL process. Appendix 1 shows an example of a complete RepoMMan Fedora object.

buildFedoraCollectionObject

The 'buildFedoraCollectionObject', is very similar to the 'buildFedoraObject' however in this case it builds an object that is used as the parent for an implicit collection, shown as a 'folder' in the tool.

Again the main principle is the same, that all the RepoMMan collection objects share essentially the same FOXML differing only in the values of key variables.

The service takes in the following input variables:-

- objectPID
- collectionPID
- memberQueryLocation
- collectionName
- rootCollection
- userID

Structurally a collection object is very similar to a singular (file) object in its XML makeup and so it is not necessary to repeat here large sections of the previous process. However there are some notable differences.

As discussed previously in this document, the purpose of a collection object within RepoMMan is to provide folder like functionality within the tool. To provide this, the object has to contain specific collection-related datastreams and disseminators. A collection object contains the DC and RELS-EXT (after all a collection can be a member of another collection) as discussed above, but with the addition of:-

memberQuery Datastream – The memberQuery datastream is another externally referenced datastream. The file that it references contains an RDF query, that is used to query the Fedora Resource Index to determine which objects declare themselves to be a member of that collection

(objects declare relationships through the RELS-EXT datastream). The MemberQuery for every collection object in the RepoMMan tool is same apart from the ObjectPID declaration (see figure 12).

```
select $member $memberPID from <#ri>
where $member <fedora-rels-ext:isMemberOf>
<info:fedora/hullwf:496c> and $member <dc:identifier> $memberPID
order by $memberPID;
```

Figure 12: An example RDF member query

In order for the RepoMMan tool to use this ‘memberQuery’ successfully, a so-called ‘disseminator’ (consisting of a Behaviour Definition (BDef) and a Behaviour Mechanism (BMech)) needs to exist for the collection object. The RepoMMan tool simply uses an adapted BDef/BMech pair from a number supplied as examples with Fedora. Their use can be seen in the listing at Appendix 2.

```
<foxml:datastream CONTROL_GROUP="E" ID="memberQuery" STATE="A" VERSIONABLE="true">
    <foxml:datastreamVersion CREATED="2007-10-25T17:22:41.820Z" ID="memberQuery.0"
LABEL="Membership query"
      MIMETYPE="text/plain" SIZE="0">
      <foxml:contentDigest DIGEST="none" TYPE="DISABLED"/>
      <foxml:contentLocation
REF="http://URL/repomman/servlet/privateCollectionQuery?objectPID=hullwf:496c"
TYPE="URL"/>
    </foxml:datastreamVersion>
  </foxml:datastream>
```

Figure 13: An example of a memberQuery datastream

The Dissemination method that the RepoMMan tool uses to gather the children of collections is the ‘List’ method. This will apply the query defined in the ‘memberQuery’ datastream against the ResourceIndex to derive all the sub-items from a collection.

An example of a complete RepoMMan collection object can be seen in Appendix 2.

The objectPID, collectionPID, and userID input variables are all built into the Collection Object in the same way as for a singular object.

The ‘memberQueryLocation’ input variable takes in the URL reference to the memberQuery text file. Because all member queries are the same for the RepoMMan tool with only the ObjectPID being different, it was seen that creating an individual text file for each collection should be unnecessary. A memberQueryServlet was defined, which simply takes the CollectionPID as an HTTP variable and outputs the query in the correct form. Every time a collection object is built within the tool the memberQueryLocation variable will be ‘http://URL/repomman/servlet/privateCollectionQuery?objectPID=hull:12c’ with the objectPID at the end.

For private collections within the tool the ‘collectionName’ variable is stored in the object Label, so that it can be used as the folder name. However when the

'buildFedoraCollectionObject' service is being used to create a root collection object (base directory for a user), the objectLabel is changed to the userID.

The service determines whether the collection being built is user-generated or a root one by the value of the Boolean variable 'rootCollection'. If it is set to true, then as described above the object label will be set to the user's ID, otherwise it will be set to the folder name supplied by the user. Using the username as the root Collection object label, allows the 'CollectionManager' process to determine whether one exists for a user or not.

Once the CollectionObject has been built the XML is returned in a string format to the BPEL process in the variable 'objectXML'.

getCollectionObjects

The 'getCollectionObjects' service is a RepoMMan service that was written to fill a gap in Fedora's functionality. It is used to retrieve the PIDs of all the child objects in a collection.

The 'getCollectionObjects' service takes the following input variables:

- collectionPID
- fedoraServerAddress
- fedoraUsername
- fedoraPassword
- defaultCollectionBDEF

Fedora provides a REST interface to run disseminators using an HTTP Request (i.e. using a web browser). The same mechanism can be used internally with a collection object. The important disseminator method for the RepoMMan tool is 'list' which is defined in the BDef adapted from a Fedora example for RepoMMan use. Invoking it in a browser it would a return a XML list of objectPID's. These ObjectPID's represent the objects that have declared themselves as children of a particular collection object through the 'RELS-EXT' datastream.

Figure 14 shows an example of XML returned from the list method when invoked through the URL:

```
'http://FedoraServiceAddress/fedora/get/hullwf:309c/hull:1002/list/'
```

Note that 'hullwf:309c' refers to the Collection object that is being queried, with 'hull:1002' being the Behaviour Definition that contains the 'List' method.

```
<?xml version="1.0" encoding="UTF-8"?>
<sparql xmlns="http://www.w3.org/2001/sw/DataAccess/rf1/result">
  <head>
    <variable name="member"/>
    <variable name="memberPID"/>
  </head>
  <results>
    <result>
      <member uri="info:fedora/hullwf:310"/>
      <memberPID>hullwf:310</memberPID>
    </result>
    <result>
      <member uri="info:fedora/hullwf:322"/>
      <memberPID>hullwf:322</memberPID>
    </result>
    <result>
      <member uri="info:fedora/hullwf:338"/>
      <memberPID>hullwf:338</memberPID>
    </result>
    <result>
      <member uri="info:fedora/hullwf:339"/>
      <memberPID>hullwf:339</memberPID>
    </result>
  </results>
</sparql>
```

Figure 14: XML returned by 'list' itemising collection members

As can be seen the XML only contains the memberPID element of the child objects, this is determined by the RDF query that the collection object uses. Figure 12, shows that the ObjectPID of the resulting items was the only object property requested.

Because all the RepoMMan collection objects use the same Member Query (albeit with differing CollectionPIDs), BMech and BDef, the process of automating the execution of the 'List' method was relatively straightforward. The 'getCollectionObjects' operation simply calls the list disseminator for the relevant collection, parses the xml and returns it to the calling BPEL process.

The input variables to the operation are probably self explanatory, however their use will be discussed. The CollectionPID variable is obviously required to know which collection is being queried. The FedoraServerAddress, Username and Password are required because to make a call to the REST interface, firstly the address of Fedora needs to be known, and because Fedora is HTTP protected, the Username and Password will be needed to authenticate. The 'defaultCollectionBDEF' is also required due to its inclusion in the REST URL, however this usually remains the same between all the Collection Objects.

The 'getCollectionObjects' returns XML array of 'collectionOutputs' which simply contains the ObjectPID's of the collection members.

This is one of the most heavily used of all RepoMMan Services, because it provides the basis for navigation around folders within the tool. When a user clicks to go into a folder, the Flex client will call the 'getCollectionItems' BPEL process, which it in turn calls the 'getCollectionObjects' to gather the objectPID's. It is also used in other BPEL processes to determine which objects already exist within a collection, for example the 'Deposit' BPEL process.

getAllCollectionObjectsChildren

The ‘getAllCollectionObjectsChildren’ service is fundamentally the same service as ‘getCollectionObjects’ however with one major difference. Instead of the service just referring to the one CollectionObject List method and returning only its sub-items, the ‘getAllCollectionObjectsChildren’ will determine if there are any collection objects in the sub-items and then recursively determine if the sub-collection objects have any children. The returning output of the service will contain all the elements of any tree structure below the collection object being queried.

As with the getCollectonObjects service, it takes the following input variables:

- collectionPID
- fedoraServerAddress
- fedoraUsername
- fedoraPassword
- defaultCollectionBDEF

The service was specifically developed for use in the ‘Delete Object’ process, due to the need to be able to delete folders and all their sub-items. The RepoMMan tool aimed to mimic the Windows functionality as much as possible, so when you delete a folder you are in turn deleting all of the folders and file objects underneath it.

Once the ‘getAllCollectionObjectsChildren’ service has recursively determined all the items in the tree it returns an array of ‘collectionOutputs’ which simply contains the ObjectPID’s.

deleteFile

The RepoMMan team chose for many reasons to store the files that the users upload “into the repository” on an external, remotely accessible file-store rather than on the Fedora server itself. The implication of this is that when an object is built for Fedora, instead of the binary content residing locally on the same server, it is referenced by a URL (see figure 11).

Fedora does not natively provide for this form of split storage. It provides the facility for referencing external content but not manipulating it. Thus if you simply delete a Fedora object, the file it references will remain on the remote store. This is in contrast to using Fedora with ‘Internally-managed Content’; in this case when you delete the object, Fedora in turn deletes the FOXML object and its binary content.

The ‘deleteFile’ service provides additional functionality for the RepoMMan tool so that when a user deletes an object (to them, a file) from their personal space, both the Fedora object and the file on remote file-store will be removed. The ‘deleteFile’ service does not concern itself with the Fedora object; this is taken care of by the ‘purgeObject’ operation from Fedora’s Management API.

The DeleteFile service takes the following Input variables:

- filename
- userID
- ftpDirectory

- ipAddress
- port
- ftpUsername
- ftpPassword

Again the input variables are self explanatory, however the architecture of the file-store needs to be briefly detailed.

The external file-store provides the area where all users' files are uploaded via FTP. The file-store has a root directory for the repository which in turn contains the root folders, identified by each user's login ID, for all RepoMMan users. For instance the root directory of the repository may be:

```
/usr/local/RepositoryFileStore
```

in which case the binary files for user '302922' will be stored under:

```
/usr/local/RepostoryFileStore/302922
```

So for example if the user uploads 'Systems-doc.pdf' into the root of their private space, the file will be stored as:

```
/usr/localRepositoryFileStore/302922/Systems-doc-
070601120100.doc'
```

(Note the addition of a date-time stamp which is invisible to the user. This allows the storage of different versions of the same binary file without an upload overwriting a previous one.)

The Delete file service will take the filename, in this case 'Systems-doc-070601120100.doc', userID, '302922', ftpDirectory '/usr/local/RepositoryFileStore/' and the FTP address and user credentials. From these details alone it is a simple process of just using FTP to delete the file.

Once the deleteFile service has deleted the file, it will return the boolean variable 'fileDeleted' to the BPEL process, set to true or false depending on the success.

getDublinCoreMetadata

The getDublinCoreMetadata service again was a RepoMMan specific requirement to gather all the metadata that had been entered into the 'DC' metadata datastream. The purpose of this operation was to enable the Metadata BPEL process to return to the user the latest stored metadata for an object (file).

The two obvious routes for implementing this would be either to download the full Fedora object, transform from it from encoded Base64Binary to Text and then parse the xml to get the DC datastream. The other option was use Fedora's REST interface (much like the method used for getCollectionObjects) to get access to the objects DC datastream xml. Obviously for speed and convenience of functionality, the latter was used for the development of the service. Figure 15 shows how the metadata is extracted from the DC datastream with an example of the REST URL.

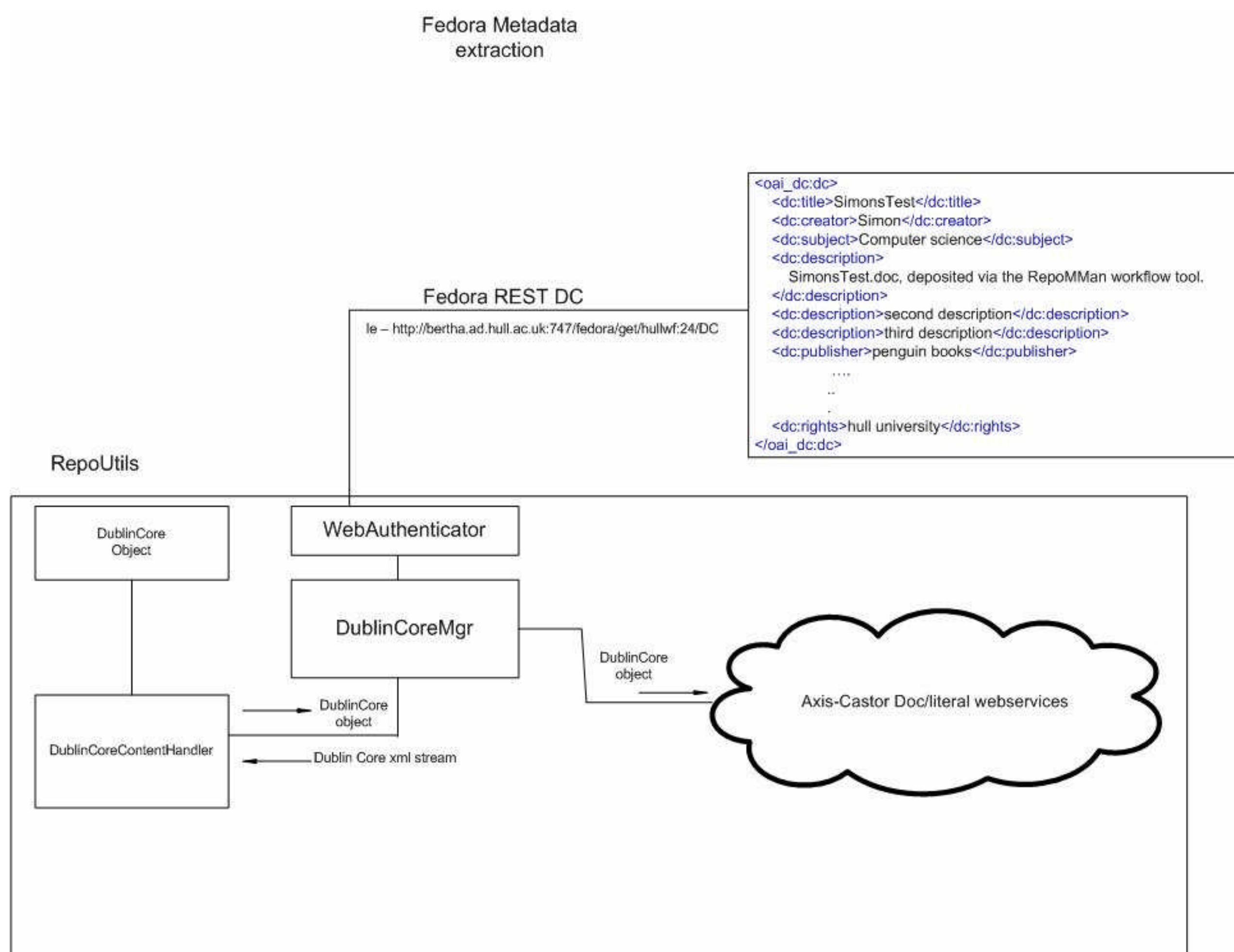

Figure 15: Extracting metadata from the DC datastream

The example shows the metadata being extracted from the URL 'http://serverAddress/fedora/get/hullwf:24/DC', this means the service is gathering it from the object PID 'hullwf:24' with the Datastream ID being 'DC'.

The 'getDublinCoreMetadata' service takes in the following input variables:-

- objectPID
- fedoraServerAddress
- fedoraUsername
- fedoraPassword

Obviously because we want the metadata attached to a particular object, the service requires the objectPID to be entered. However because the RepoMMan tool always uses the 'DC' identified datastream for the storing the metadata, the service can itself construct the required Fedora URL. Again the Fedora server address and username and password are used to make the call to Fedora, and to authenticate.

Once the service has downloaded the XML from the DC datastream, it will return the results to the calling client (in this case the BPEL process). The following is list of the metadata fields the service returns:-

- Title
- Creator
- Subject
- descriptionVersion
- descriptionCitation
- descriptionAbstract
- publisher
- contributor
- date
- type
- format
- identifier
- source
- language
- relation
- coverage
- rightsCopyright
- rightsAccessRights

The returned DC metadata includes the standard fifteen properties, with 'description' and 'rights' being repeated a number of times for extra properties within the RepoMMan metadata.

getDublinCoreXML

The 'getDublinCoreXML' service was required to give the tool the ability to update the currently existing DC datastream with an updated version. As described in the BPEL process 'Set Metadata' section, to update a Datastream of any given object within Fedora, the XML of the updated Datastream needs to be built and then encoded in Base64Binary. This service will simply build a valid Dublin Core XML datastream based upon the fields that process passes into it.

The 'getDublinCoreXML' service input variables are the exactly the same as the 'getDublinCoreMetadata' output variables, obviously because it is the reverse process. Once the service is invoked, it simply builds up an XML string with the data supplied in the input variables and returns it to the client application. Figure 16 shows an example of a valid Dublin Core metadata Datastream built by the 'getDublinCoreXML'.

```
<oai_dc:dc xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:oai_dc="http://www.openarchives.org/OAI/2.0/oai_dc/">
   <dc:title>Risks to the Public of Computer Software Report Scenario</dc:title>
   <dc:creator>Simon Lamb</dc:creator>
   <dc:subject>USS Yorktown | London Ambulance Service |  Medical Systems</dc:subject>
   <dc:description>1.0</dc:description>
   <dc:description>Computer Science</dc:description>
   <dc:description>Computer software carries risks..</dc:description>
   <dc:publisher>Publish text</dc:publisher>
   <dc:contributor></dc:contributor>
   <dc:date>18 September 2007</dc:date>
   <dc:type></dc:type>
   <dc:format></dc:format>
   <dc:identifier>hullwf:401</dc:identifier>
   <dc:source></dc:source>
   <dc:language>English</dc:language>
   <dc:relation></dc:relation>
   <dc:coverage></dc:coverage>
   <dc:rights>University of Hull</dc:rights>
   <dc:rights>Everyone</dc:rights>
</oai_dc:dc>
```

Figure 16: An example of a valid Dublin Core metadata Datastream built by the 'getDublinCoreXML'

GenerateMetadata

Part of the RepoMMan project brief was to investigate the use of tools to automatically generate descriptive metadata. As discussed earlier DataFountains provide an open-source software package that includes an advanced metadata generation tool called 'libiViaMetadata'. The iViaMetadata tool can generate contextual metadata from documents in a range of formats (currently doc, pdf and html). A few examples of this metadata includes:

- document title
- keywords
- abstract
- description
- creators

DataFountains and its family of dependencies (including iViaMetadata) are written in C++, so the main work involved in writing this service comprised of producing a Java Web service wrapper.

The iViaMetadata package itself includes a command-line tool that allows metadata to be generated by pointing at a URL to a file of appropriate type. Figure 17 shows an example of the metadata generated for the 'Hull University News page' at a particular point in time.

```
e-sig:/home/simon/libiViaMetadata-installed/bin# ./iViaMetadata-assign -T -a -c -D -k
-d http://www.hull.ac.uk/05/aboutus/news/may07/cancer.html
abstract:
creators:
date:
description:    Radiotherapy training in England is to be revolutionised following a
report published this week by the National Radiotherapy Advisory Grou p (NRAG).

key_phrases:    Radiotherapy training in England is to be revolutionised following a
report published this week by the National Radiotherapy Advisory Grou p (NRAG).
        National Radiotherapy Advisory Group
        university
        hull
        news
        radiotherapy training
        university of hull
        virtual training to revolutionise cancer treatment
        radiotherapy advisory
        national radiotherapy advisory
        radiotherapy training in england

titles:         Virtual training to revolutionise cancer treatment
        > Text version | Content as pdf | Website structure...
```

Figure 17: Metadata generated for the 'Hull University News page' at a particular point in time

The command-line tool was the basis for the integration with Java to provide the Web Service required for BPEL integration. Within the Java IO packages, there exists classes that enable command-line routines to executed from a Java object The command line tool also provided the functionality of returning the metadata results in an XML format by adding '-X', which meant parsing the results would be much simpler.

The 'GenerateMetadata' service takes the following input variable:

- contentURL

When the 'GenerateMetadata' service is invoked, the contentURL of the file that requires metadata to be generated is passed through as an input variable. Within the RepoMMan tool, this URL reference is usually the same one that exists in the Fedora object 'file' datastream. Once the service knows the contentURL, it then simply creates the correct command-line to invoke the iViaMetadata tool. For example:

```
iViaMetadata-assign -T -a -p -C -D -X -k -d -b
http://www.google.com/test.doc
```

This would then be executed and the iViaMetadata tool will return the specific metadata based upon the URL and the options chosen.

As detailed above, the service is configured to return the results in an XML form, which means it they can be simply interpreted using a Java SAX parser. Once the XML results have been parsed, they are returned back to the calling client/BPEL process. The following is an example of output variables from the 'GenerateMetadata' service:

- URL
- abstractText
- broad_subject_categories

- creators
- description
- format
- key_phrases
- language
- lcc
- rich_full_text
- titles
- proper_names_and_capitalized_phrases

Generally speaking, the service will not always return results for each of the above fields. However this very much depends on the content of the document, obviously the larger the amount of text in document the more verbose the generated metadata will be.

This service was then integrated into the 'getMetadata' BPEL process, which would call it depending on the mimeType of the content being processed.

StringUtils

The idea of the StringUtils set of services was to provide simple functionality that was specific to the needs of the RepoMMan tool and the process requirements. Although the BPEL specification includes some String functions (including concat, substring, contains etc.) sometimes these were not sophisticated enough, and so very simple Java services were written to do the job.

This section will briefly discuss the separate operations to complete the discussion on RepoMMan services.

EncodeString and DecodeString

The 'EncodeString' and 'DecodeString' services provide the functionality to take a String and encode it as Base64Binary, or conversely to decode a Base64Binary String.

At the time of developing the initial services for 'Depositing' materials into the Repository, it was quickly discovered that the requirement to encode a String into Base64Binary was required (Fedora FOXML1.0 objects are required in Base64Binary on ingest). At the time it seemed reasonable to assume that such a thing may exist in the feature set of the BPEL String functions or indeed a service available online. Unfortunately this was not the case, so the RepoMMan team decided to develop one themselves.

The idea of the service was to take any string and encode it into the Base64Binary and return it as a string (see figure 18). This could then be used to encode the FOXML objects prior to ingest, and also employed for other 'Update' processes.

Normal string:

'This is a string'

Base64Encoded string:

VGhpcyBpcyBhIHN0cmluZw==

Figure 18: Base 64 binary encoding

The Java code behind the services is very simple, however it plays an important role within the 'Deposit' and 'SetMetadata' processes where non-encoded strings cannot be used.

The Encode/Decode operations take an input of 'inString' and return the output of 'outString'.

getFilenameFromURL

The URL location of an object's binary content is passed around quite frequently within the RepoMMan BPEL processes. The Deposit process itself only requires the 'contentLocation' as of one its input parameters, so therefore small string utilities are required to derive more information from it.

One such utility is 'getFilenameFromURL', this operation will simply take a URL that points to a file (i.e. http://www.test.co.uk/Document.txt) and return the Filename that it references (in this case Document.txt).

RemoveFilenameTimeStamp

With the need for versioning of the files within the RepoMMan tool, date-time stamps are added to the filenames of deposited files when uploaded to the remote filestore. This process allows multiple versions of the 'same' binary content, with the same user filename, to be held without an upload to the file store over-writing the previous version.

There is sometimes a requirement within a BPEL process to get the name of the file, without the filestamp appended. So for instance, we may know the file on the filestore was named 'ChemicalReactions-070606144221.doc' but want the filename that appears on the user's home computer i.e. ChemicalReactions.doc. Unfortunately, again the string functions within our BPEL implementation did not provide the complexity required to achieve this, however a simple line of Java would.

Again the RemoveTimeStamp operation is a very simple String manipulation service that takes in a string stamped filename and returns an unstamped filename.

RemoveFileExtension

The RemoveFileExtension service was born out of the requirement to have the name of the file without the file extension. For example when an object is being

deposited, the title of the object can be set with the name of file without its extension.

Again this was a requirement that the BPEL string functions did not provide, so a trivial service was written 'RemoveFileExtension'.

Testing

As all software should, the RepoMMan tool underwent testing at a number of levels to ensure that it did it what was expected of it and also performed in a way that was acceptable. Because the RepoMMan tool contains a number of distinct layers, i.e Java, Web Services, BPEL etc, it was necessary to undertake the appropriate methodologies at each stage.

At the bottom layer of the RepoMMan stack is the Java code that undertakes the bulk of the work, i.e. building objects, communicating to file store, parsing xml etc... A unit testing methodology was used at this stage, to check that each of the methods and algorithms developed behaved as expected. The Java testing was undertaken using the Eclipse Java Development framework.

A level above the Java business logic are the Web Services that expose the functionality to remote clients and, in the RepoMMan tool, the BPEL processes. Developing Web Services brings in the added complications of writing the WSDL and XML Schema files; this requires different methods of testing. It is important to validate that the methods exposed as Web Service operations are doing the right thing, but it also necessary to verify that the WSDL and XML schemas are valid in there own right. To undertake this level of testing, SoapUI 1.7 provides the ability to call Web Services with a range of input data to test that they return the correct output data. It also provides the ability to validate the SOAP packet structure, i.e. the WSDL and XML schema format. This type of testing was very important when having to integrate the Web Services into other software such as BPEL, Flex etc. Many software packages are very 'picky' when it comes to validity of Web Services.

Once the Web Services and Java code underneath them had been tested fully, the BPEL processes which consume them were validated and verified. The ActiveBPEL Designer tool is a user-friendly GUI interface for orchestrating Web Services, it also provides a nice way of the testing the process behaviour. After a BPEL process has been developed, the Designer tool provides 'Simulation' routines. This facility allows you to simulate with test variables a run through the BPEL process with the ability to pause, step-over et This was a very important part of the testing, as it allowed a range of data to be passed into the simulation and therefore errors could be determined without having the 'physically' execute the process. For each of the RepoMMan processes developed significant simulated executions were undertaken as part of developing and testing.

Once the BPEL processes were deployed onto the ActiveBPEL Server, SoapUI could be used to invoke and therefore test them. Obviously because the processes were simulated many times it with the ActiveBPEL Designer, it was expected that they would behave as expected. However to test the processes with 'real' data, it was a standard procedure to validate them using the SoapUI tool.

As the RepoMMan tool was seen as software that would integrate into many users' day-to-day workflow, performance was almost as important as reliability

and needed to be tested. Again because the bulk of the processing is orchestrated by the BPEL processes, it was seen that testing performance of the tool should revolve around them. The SoapUI tool again provided the important functionality to achieve this: the ability to load test specific services with a range of data. Much like the standard way of testing a Web Service, SoapUI provided the ability to pass sample data to service and then record the response time. Added to this was the ability to execute multiple threads against the BPEL processes at regular time intervals for a given amount of time. All of the BPEL processes were subject to this sort of load testing and were invoked with a range of different threads and time intervals. It was found that even under quite strenuous load, the processes' performance remained at an impressive level. Overall all of the processes response times were more than acceptable for the likely usage within the University's proposed system

Usability testing was undertaken with a range of users both within and without the University. The comments received from testers were used to inform the iterative development, and hopefully improvement, of the interface.

Appendix 1

FOXML for an example file object


```
<?xml version="1.0" encoding="UTF-8"?>
<foxml:digitalObject PID="hullwf:485"
  fedoraxsi:schemaLocation="info:fedora/fedora-system:def/foxml#
http://www.fedora.info/definitions/1/0/foxml1-0.xsd"
  xmlns:audit="info:fedora/fedora-system:def/audit#"
xmlns:fedoraxsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:foxml="info:fedora/fedora-system:def/foxml#">
  <foxml:objectProperties>
    <foxml:property NAME="http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
VALUE="FedoraObject"/>
    <foxml:property NAME="info:fedora/fedora-system:def/model#state"
VALUE="Active"/>
    <foxml:property NAME="info:fedora/fedora-system:def/model#label"
VALUE="system-documentation.doc"/>
    <foxml:property NAME="info:fedora/fedora-system:def/model#ownerId"
VALUE="302922"/>
    <foxml:property NAME="info:fedora/fedora-system:def/model#createdDate"
VALUE="2007-10-10T10:00:33.517Z"/>
    <foxml:property NAME="info:fedora/fedora-
system:def/view#lastModifiedDate" VALUE="2007-10-10T10:00:33.517Z"/>
  </foxml:objectProperties>
  <foxml:datastream CONTROL_GROUP="X" ID="RELS-EXT" STATE="A"
VERSIONABLE="true">
    <foxml:datastreamVersion CREATED="2007-10-10T10:00:33.517Z" ID="RELS-
EXT.0"
      LABEL="Fedora Object-to-Object Relationship Metadata"
MIMETYPE="text/xml" SIZE="292">
      <foxml:contentDigest DIGEST="none" TYPE="DISABLED"/>
      <foxml:xmlContent>
        <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rel="info:fedora/fedora-system:def/relations-external#">
          <rdf:Description rdf:about="info:fedora/hullwf:485">
            <rel:isMemberOf rdf:resource="info:fedora/hullwf:368c"/>
          </rdf:Description>
        </rdf:RDF>
      </foxml:xmlContent>
    </foxml:datastreamVersion>
  </foxml:datastream>
  <foxml:datastream CONTROL_GROUP="E" ID="file" STATE="A"
VERSIONABLE="true">
    <foxml:datastreamVersion CREATED="2007-10-10T10:00:33.517Z" ID="file.0"
LABEL="system-documentation.doc"
      MIMETYPE="application/msword" SIZE="0">
      <foxml:contentDigest DIGEST="none" TYPE="DISABLED"/>
      <foxml:contentLocation REF="http://150.237.54.112/users/302922/system-
documentation-071010112629.doc" TYPE="URL"/>
    </foxml:datastreamVersion>
  </foxml:datastream>
  <foxml:datastream CONTROL_GROUP="X" ID="DC" STATE="A" VERSIONABLE="true">
    <foxml:datastreamVersion CREATED="2007-10-10T10:00:33.517Z" ID="DC1.0"
LABEL="Dublin Core Metadata"
      MIMETYPE="text/xml" SIZE="707">
      <foxml:contentDigest DIGEST="none" TYPE="DISABLED"/>
      <foxml:xmlContent>
        <oai_dc:dc xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:oai_dc="http://www.openarchives.org/OAI/2.0/oai_dc/">
          <dc:title/>
          <dc:creator/>
          <dc:subject/>
          <dc:description/>
          <dc:description/>
```

```
          <dc:description/>
          <dc:publisher/>
          <dc:contributor/>
          <dc:date>2007-10-10</dc:date>
          <dc:type/>
          <dc:format>application/msword</dc:format>
          <dc:identifier>hullwf:485</dc:identifier>
          <dc:source/>
          <dc:language/>
          <dc:relation/>
          <dc:coverage/>
          <dc:rights/>
          <dc:rights/>
        </oai_dc:dc>
      </foxml:xmlContent>
    </foxml:datastreamVersion>
    <foxml:datastreamVersion CREATED="2007-10-10T10:00:33.517Z" ID="DC.1"
LABEL="Dublin Core Metadata"
      MIMETYPE="text/xml" SIZE="327">
      <foxml:contentDigest DIGEST="none" TYPE="DISABLED"/>
      <foxml:xmlContent>
        <oai_dc:dc xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:oai_dc="http://www.openarchives.org/OAI/2.0/oai_dc/">
          <dc:title>system-documentation</dc:title>
          <dc:identifier>hullwf:485</dc:identifier>
          <dc:subject>system-documentation.doc</dc:subject>
          <dc:description>system-documentation.doc</dc:description>
        </oai_dc:dc>
      </foxml:xmlContent>
    </foxml:datastreamVersion>
  </foxml:datastream>
</foxml:digitalObject>
```

Appendix 2

FOXML for an example collection object


```
<?xml version="1.0" encoding="UTF-8"?>
<foxml:digitalObject PID="hullwf:496c"
  fedoraxsi:schemaLocation="info:fedora/fedora-system:def/foxml#
http://www.fedora.info/definitions/1/0/foxml1-0.xsd"
  xmlns:audit="info:fedora/fedora-system:def/audit#"
xmlns:fedoraxsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:foxml="info:fedora/fedora-system:def/foxml#">
  <foxml:objectProperties>
    <foxml:property NAME="http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
VALUE="FedoraObject"/>
    <foxml:property NAME="info:fedora/fedora-system:def/model#state"
VALUE="Active"/>
    <foxml:property NAME="info:fedora/fedora-system:def/model#label"
VALUE="Test"/>
    <foxml:property NAME="info:fedora/fedora-system:def/model#ownerId"
VALUE="302922"/>
    <foxml:property NAME="info:fedora/fedora-system:def/model#createdDate"
VALUE="2007-10-25T17:22:41.820Z"/>
    <foxml:property NAME="info:fedora/fedora-
system:def/view#lastModifiedDate" VALUE="2007-10-25T17:22:41.820Z"/>
  </foxml:objectProperties>
  <foxml:datastream CONTROL_GROUP="E" ID="memberQuery" STATE="A"
VERSIONABLE="true">
    <foxml:datastreamVersion CREATED="2007-10-25T17:22:41.820Z"
ID="memberQuery.0" LABEL="Membership query"
      MIMETYPE="text/plain" SIZE="0">
      <foxml:contentDigest DIGEST="none" TYPE="DISABLED"/>
      <foxml:contentLocation
REF="http://150.237.4.35/repomman/servlet/privateCollectionQuery?objectPID=h
ullwf:496c" TYPE="URL"/>
    </foxml:datastreamVersion>
  </foxml:datastream>
  <foxml:datastream CONTROL_GROUP="X" ID="RELS-EXT" STATE="A"
VERSIONABLE="true">
    <foxml:datastreamVersion CREATED="2007-10-25T17:22:41.820Z" ID="RELS-
EXT.0"
      LABEL="Fedora Object-to-Object Relationship Metadata"
MIMETYPE="text/xml" SIZE="293">
      <foxml:contentDigest DIGEST="none" TYPE="DISABLED"/>
      <foxml:xmlContent>
        <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rel="info:fedora/fedora-system:def/relations-external#">
          <rdf:Description rdf:about="info:fedora/hullwf:496c">
            <rel:isMemberOf rdf:resource="info:fedora/hullwf:321c"/>
          </rdf:Description>
        </rdf:RDF>
      </foxml:xmlContent>
    </foxml:datastreamVersion>
  </foxml:datastream>
  <foxml:datastream CONTROL_GROUP="E" ID="memberList.xml" STATE="A"
VERSIONABLE="true">
    <foxml:datastreamVersion CREATED="2007-10-25T17:22:41.820Z"
ID="memberList.xml.0" LABEL="member list"
      MIMETYPE="text/xml" SIZE="0">
      <foxml:contentDigest DIGEST="none" TYPE="DISABLED"/>
      <foxml:contentLocation
REF="http://local.fedora.server/fedora/get/PID/hull:1002/list" TYPE="URL"/>
    </foxml:datastreamVersion>
  </foxml:datastream>
  <foxml:datastream CONTROL_GROUP="E" ID="viewStylesheet.xsl" STATE="A"
VERSIONABLE="true">
```

```
    <foxml:datastreamVersion CREATED="2007-10-25T17:22:41.820Z"
ID="viewStylesheet.xsl.0" LABEL="Stylesheet"
      MIMETYPE="text/xml" SIZE="0">
      <foxml:contentDigest DIGEST="none" TYPE="DISABLED"/>
      <foxml:contentLocation
REF="http://local.fedora.server/repomman/behaviours/repomman-docs-
viewstylesheet.xsl" TYPE="URL"/>
    </foxml:datastreamVersion>
  </foxml:datastream>
  <foxml:datastream CONTROL_GROUP="X" ID="DC" STATE="A" VERSIONABLE="true">
    <foxml:datastreamVersion CREATED="2007-10-25T17:22:41.820Z" ID="DC1.0"
LABEL="Dublin Core Metadata"
      MIMETYPE="text/xml" SIZE="238">
      <foxml:contentDigest DIGEST="none" TYPE="DISABLED"/>
      <foxml:xmlContent>
        <oai_dc:dc xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:oai_dc="http://www.openarchives.org/OAI/2.0/oai_dc/">
          <dc:title>Test</dc:title>
          <dc:title/>
          <dc:identifier>hullwf:496c</dc:identifier>
        </oai_dc:dc>
      </foxml:xmlContent>
    </foxml:datastreamVersion>
    <foxml:datastreamVersion CREATED="2007-10-25T17:22:41.820Z" ID="DC.1"
LABEL="Dublin Core Metadata"
      MIMETYPE="text/xml" SIZE="245">
      <foxml:contentDigest DIGEST="none" TYPE="DISABLED"/>
      <foxml:xmlContent>
        <oai_dc:dc xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:oai_dc="http://www.openarchives.org/OAI/2.0/oai_dc/">
          <dc:title>Test</dc:title>
          <dc:identifier>hullwf:496c</dc:identifier>
          <dc:description>302922</dc:description>
        </oai_dc:dc>
      </foxml:xmlContent>
    </foxml:datastreamVersion>
  </foxml:datastream>
  <foxml:disseminator BDEF_CONTRACT_PID="hull:1002" ID="DISS1" STATE="A"
VERSIONABLE="true">
    <foxml:disseminatorVersion BMECH_SERVICE_PID="hull:1003" CREATED="2007-
10-25T17:22:41.820Z" ID="DISS1.0" LABEL="Dissem.">
      <foxml:serviceInputMap>
        <foxml:datastreamBinding DATASTREAM_ID="memberList.xml" KEY="LIST"
LABEL="Binding to member list" ORDER="0"/>
        <foxml:datastreamBinding DATASTREAM_ID="memberQuery" KEY="QUERY"
LABEL="Binding to Membership query" ORDER="0"/>
        <foxml:datastreamBinding DATASTREAM_ID="viewStylesheet.xsl"
KEY="XSLT" LABEL="Binding to Stylesheet" ORDER="0"/>
      </foxml:serviceInputMap>
    </foxml:disseminatorVersion>
  </foxml:disseminator>
</foxml:digitalObject>
```