Web Services for Greenstone 3

A dissertation
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Computing and Mathematical Sciences
at the
University of Waikato
by
ANUPAMA KRISHNAN

University of Waikato

2008

Abstract

The project on web services for Greenstone 3 was primarily implementation oriented and consisted of three parts. This project was primarily implementation oriented and consisted of three parts. The first involved designing and implementing a set of enhanced web services for Greenstone 3. The second part required implementing a Greenstone 3 demo-client that would make use of the web services in order to access a repository backed by Greenstone 3. While Greenstone 2 had a demo-client that allowed end-users to perform tasks comparable to those possible through a Greenstone powered browser, a similar client had not yet been built for Greenstone 3. The final part of the project was to demonstrate functional interoperability by also enabling access to a Fedora-backed repository of Greenstone-collection documents from within Greenstone 3's Java-based client application.

There were design decisions pertaining to each part of the project. When it came to the web services, the questions were on what parts of Greenstone's functionality to map into web services and what method definitions would be easiest for clients, in terms of the types of parameters and return values and the most suitable way to deal with optional parameters. The design issues surrounding the Java-based client application were far more straightforward as they were centred around how best to organise the classes and how to add support for Fedora and still ignore the details of which digital library is being used at any one time. Designing the third part of the project was more intensive. The Greenstone documents required custom representation in Fedora's repository, in order to provide the same information as in Greenstone. It was decided to build a stand-alone component that could interface with Fedora's repository on one end and with Greenstone 3's Java-client on the other. This meant that the intermediate component needed to know about Fedora's Access web services in order to access the repository, while on the other end, it needed to mimic the protocol with which Greenstone communicates with the Java-client. As such, this component—FedoraGS3—acted as a translator: converting requests for data sent from the client into web service calls to access Fedora, and converting data returned by Fedora into response messages that the client would understand. Finally, the FedoraGS3 component's functionality was integrated successfully into the Java-client, just like the Greenstone 3 web services it used to access the Greenstone repository's contents.

Evaluating the three parts of this project showed up some limitations in the web services, which suggested changes to their design: some methods were removed while others were added. The new set of web services still requires further evaluation, before we can determine whether its API has met the goal of being general yet useful enough. Due to some slight differences between the way data is represented in Fedora and the way it is represented in Greenstone, some digital library services offered for Fedora—such as querying and browsing—worked a bit differently or required extra effort to bring it closer to Greenstone's functioning. Though complete functional interoperability was not achieved, we had significant success. The Java-client was also successful in that it was able to emulate Greenstone 3's browser interface to a considerable extent and serves as a good demonstration of a Java-based client for Greenstone 3 and how one may be built.

Acknowledgements

With many thanks to my supervisors, Dr David Bainbridge, Dr Dave Nichols and Professor Ian Witten, and my family.

Table of Contents

List of Tables and Figures

Chapter 1. Introduction

1.1. Background

Greenstone is an open-source, cross-platform, general-purpose digital library system that allows users to construct digital library collections which can function in a distributed environment. Witten and Bainbridge cover the origins and development of Greenstone over its first decade in [17]. Its development started around 1996 as the New Zealand Digital Library (NZDL) project. It was designed with multilingual support in mind and supports different media content (text, audio, images, and more). Initially, the focus was to make Greenstone a tool to produce and distribute humanitarian information collections, though it required software expertise in order to be able build new collections. Over time, the aim expanded to one of making it easy for librarians and other end-users who might not be IT specialists to create and distribute collections with Greenstone. This led to the development of the Greenstone Librarian interface, an application for designing and building collections and enriching the contents with metadata. Greenstone has been used to produce humanitarian information collections for the Human Info NGO and UNESCO that have been distributed widely. A major Greenstone project involved digitising Maori language newspapers from which the *Niupepa* collection was created and which has been made available online.

Greenstone allows for interoperability through its:

- document and multimedia plugins that are used to ingest many recognised data formats into its repository, as well as other digital library systems' formats, like DSpace;
- plugouts, used to export Greenstone contents into a variety of standardised formats, like METS and MARCXML. Greenstone can also convert its collection data into formats specific to other digital library systems, such as DSpace and FedoraMETS which can be ingested by DSpace and Fedora, respectively;
- use of established protocols such as the Open Archives Initiative Protocol for Metadata Harvesting (OAI-PMH) and z39.50.

In spite of the degree to which Greenstone promotes interoperability with other digital library softwares, Witten and Bainbridge noted that there had been little interaction between different digital library projects in terms of learning from each other's open-source research and incorporating available solutions.

1.1.1. Interfaces to Greenstone

Greenstone provides two distinct interactive interfaces [17]:

- The above-mentioned Greenstone Librarian Interface (GLI), a Java-based application that is a powerful tool to design and create information collections. It enables one to add documents into a collection, import collections, export them into different media and formats, design metadata sets, assign and edit metadata and perform other tasks related to constructing a digital library. While supporting recognised metadata

formats, Greenstone also allows end-users the freedom to design collections with metadata sets that are specific to their needs.

- The Reader interface, a Greenstone-powered browser interface that allows users access to the repository content. Digital libraries built with Greenstone—where metadata forms a prominent part of the construction—are fully searchable. The Reader enables users to carry out tasks such as searching Greenstone collections and browsing them by various categories.

1.1.2. Greenstone 3

Bainbridge, Don, et al. [3] describe the considerations that gave rise to Greenstone 3 and its redesigned architecture. To summarise, Greenstone's earlier architecture had some limitations due to its design. Dynamic configuration was not supported by the runtime system, which had to go offline to take changes to settings into account. Client-server interaction was facilitated by the CORBA protocol, with which applications could be built that made use of Greenstone functionality. For instance, Greenstone 2's Java-based client offered users an application interface through which they could accomplish tasks similar to what they could do through the Reader, its browser interface. The underlying communication between this client application and Greenstone 2 proceeded via the CORBA protocol. However, there were disadvantages to using a CORBA-based protocol for client-server communication in Greenstone. Minor changes to the API of operations provided would require significant corresponding changes elsewhere.

The developers considered a new design for Greenstone based around requirements that included backwards compatibility, support for dynamic configuration (at runtime, versus the server having to go offline), a service-centred, distributed and modular architecture where the modules can send machine-readable messages such as for describing their capabilities to clients. The Greenstone 3 project started off as a research framework that would ensure compatibility with Greenstone 2, while keeping up-to-date through adoption of improvements in technology. It was a redesign of Greenstone's architecture and was now implemented in Java while still providing all of Greenstone 2's C++ capabilities. Greenstone 2 continues to be developed concurrently due to its prolific use around the world.

Bainbridge, Don, et al. then detail how Greenstone 3 overcomes a lot of the earlier difficulties, enabling dynamic configuration of collections and services, and the addition of new content. Its architecture, that makes this possible, is organised in a modular fashion, where the network of modules represents a digital library. Communication between the modules is achieved by passing request and response messages in XML, such as a 'describe yourself' message. Such a design also enabled remote communication in a distributed set-up, as XML messages can be passed between remote modules using the Simple Object Access Protocol (SOAP) which is used in web services. As such there was already some rudimentary web service support that could allow client applications to communicate with a remote Greenstone server by sending XML request messages.

Even though there was some soap-based web service support in Greenstone 3, it did not yet provide a set of convenient-to-use web service operations that developers of client applications might find easier to invoke.

1.2. Aims

The general aims of this project were to provide enhanced web services for the Greenstone 3 digital library software and build a demo-client application for Greenstone 3 that would also demonstrate functional interoperability with another digital library system.

1.3. Overview of Greenstone 3

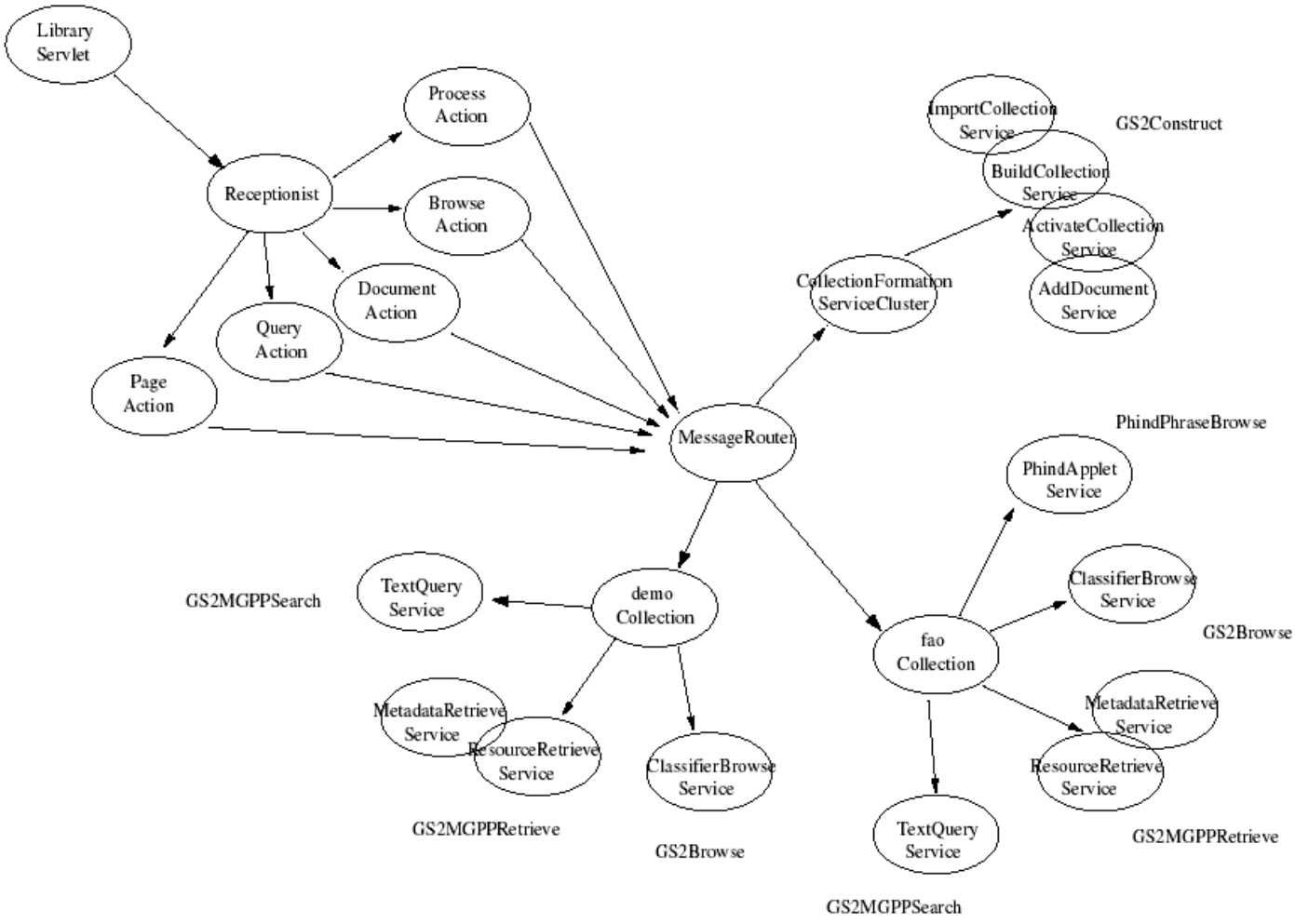


Figure 1.1: "A simple stand-alone site" in Greenstone 3. Image reproduced from "Greenstone3: A modular digital library" [8]

Greenstone 3 consists of a front-end, which handles the user interface side of things, and the back-end, which is the server (called a *site* in Greenstone). In Figure 1.1, the Library Servlet and the Receptionist module with its Actions form the front-end. The other modules are part of the server (the two Collections along with their services, the services comprising the Collection Formation Service Cluster and the Message Router). A Service Cluster groups together related services—in this case the services concern tasks related to creating collections [3].

Greenstone 3's central module is the *Message Router*, which receives requests sent from the user interface end. Its main method is `process()` which takes a string that represents the XML *request message* to be sent to Greenstone's core, and returns the *response message* which is also an XML string. A single XML request message sent to `process()` could encapsulate different types of requests. The Message Router will receive them all and sort out the rest. Some of the incoming requests are handled by the Message Router itself, whereas more specific ones are delegated to the Greenstone server's specialist modules. The latter send their response back to the Message Router which, in its turn, returns a response to the client. The Message Router is therefore the only interface to Greenstone's server end which the outside world need concern itself with, and its `process()` method serves as the primary point of contact.

The extant web service for Greenstone 3 consists of a single operation—*also* called `process`—which is a direct mapping of the Message Router's central method of the same name. Thus, the existing web service operation effectively exposes all of the externally-accessible Greenstone functionality that is available through the Message Router.

The Greenstone 3 Developer's Manual [8] explains in detail the structure of the various request messages that can be sent to Greenstone 3 and of the response messages that are returned. It is therefore possible for any programmer who has worked through the Developer's Manual to make use of the original web service operation when they have become familiar with the format of the XML messages being passed to-and-fro, as they then know how to formulate the request messages. However, even though the `process` web service is already sufficient where access to all the Message Router's capabilities is concerned, it is rather general. Web service operations that map to more specific, individual aspects of Greenstone's functionality, and which would be easier to invoke, could therefore be a useful addition.

1.4. Objectives

This project comprised three specific objectives. These were:

1. To provide a set of web services for Greenstone 3 that are general enough, yet useful. This involved:
   - determining what aspects of Greenstone 3's functionality to map into web services;
   - deciding on the kind of input parameters and return values for the web service methods.

The web services originally available for Greenstone 3 offered a single but important operation. They exposed the Message Router's central `process()` method as a web service, thus giving developers of client applications direct access to Greenstone 3's core functionality.

The intent behind this project's first objective was to provide a set of enhanced web services. Although they would ultimately be giving access to the same functionality, the aim was to design their methods in such a way that invoking them might at times be a more convenient alternative. As such, it entailed designing the method signatures of the web service operations.

2. To implement a client application in Java that could demonstrate one way of presenting the Greenstone 3 functionality that is available through a browser interface in a GUI-based application. There were several reasons for building a client application for Greenstone 3. Greenstone 2 already had a demo-client exhibiting some of its capability from within a GUI interface. We wished to show that a client application could be built for Greenstone 3 and to demonstrate one of several ways in which this could be accomplished. (The Greenstone 3 Developer's Manual [8] outlines the possibilities.)

   A further objective was to let the client make use of the web services for Greenstone 3 in order to access the necessary functionality. This would simultaneously serve as a means to evaluate the first part of the project.

3. To demonstrate interoperability with another digital library software by allowing the client application to also access a Fedora repository of Greenstone content, next to its working with a Greenstone-backed repository. This part of the project involved the following steps:
   - Greenstone collection and document data needed to be represented and stored in a custom format in Fedora;
   - deciding on the sorts of operations that would be necessary in order to work with a Fedora repository containing Greenstone documents. They would need to provide access to information that is specific to Greenstone content;
   - creating an intermediate component that would convert the Greenstone data returned from accessing the Fedora-backed repository into the format understood by the client application (which was primarily designed to work with Greenstone-backed repositories), in order to integrate it successfully with the same.

The second and third objectives have been accomplished: we have successfully implemented a demo-client for Greenstone 3 that is also able to interact with a Fedora-backed repository of Greenstone content to provide end-users with a similar experience. Yet, while web services for Greenstone 3 have been designed, implemented and improved upon after some simple evaluation, it has been hard to estimate the degree of success in their design. Other, more rigorous methods of evaluation—such as practical use cases—would go a long way towards determining how complete and useful the web services that have now been provided are.

Chapter 2. System review

2.1. Digital Library Systems

In this chapter we review different digital library systems, with emphasis on Fedora which we worked with extensively in this project in addition to Greenstone.

2.1.1. DSpace

MIT and HP Labs' DSpace [13] is open source digital library software that runs on Linux and which is used for creating institutional repositories to manage and preserve publications and research papers. These are then indexed for searching and browsing, and distributed on the web. Content can be in various formats (text and multimedia). DSpace provides a web-based interface to submit material to be included in the repository. Users upload the content along with descriptive metadata. Since the purpose of this digital library system is very specific, it works with a predefined metadata set.[1] Only the Dublin Core metadata set is supported[2], of which only a few fields are compulsory. OAI-PMH is supported in DSpace, which is a data provider in that it exposes its metadata for harvesting. Its Manakin project allows users to customise the look-and-feel of their DSpace installation.[3] Manakin is "an abstract framework for building repository interfaces that currently provides an implementation for DSpace." [12] Three constructs have been developed to achieve this:

- The Digital Repository Interface (DRI) XML Schema that provides an abstract representation of a repository page (containing both the structural data and metadata).
- Aspects, which are stand-alone components built with Apache Coccoon, which modify existing features or provide new ones for a repository. Aspects take DRI documents as input and produce the same as output allowing for 'aspect chaining' whereby many aspects can be combined into the final page that is generated by incorporating one aspect after another.
- Themes, that apply XSL stylesheets to DRI documents to provide a specific look-and-feel to a repository, collection or community.

2.1.2. EPrints

EPrints is an OAI-compliant, multi-lingual, open source archive system for building digital repositories that are Open Access, particularly institutional repositories and multimedia-based collections.[4] "Open Access (OA) is free, immediate, permanent online access to the full text of research articles for anyone, webwide."[5] The EPrints repository software runs on Linux. Its repository can store data of various formats (from documents to multimedia), comes with a depositing interface that allows users to add their content

[1] http://www.dspace.org/index.php?option=com_content&task=blogcategory&id=44&Itemid=156
[2] http://www.dspace.org/index.php?option=com_content&task=blogcategory&id=40&Itemid=88#standards
[3] http://wiki.dspace.org/index.php//Manakin
[4] http://www.eprints.org/
[5] http://www.eprints.org/openaccess/

into the repository, and provides browsing and searching capability [14]. The recent EPrints Version 3 makes more complex browsing possible. It also offers enhanced interoperability: the facility to export data and features (like the results of a search) to various standard Digital Library formats like METS and Dublin Core, web services like Google Earth and a number of bibliography managers [6].

Web services for EPrints are in the works, and a tentative API listing is viewable.[6] Though the web services will not be part of the next EPrints release, they will be available for download separately.

2.1.3. Fedora

Fedora is software that allows one to design and build digital libraries and repositories of *digital objects* (such as documents, images and other content). It is much more general than Greenstone or DSpace, as it allows users to decide on what functionality is necessary and implement it themselves in order to construct a customised digital library with its own services. Custom operations can be tied to the digital objects on which they act. Next to its own internal metadata set, it has built-in support for Dublin Core metadata, of which some elements are compulsory or are otherwise automatically generated. However, the general-purpose design of Fedora allows users to include other kinds of metadata as well. Fedora 2.2.1 does not have built-in full-text indexing and search capabilities. That functionality is implemented separately by *Fedora Generic Search*, which will be discussed further on in this section.

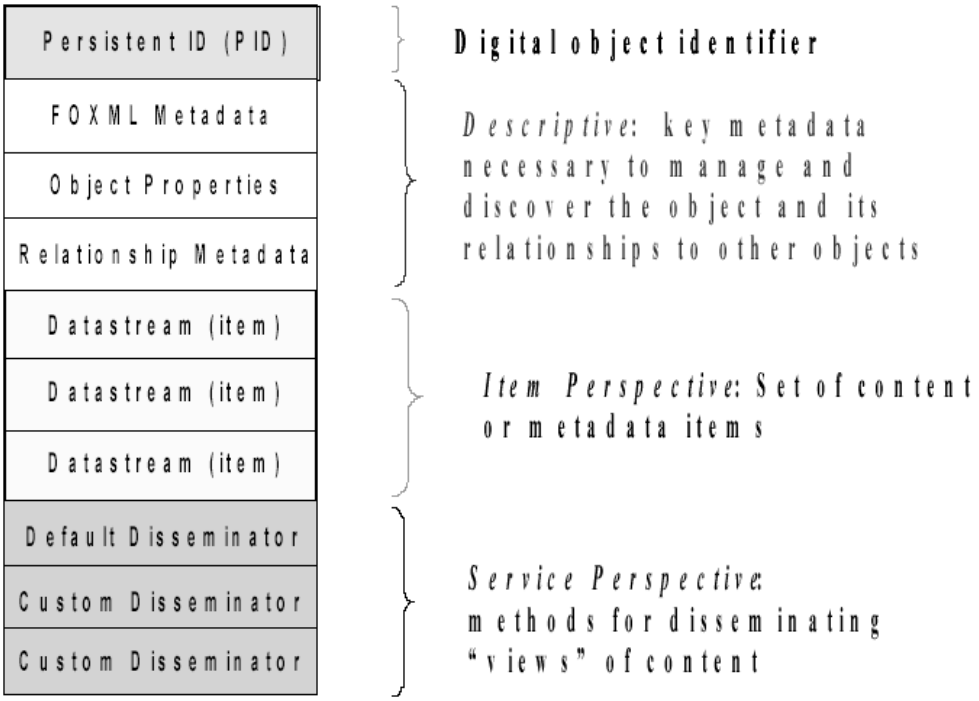


Figure 1.2: "Fedora Digital Object architectural overview", reproduced from Fedora Tutorial #1 [15].

[6] http://wiki.eprints.org/w/Web_Services

Next to its REST-interface (web services that are accessible in a browser via URLs), Fedora also exposes its public functionality using SOAP, a protocol for data-exchange that is used by web services. Its SOAP-based web services are split over the following two APIs[7]:

- the Management API, that deals with managing the Fedora objects stored in its repository. Operations such as ingesting new content into the repository and purging existing digital objects from it are part of the Management API;
- the Access API, which is concerned with providing access to the data that have already been ingested. It includes operations such as searching on the default Fedora-assigned DC metadata of a digital object, or retrieving the contents of a stored object's *datastreams*. Datastreams give access to the individual data cotnents of a digital object. For instance, if a document made up of text and associated images is represented by a single digital object in Fedora, the text and images constitute its datastreams. Figure 1.2 shows the parts that make up a digital object in Fedora, including datastreams.

Behavioural Definitions and Behavioural Mechanisms

Fedora does not merely allow one to store data in its repository as digital objects, but also to define a set of operations that are applicable to a whole class (or collection) of related digital objects. Such method definitions are called Behavioural Definitions or BDef in Fedora and are represented by digital objects themselves. A BDef merely defines a method and the MIME type of the input and of the output—acting rather like a Java interface definition in that respect. The actual implementation(s) of a BDef are provided by web services whose locations in the form of URLs are stored in a separate Fedora digital object called a Behavioural Mechanism or BMech. A BDef can be implemented by one or more different web services, each referenced by a BMech. The web service for a BMech takes a Fedora digital object of the input MIME type specified in the associated BDef and will transform this dynamically to produce content of the output MIME type also defined in that same BDef.

Through BDefs and BMechs, Fedora allows users to create repositories that store digital objects' data in one (internal) format, while giving them the capability to dynamically convert that data to other formats, thus producing different representations of the same content. For instance, a BMech may apply one or more XSLT files to several XML input files to produce HTML files as output. Another BMech could be applied to turn the XML files into plain text. Thus, different views of the same digital object are possible even though the repository stores only a single copy (in XML, in this example) of its actual data content [10].

The structure of a Fedora digital object

For each digital object stored in the repository, Fedora allows one to access and view:

- the Object's Profile, which contains the object's PID (persistent identifier) in Fedora.

[7] http://www.fedora.info/definitions/1/0/api/

- its Disseminations. These are the different views of a digital object's content that are generated by the object's *disseminators*—the operations that can be performed on its content. These operations are considered part of a digital object and tied to its data as depicted in Figure 1.2. The disseminations page contains the default behaviours that Fedora automatically provides for each digital object, as well as any custom Behavioural Definitions and Behavioural Mechanisms peculiar to it or a set of similar digital objects. Default behaviours include generation of the Object Profile page, the ItemIndex page (see below) and the Disseminations page.
- the ItemIndex. This is a list of all the datastreams associated with the digital object. These datastreams can be either static ones—meaning they are stored—or otherwise they are dynamically created by the BMechs in its Disseminations.

These three views can also be seen through Fedora's REST interface. A digital object's Object Profile page, viewable through the REST interface, gives access to both the Disseminations and ItemIndex pages.

Fedora Generic Search

Fedora does not come with built-in search services to index the text-based contents of datastreams, whether full-text or user-added metadata. Rather, it only provides the ability to search the Fedora-generated metadata fields—such as the PID field—and the compulsory DC datastream of a digital object. As a consequence of the lack of full-text searching, Fedora does not provide web services for such search functionality either. However, separate components can be written to provide the same. *Fedora Generic Search* (FedoraGSearch) [11] is one such external component that provides full-text indexing and searching for Fedora repositories. Its operations are also made available as web services upon installation.

2.2. Web services

Web services enable distributed computing: communication and exchange of data over the Internet. It is built on open standards, such as various XML formats. The use of XML in data exchange is advantageous since XML is a textual format that can represent any data (including data structures). Web services are therefore independent of the programming language or platform in which the client operates and vice-versa [7]. They also promote code reuse: if a web service already contains the code to execute a task that one's own application needs, then there's no need to rewrite it oneself.

Examples of web services include online currency converters, world clocks and weather databanks. Client applications can build their functionality on top of those provided by existing web services and can either stop there or turn this new composite functionality into web services of their own for other clients to use.

One of the protocols used in web services for the exchange of data is the XML-based SOAP format (Simple Object Access Protocol), where XML messages encapsulating the data—called SOAP messages—are passed between remote applications on a network, even where these applications reside behind firewalls. SOAP over HTTP makes messaging over the web possible. Input and output data are encoded as SOAP for

transfer. SOAP-based web services offer functionality (operations) that remote applications can invoke by passing data in the form of XML.

The Web Services Description Language (WSDL), which is also XML-based and therefore machine-readable, describes the location of a web service, the operations it offers and the types of the parameters and return values of those operations. Applications that wish to invoke a particular web service can download its published WSDL, look up the service-endpoint (URL location) of the web service, and invoke the required operation as specified in the WSLD file [7].

Although security is an issue when it comes to web services—for instance, security has not been built into SOAP, yet it can bypass firewalls—an additional protocol layer can be added on top to minimise the usual dangers inherent in data exchange over the Internet or other networks. Solutions to ensure web services security include encryption and digital signatures, and fall under the categories of authentication (that you are indeed who you say you are) and authorisation (that you are recognised as someone who has the right to access specific data). For example, anyone with a hotmail account can *authenticate* themselves in hotmail with their username and password. However:

- the same combination may not authenticate them in Gmail;
- their username and password do not authorise their access to someone else's hotmail account, even though hotmail knows who they are. This is because hotmail's login procedure only *authorises* them to view their own mail account

Beyond the existing security measures used to transmit data over HTTP, new XML technology standards have been developed to enable authentication and authorisation for web services. These standards include XML Key Management Specification (XKMS), Security Assertion Markup Language (SAML) and Extensible Access Control Markup Language (XACML) [7].

2.2.1. The role of web services

A great many companies have started making their functionality available through web services:

- Google provides the Google Web APIs. For instance, there's Google's Search API which client applications can invoke to provide custom search facilities. Though its SOAP Search web services are no longer being developed, Google offers the Ajax Search API instead. Client applications can embed search forms in their interface and use it to customise search results on their page.
- Amazon, whose web service APIs include Alexa Web Search. There's also the Amazon cloud web services which allow end-users to rent vast amounts of hardware (to perform intensive calculations, for instance).[8]

Among the advantages of all this is the fact that expert code is made available for public use—in some cases for free, at other times for a fee. Google's Search API would have been optimised for speed and would have been tested. Incorporating the available functionality, where appropriate, could save developers time and effort. Next to

[8] "The Death of Hardware" by Quentin Hardy for Forbes, 02/11/08, http://www.forbes.com/technology/forbes/2008/0211/036.html

functionality, Amazon cloud is giving clients the use of a *resource*—hardware. Developers can forego trying to acquire vast quantities of hardware for themselves, especially when they might only need this for a one-off project or for a short period of time.

One role web services can play in digital libraries is to make their expert functionality and data—such as Greenstone's search services and publicly hosted collections—available for external client applications to embed, build on or access, without clients having to install the software or keep local copies of the data. For example, a developer might need to implement a client to provide a unique visual representation of search results retrieved from a particular Greenstone collection hosted online. Underlying their code would be invocations to the remotely hosted Greenstone collection's query service for executing the searches.

Web services would allow communication between a digital library system and a client that are operating on different platforms and using different (web service-enabled) programming languages. XML allows the client to remain ignorant about the details of location or implementation of the digital library system whose web services are being called. For instance, while Greenstone 3 is implemented in Java, a client application invoking its web services could be written in Perl, which also has support for web services. This means that developers of clients for Greenstone 3 do not need to write in Java to instantiate a Message Router instance in order to get access to a Greenstone repository and do custom processing. Neither do they have to download and include Greenstone source code to compile their client with.

2.3. Interoperability

The widely-used OAI-PMH is a protocol that digital library systems can follow to expose their metadata, which may otherwise be stored internally in a custom archival format, and to build services that make use of others' metadata.[9] Though the protocol specifies a uniform way for various digital libraries to make their metadata available for harvesting, it does not specify a similarly uniform way to expose other repository content. That is, it does not provide for a way where document content stored in a system-specific internal format can be harvested so that it can be accessed by other digital library systems. OAI-PMH is concerned with interoperability of metadata. Since repositories tend to store more than metadata, this does not provide complete interoperability between different digital library systems even where those concerned include support for OAI-PMH [16]. However, OAI has been developing the Object-Reuse and Exchange (ORE) specification that is intended to allow a common representation of digital objects in order to facilitate inter-repository exchange of and access to content.[10]

As mentioned earlier on, Greenstone supports interoperability by being able to import content stored in the formats of certain other digital library softwares, and by being able to export Greenstone documents into other standardised or specific formats like Fedora

[9] http://www.openarchives.org/OAI/openarchivesprotocol.html
[10] http://www.openarchives.org/ore/

and DSpace. For instance, the StoneD tool was crafted to convert Greenstone content and metadata into DSpace's format and to convert DSpace content and metadata into Greenstone [16]. It is available through the Greenstone Librarian Interface. More recently, a tool for exporting from Greenstone into FedoraMETS, a format ready for ingesting into Fedora, has been written as well. It is also available through the GLI. Witten and Bainbridge [17] have observed how Greenstone's "importing interface can be used to facilitate document exchange and interoperability between various systems: for example, going from DSpace to Fedora and back again—without involving Greenstone at all!" Bainbridge, Kaun-Yu, et al. describe how they have expanded the capabilities of the GLI tool to act as a "document and metadata exchange center", using precisely the importing plugins and exporting plugouts of Greenstone. Interoperability between digital library systems may therefore also be functional, in which case digital libraries could use the operational capabilities provided by another digital library system, without necessarily accessing the repository contents. Web services can contribute to such interoperability as well, in that the publishing of a digital library system's functionality is essentially giving external applications—including other digital libraries—a way to access services (and through them, the content, if required) of a different repository system.

2.3.1. The Simple Digital Library Interoperability Protocol (SDLIP)

In Stanford's SDLIP protocol, clients can send search requests to repositories over HTTP or CORBA transports. Clients can directly send their requests to any repositories that support SDLIP. However, repository systems need not implement SDLIP themselves. In such a case, another component—a Library Service Proxy—can act as an intermediary between client and repository and provide SDLIP support. On the proxy's client-end, it implements the SDLIP protocol for returning responses, but where it connects to one or more repositories, it could be dealing with different protocols specific to each of them. Incoming SDLIP search requests are translated by the proxy and passed onto the repository, while the data returned are translated back into SDLIP format and sent to the client [1].

Bainbridge, Witten, et al. [2] explain how by implementing the SDLIP protocol using CORBA, Greenstone 2 was able to interact with an SDLIP client. An intermediate component, here called the Translation server, was constructed that understood both SDLIP and Greenstone 2's own protocols. The component intercepted requests from clients and converted them into Greenstone 2's protocol calls. The data returned from Greenstone was then converted into the SDLIP protocol format and sent to the client in response.

The third objective of this project, outlined in Section 1.4, is concerned with letting our Greenstone 3's client application access a Fedora repository. Though it does not involve SDLIP, here a similar solution was used where an intermediate component does the work of translating between Fedora and the client application that expects Greenstone's request-response protocol.

Chapter 3. An extended worked example

This chapter will look at how the implemented Greenstone 3 Java-client works, including the manner in which users can make a repository (Greenstone or Fedora) active, and how they can subsequently perform tasks like choosing the collection they wish to work with, executing queries, viewing search results and browsing by available categories.

Figure 3.1: Screenshot of the Java-client's start-up dialog requesting the location of the WSDL file for Greenstone 3's web services

Figure 3.2: Screenshot of the start-up dialog requesting initialisation information required for connecting to Fedora's repository

Upon starting the Greenstone 3 Java-client application, a dialog requests the user to input the URL of the WSDL file of the Greenstone 3 web services, as seen in Figure 3.1. In the first instance this location is set assuming Greenstone is installed and running locally in the default location, with the user free to edit this as they wish. The requested WSDL file is needed to establish a connection with the Greenstone 3 web services, which the Java-client will subsequently use to access Greenstone's functionality and repository.

Once the user clicks OK—or has finished modifying the default text to point to the correct WSDL location—another dialog appears, requesting the user to submit the information necessary to connect to the local Fedora repository. The requested information consists of the host and port of the Fedora server, and its administration username and password as can be seen in Figure 3.2.

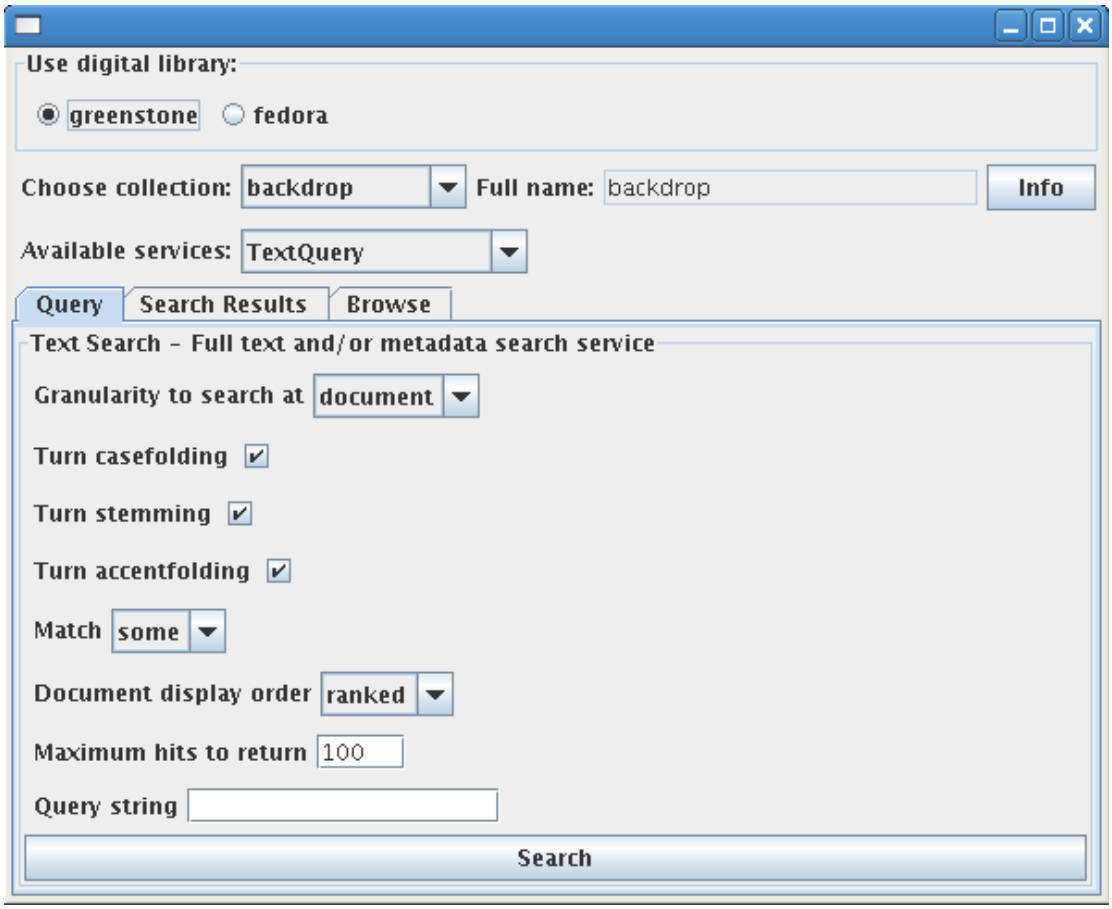


Figure 3.3: Screenshot of the Java Client's main interface

In the event that neither connection can be made (for instance, because the user pressed cancel in both connection dialogs, or if they entered invalid data or due to some other difficulty such as the Fedora server not running yet), the client application will exit. If a connection to one of the digital libraries was established successfully, or if both connections were successfully made, then the Java-client application window opens displaying its main interface, as shown in Figure 3.3. This window consists of two sections:

- The top section is where the user can choose which digital library they want to make active—Greenstone or Fedora (they can switch between the two throughout). Here they can also select the collection they wish to work with and then the service they want to make use of.
- The central portion of the window is the activity section containing the Query, Search Results and Browse tabbed panes.

In Figure 3.3, Greenstone is selected as the currently active digital library.

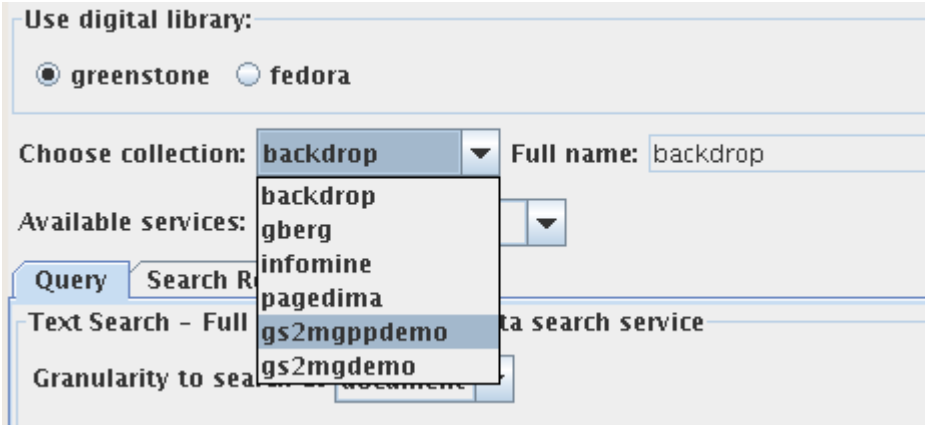


Figure 3.4: Portion of the main interface showing the available collections in the Greenstone repository

Once the user has chosen which repository they want to work with, the list of collections in that repository is retrieved and displayed in the *Collections* drop-down box.

In Figure 3.4 the user has clicked on the *Collections* drop-down box which shows Greenstone's 'short name' next to it for each available collection. These are Backdrop, Gutenberg, Infomine, Paged Images, GS2MGDemo (a demo collection built with Greenstone 2 that uses Managing Gigabytes for compressing and indexing) and GS2MGPPDemo which stands for "Greenstone 2 Managing Gigabytes Plus Plus".

The Greenstone Digital Library software allows end-users to create collections with short names and full display names as well as provide short descriptions for them. This information is also viewable in the client application. If the selected collection has a full name then this is displayed in the *Full Name* field beside it, otherwise the short name shown in the *Collections* drop-down box is visible here. When the button labelled *Info* and located next to the *Full Name* field is pressed, a small dialogue will appear containing the brief textual description of the collection, if one is available.

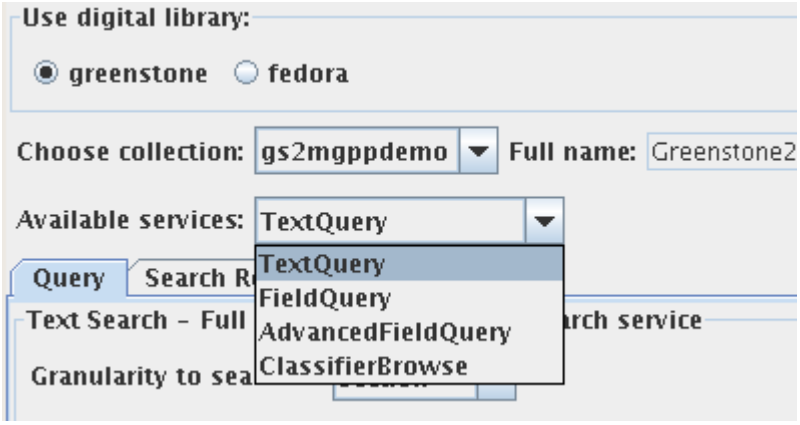


Figure 3.5: Portion of the main interface showing the services available for the selected GS2MGPPDemo collection

Upon choosing the collection they wish to access, a list of all the query and browse services that are offered by the selected collection is dynamically uploaded and shown in the *Services* drop-down box (Figure 3.5).

At present all the available services for a collection are displayed in the drop-down, though the only services that allow the user to actually work with them upon selection are the Query-type services (Text Query, Field Query and Advanced Field Query) and Browse-type services (Classifier Browse). Some other services in the list—such as the Retrieve-type services for retrieving document structure, metadata and content and for retrieving metadata for browsing classification hierarchies—are either invoked upon executing Query-type and Browse-type services or they do not do anything in the Java-client.

One of the available collections is GS2MGPPDemo. The services in this collection that users can interact with include the three Query services and the one Browse service shown in the screenshot of Figure 3.5. The others do nothing upon selection.

3.1. The Query pane

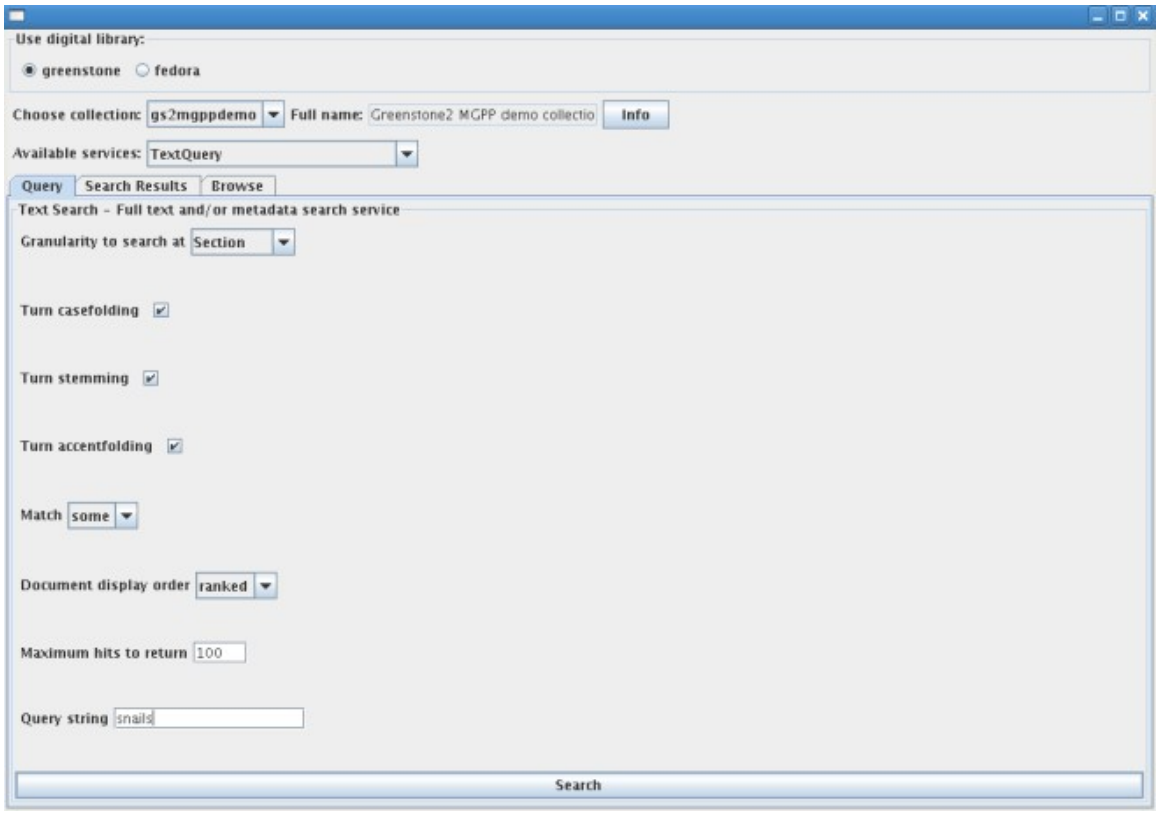


Figure 3.6: Screenshot of a form generated for the Text Query service of Greenstone 3.

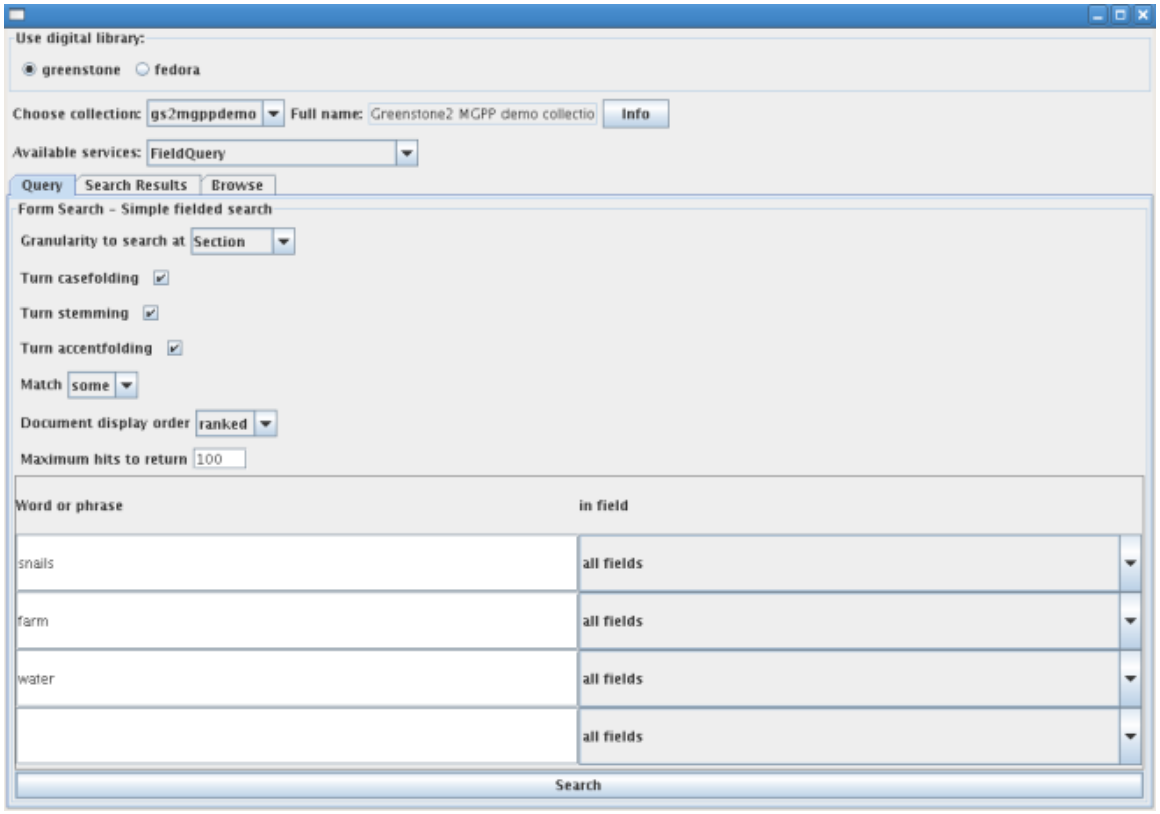


Figure 3.7: Screenshot of Greenstone's Field Query form.

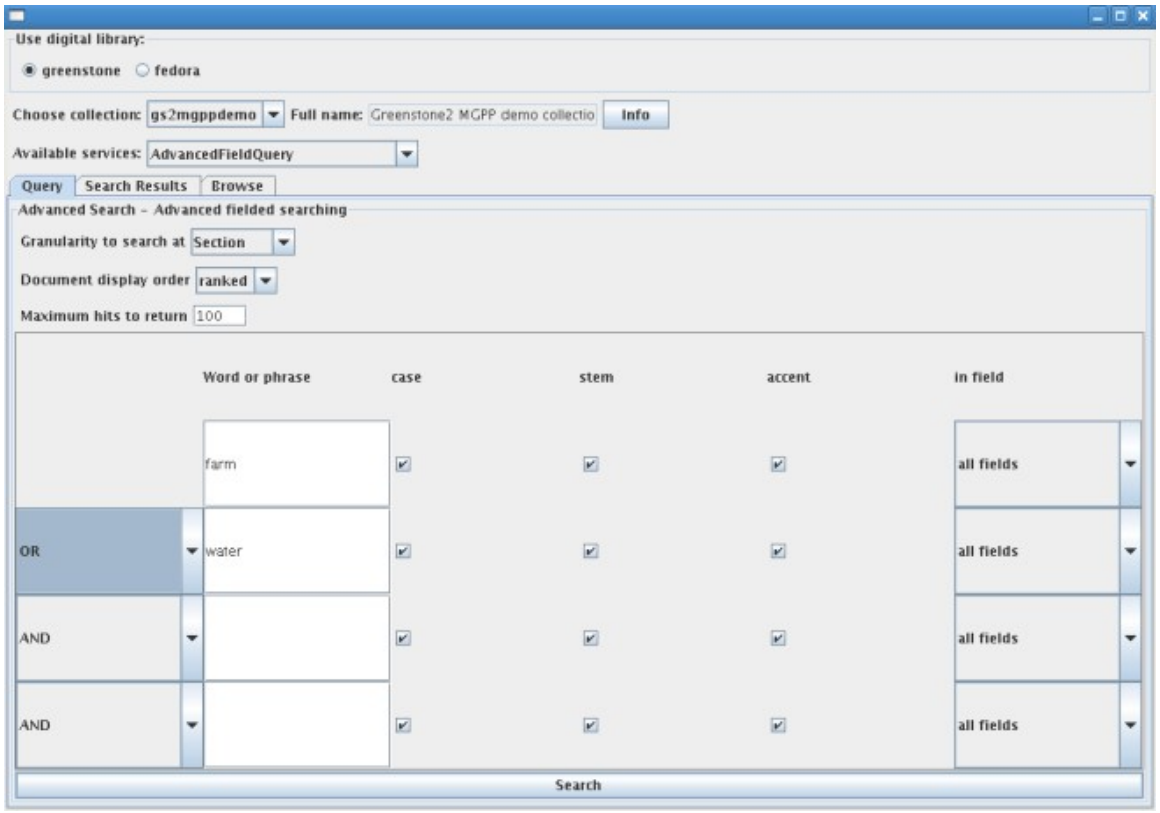


Figure 3.8: Screenshot of Greenstone's Advanced Field Query form.

Choosing any of the Query services in the *Services* drop-down box will bring the *Query tab* into focus. When the active digital library is Greenstone, the Text Query will be available for collections and, depending on the collection, possibly Field Query and Advanced Field Query. This query functionality is provided by Greenstone for its collections. (The particular Query services offered for a collection depend on which ones were chosen when creating the collection in Greenstone.)
When a Query service is selected, the *Query tab* will contain a form whose controls are entirely determined by Greenstone—as communicated to the Java-client in response to a describe request for the Query service. It is due to Greenstone 3's describe operation for query services that the Java-client knows to produce query forms similar to the search forms of Greenstone's browser interface.

Figure 3.9: Screenshot of the Greenstone 3 browser interface's Text Query form.

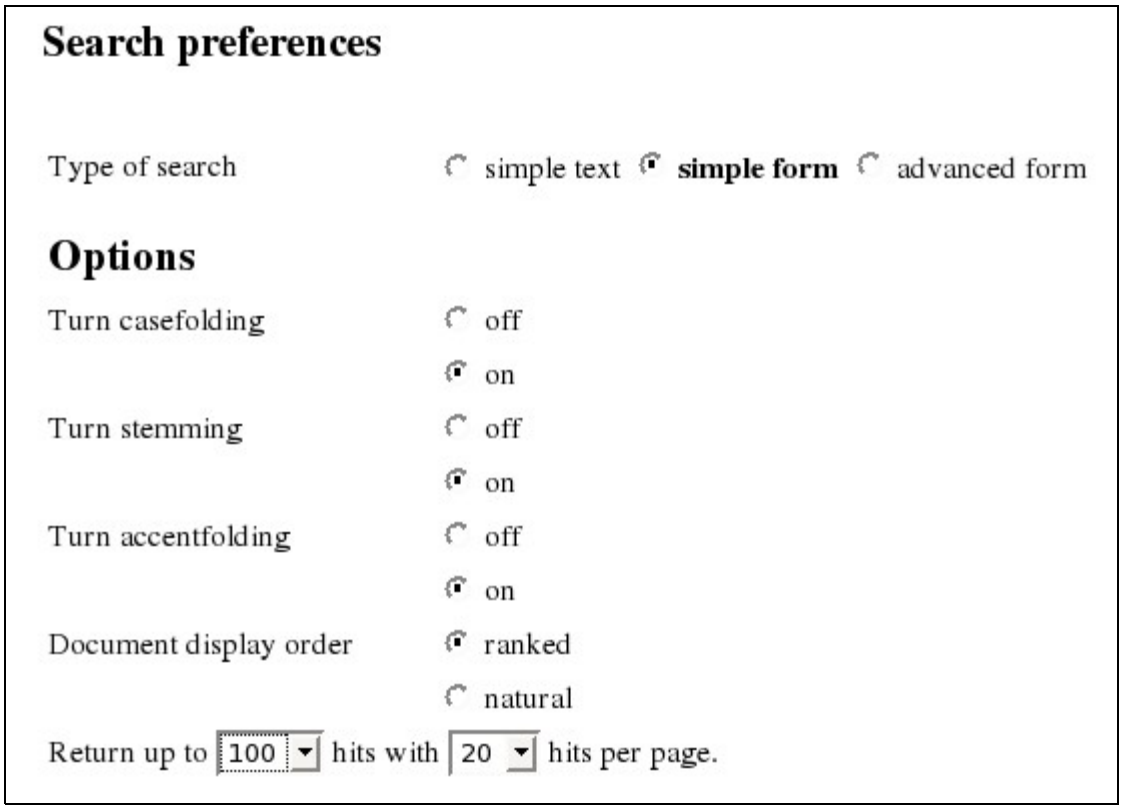


Figure 3.10a: Screenshot of the search form control settings available for the Greenstone 3 browser interface's Field Query of the GS2MGPPDemo collection through the Preferences page.

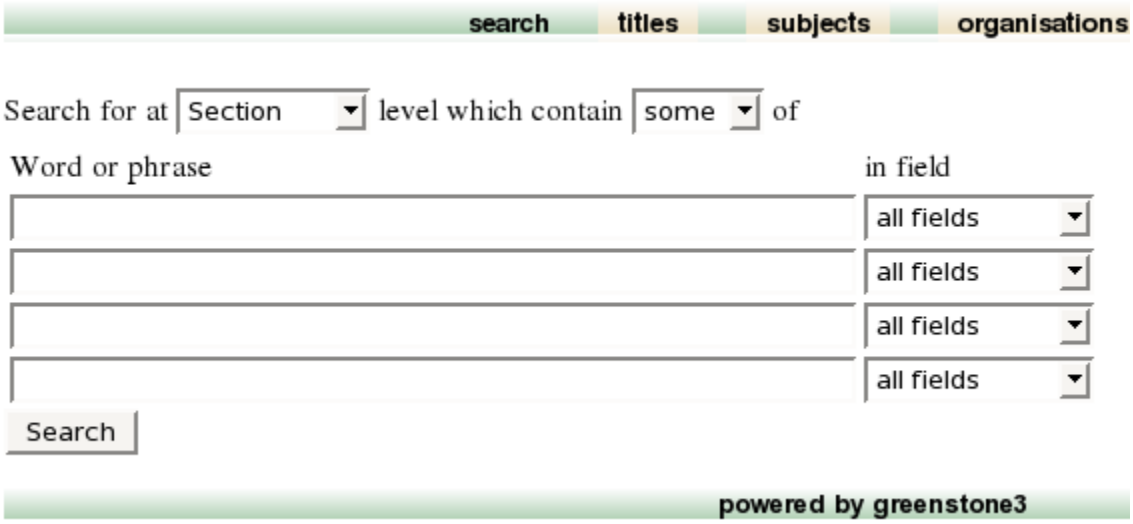


Figure 3.10b: Screenshot of the Greenstone 3 browser interface's Field Query form.

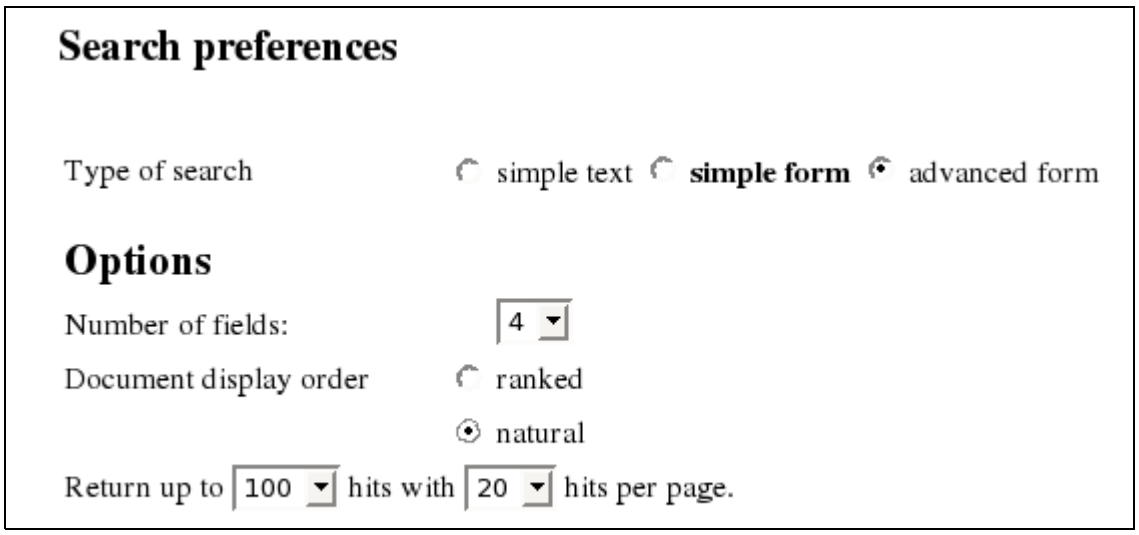


Figure 3.11a: Screenshot of the search form control settings available for the Greenstone 3 browser interface's Advanced Field Query of the GS2MGPPDemo collection through the Preferences page.

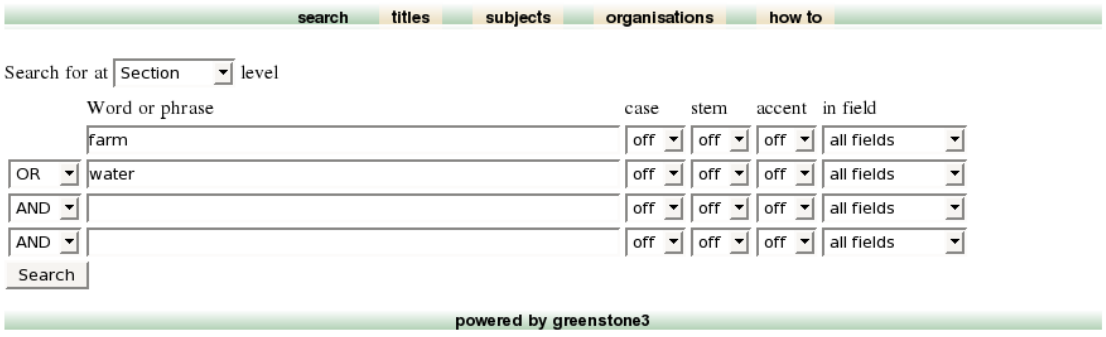


Figure 3.11b: Screenshot of the Greenstone 3 browser interface's Advanced Field Query form.

In Figures 3.6, 3.7 and 3.8 one can see the forms the Java-client generates for Greenstone's Text Query, Field Query and Advanced Field Query services that are available for the GS2MGPPDemo collection. Some of these forms contain multiple fields to enter search terms into, while Advanced Field Query allows the user to decide what Boolean operators to apply when combining the term fields. These are similar to the Greenstone 3 browser interface's Query forms. A user can set the properties of the various forms' controls just as they do in the Preferences menu of the Greenstone's browser interface, including specifying whether they want to turn on case stemming and case folding, what portion of a document to restrict searches to, and how many documents to retrieve at maximum. Figures 3.9, 3.10a, 3.10b, 3.11a and 3.11b are screenshots of the Greenstone browser interface's equivalent search forms and of some of the adjustments that can be made for them through the Preferences page, presented here for comparison.

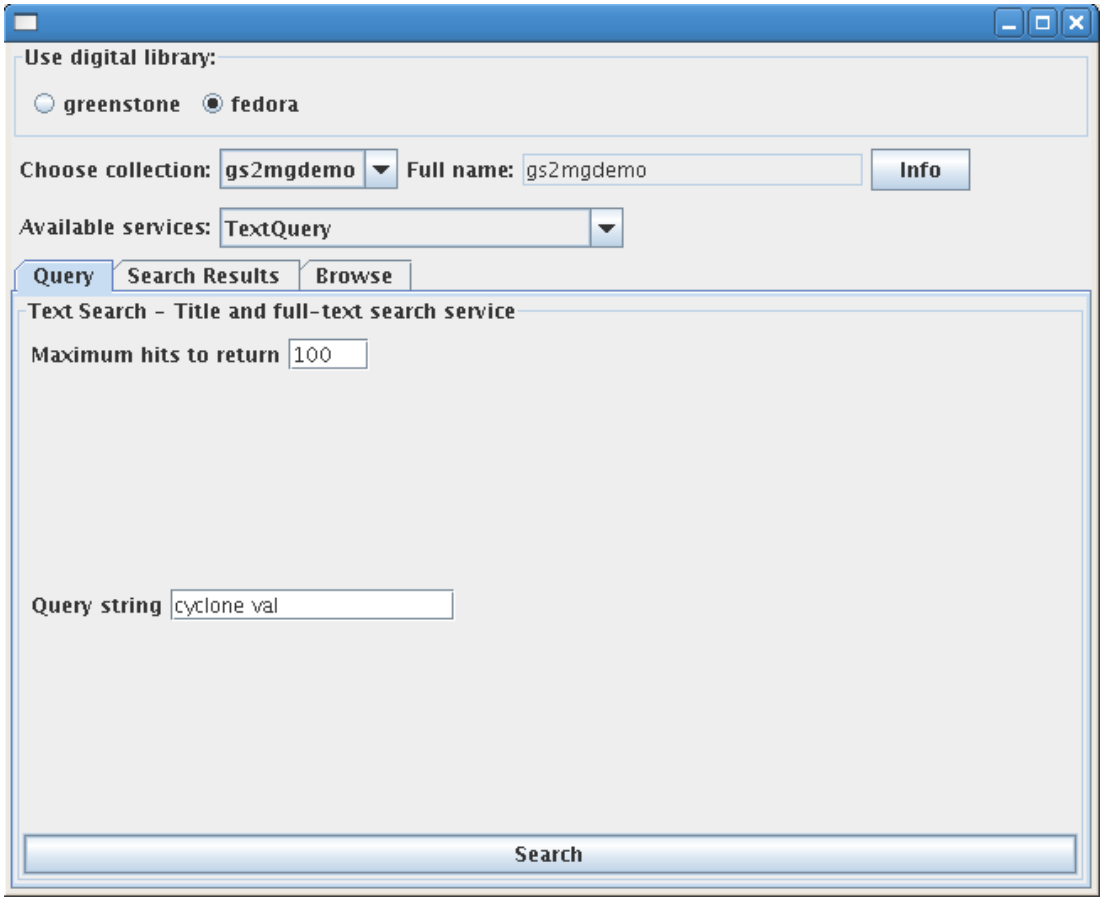


Figure 3.12: The Text Query form available to users when searching Fedora's repository

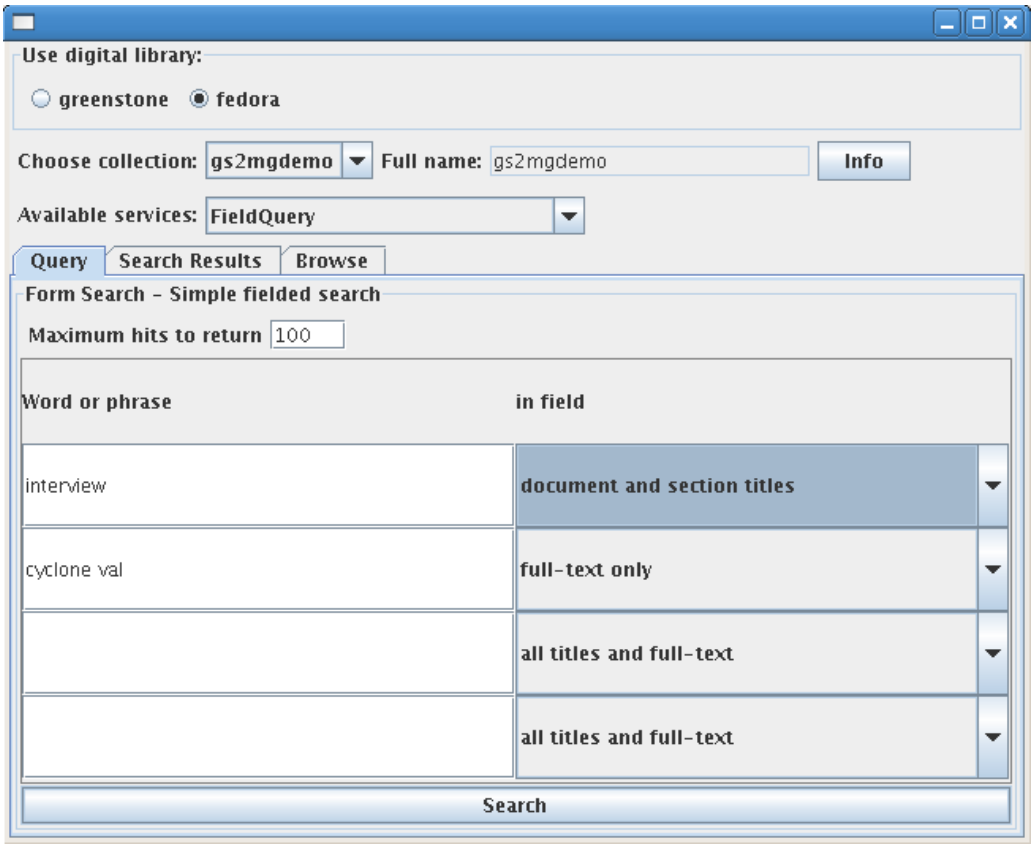


Figure 3.13: The Field Query form available to users when searching Fedora

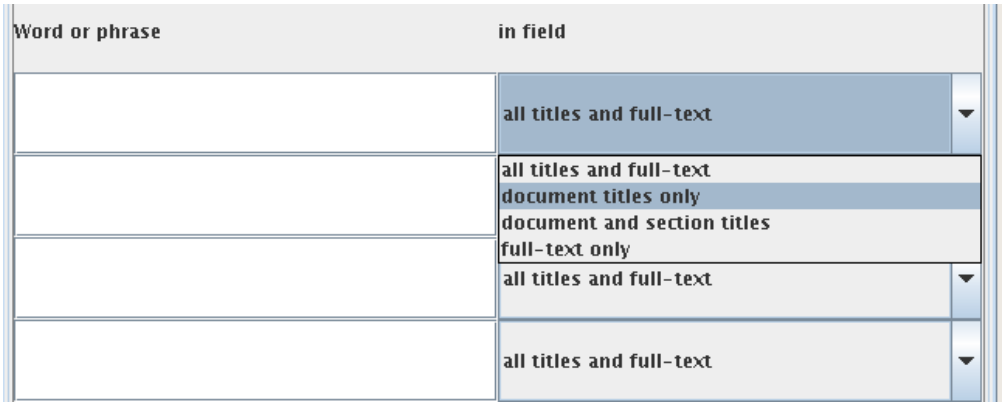


Figure 3.14: The available field search options in Fedora's Field Query form

Full-text searching capability for Fedora was enabled by installing the separate component *Fedora Generic Search* and using its web services. Fedora Generic Search made it possible to search section-level titles as well, while document-level titles could be queried using Fedora's API-A web services. A combination of these underlies the Text and Field Query services presented in the client application, which can be seen in Figures 3.12 and 3.13. Both these query services are available to all Greenstone collections stored in the Fedora repository (these are the collections that have been exported from Greenstone and ingested into Fedora). The Field Query form generated for searching

Fedora's repository offers the user the ability to perform queries on document titles, document and section titles, full-text or in all three of these indexed fields. This is shown in Figure 3.14.

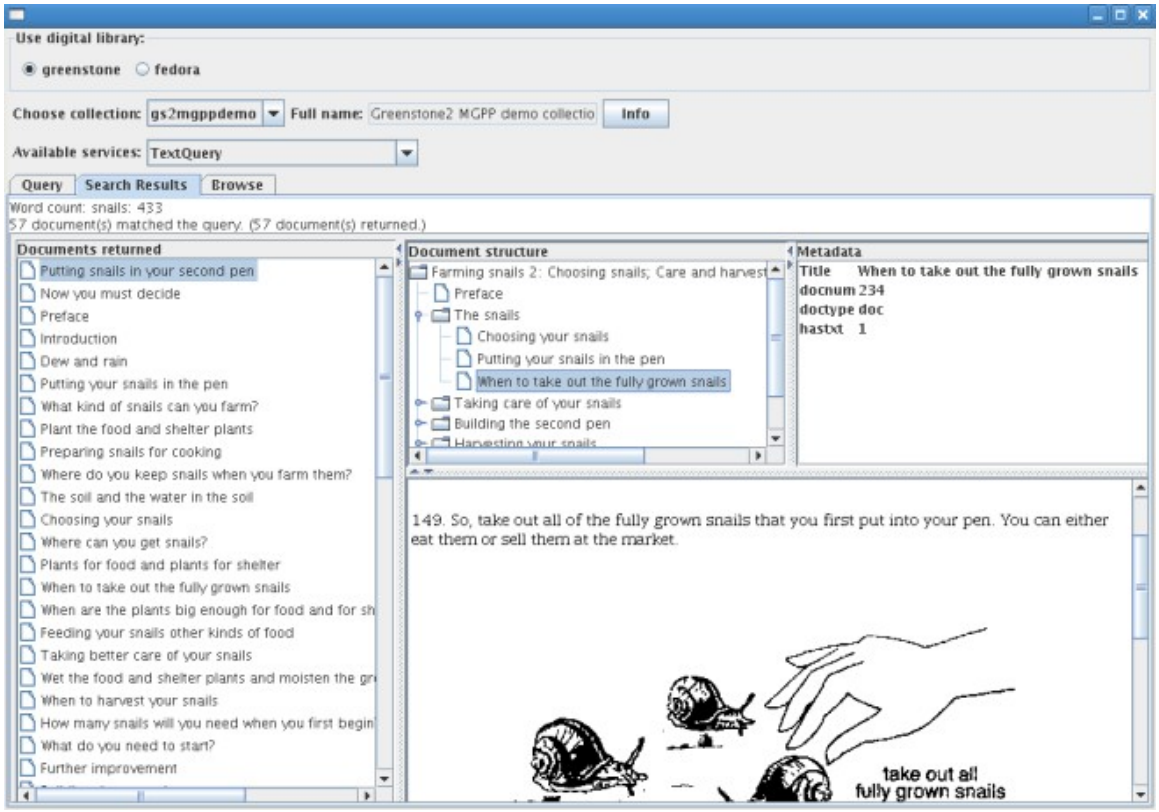


Figure 3.15: Performing a Text Query for the term "snails" on the documents of the GS2MGPPDemo collection

3.2. The Search results pane

When the user executes a search by pressing the long *Search* button at the bottom of the query form (which can be seen in Figure 3.8, for example), the focus changes to the *Search Results tab* and its pane opens to show the documents retrieved as a result of performing the search. The screenshot of Figure 3.15 shows how the *Search results pane* is divided into 4 panels. The left-most panel is a tree of the search results (*Documents returned tree*), which displays the list of top-level documents or document sections found. Clicking on an item in it will show the document's outline in the *Document structure tree* located in the panel next to it. The user can expand nodes in this second tree to view the document's outline and click on any of its sections to view them. For any document or section that is selected either in the *Document structure tree* or in the *Documents returned tree*, its contents are displayed in the main panel. Since documents are plain text or HTML, these are displayed—along with any embedded images they may contain—in the *Contents panel* (the bottom right panel in Figure 3.15). The metadata for any top-level document or any selected section thereof is shown in the *Metadata panel*.

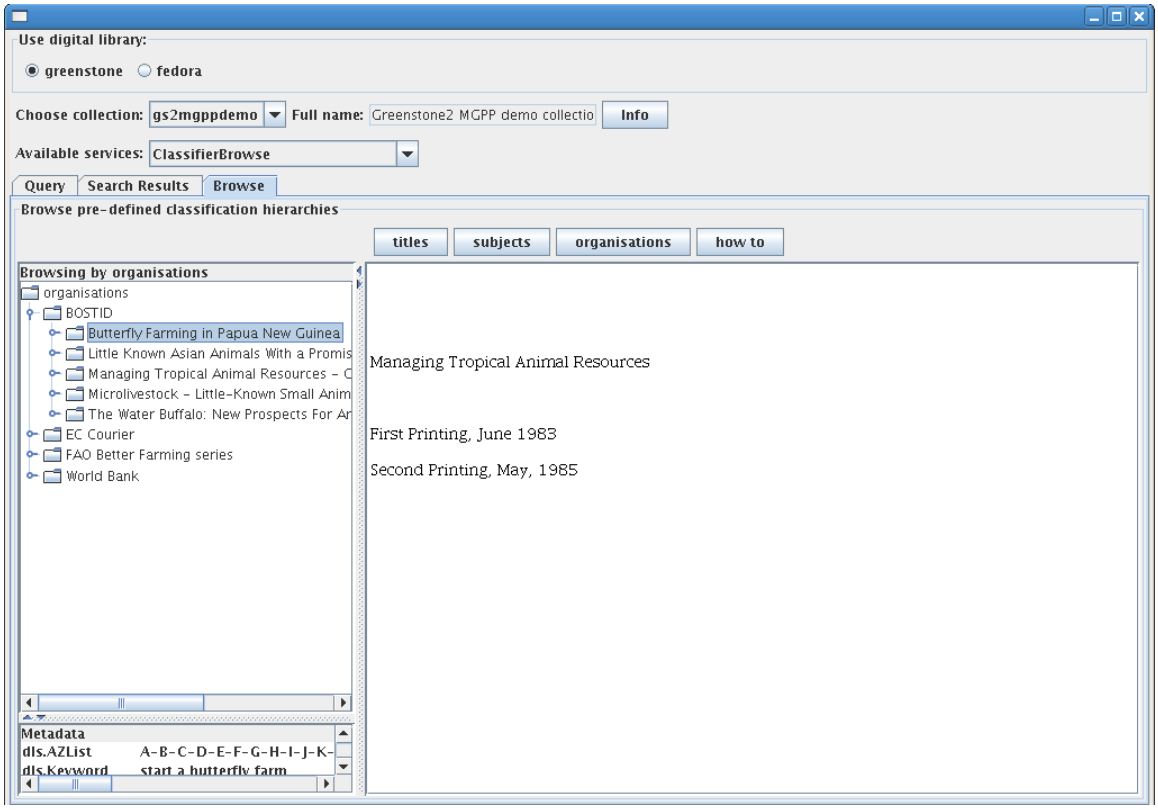


Figure 3.16: Screenshot of browsing a sub-category ("BOSTID") of the "organisations" classifier in Greenstone's GS2MGPPDemo collection.

3.3. The Browse pane

If the Classifier Browse service is selected from the *Services* drop-down box, then the *Browse tab* will become active. The *Browse pane* that opens as a result contains 3 panels and a bar of *Classifier buttons* along the top, as shown in the screen capture of Figure 3.16.

The *Classifier buttons* are for the various browseable categories in this collection (the buttons correspond to the classifiers associated with the currently active collection). Clicking on one of the buttons will make that classifier's browsing structure available for viewing in the left-most panel, the *Browse structure tree*, which displays the selected browsing category's hierarchical organisation.
The *Browse structure tree* uses a folder structure to display the top-level classifier (the browse category) which can be expanded to show any subcategories and then opened up further, all the way down to the documents in the category and their sections. Unlike in the *Search Results Pane*, there is no separate panel for viewing the document structure, because the *Browse structure tree* already shows the entire outline of documents—albeit in the context of the overall category that a document is found in.

Upon clicking a classifier, sub-classifier, document or subsection in the *Browse structure tree*, the metadata associated with that classifier or document section is shown in the *Metadata panel* that is located at the left-bottom of the *Browse pane*. When a top-level

document or any of its sections are clicked, their contents are displayed in the *Contents panel* in the centre of the pane. (Classifiers, being categories, have no content.)

The screenshot of Figure 3.16 shows a part of the browsing structure of the Organisations classifier. This classifier has sub-categories (sub-classifiers), such as the visible BOSTID. However, not all browsing categories contain subcategories and expand to directly reveal top-level documents.

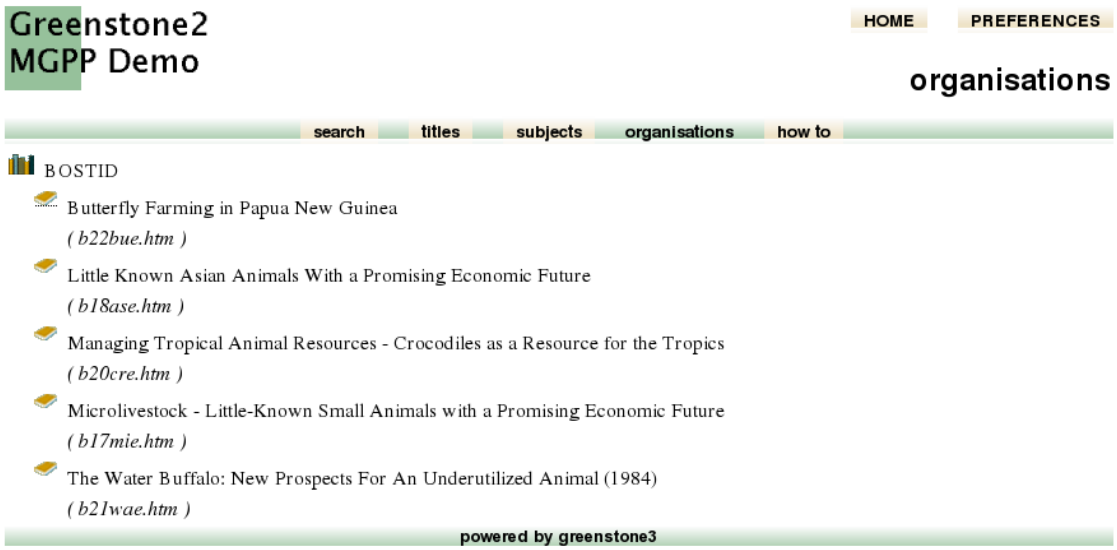


Figure 3.17: Browsing the demo collection GS2MGPP by organisation in Greenstone 3's browser interface. The browse classifier BOSTID has been expanded to show the top-level documents it contains.

Figure 3.17 is a screenshot of browsing the GS2MGPPDemo collection by organisation using Greenstone 3's browser interface, where the selected organisation is the BOSTID subclassifier. This is the way the browser interface presents the same information as the Java-client did in Figure 3.16. Each subclassifier—like BOSTID—is shown on its own web page, whereas the Java-client creates an expandable node for each subclassifier.

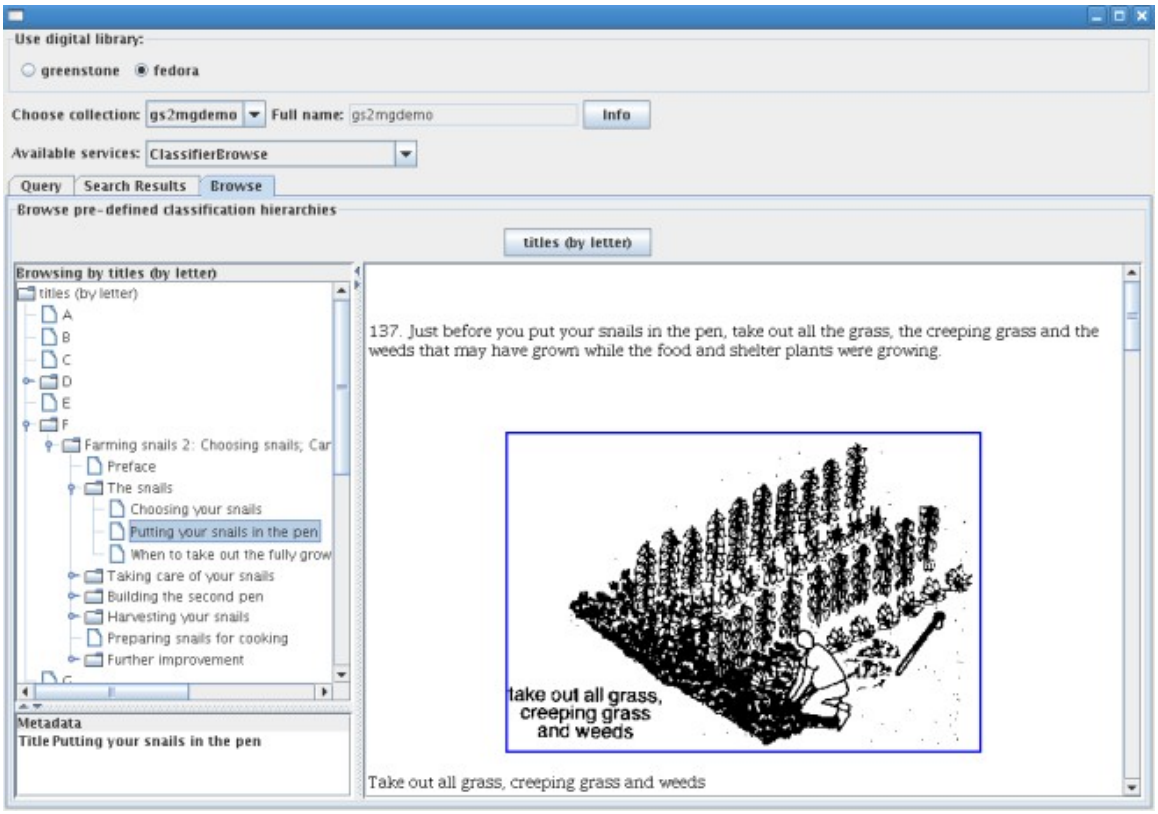


Figure 3.18: Screenshot of browsing through the Browse Titles by Letter classifier offered by the FedoraGS3 component's ClassifierBrowse service

The FedoraGS3 component we have implemented, and which we will be discussing in detail in the next Chapter, mediates between the Fedora repository and Greenstone 3's Java-client. It provides various Greenstone-like services, such as a query service, that the Java-client interface knows to present in a manner similar to Greenstone's services. One of the services FedoraGS3 provides is a simple Classifier Browse service for the Greenstone documents stored in the Fedora repository. It allows users to browse titles according to their first letter, as shown in Figure 3.18.

3.4. Displaying images

Greenstone collections need not consist of just text-based documents such as Word and HTML documents. It allows users to create different kinds of data collections, including image collections where the documents are either images, or are scans of text documents that may or may not have associated OCR text.

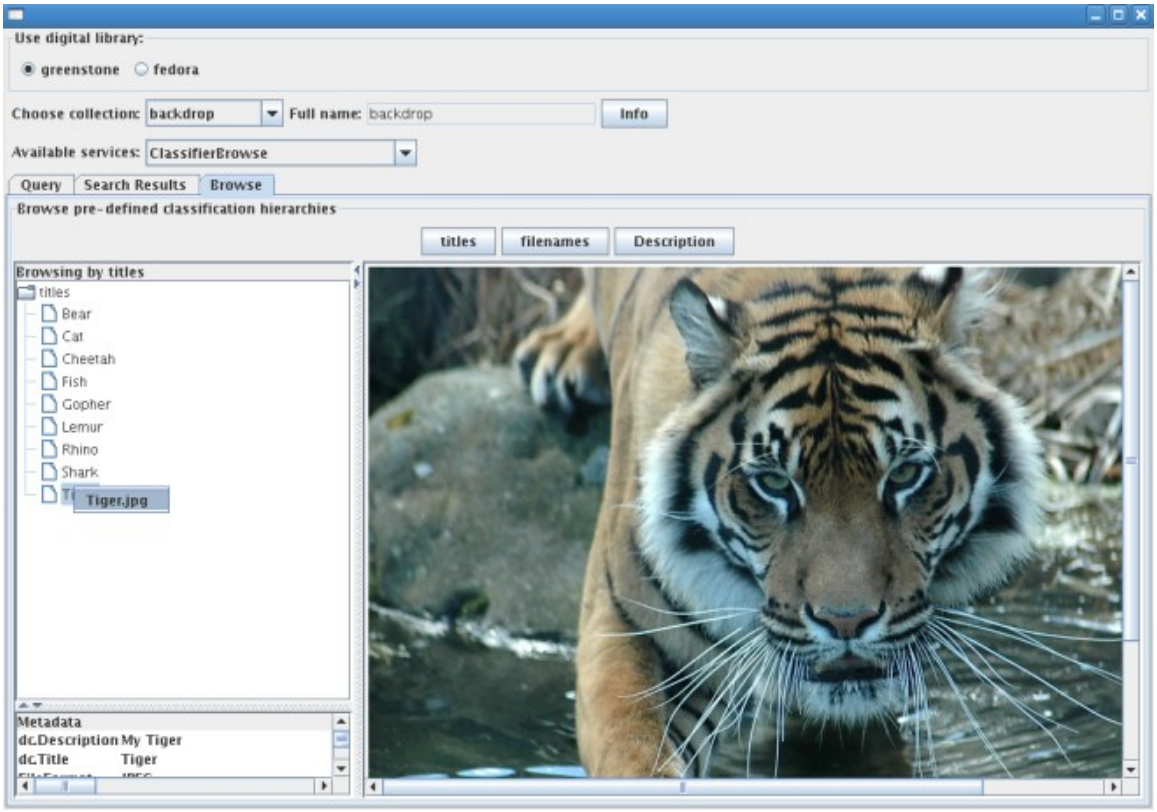


Figure 3.19: Viewing an image from the image collection Backdrop. Right-clicking on a document shows a popup listing the name of the image (clicking it will reload the image in this case).

Figure 3.19 shows an example image document from the Backdrop collection that is solely made up of images.

In the Java-client, support for non-textual content such as the Demo Image collection in Greenstone is accessed through right-clicking on a document in:

- the document section in the *Documents returned tree* of the *Search results pane*,
- the top-level document in the *Document structure tree* of the *Search results pane*, or
- the top-level document in the *Browse structure tree* of the *Browse pane*.

Selecting any image listed in the context menu that subsequently pops up will load the image in the pane's *Contents panel*.

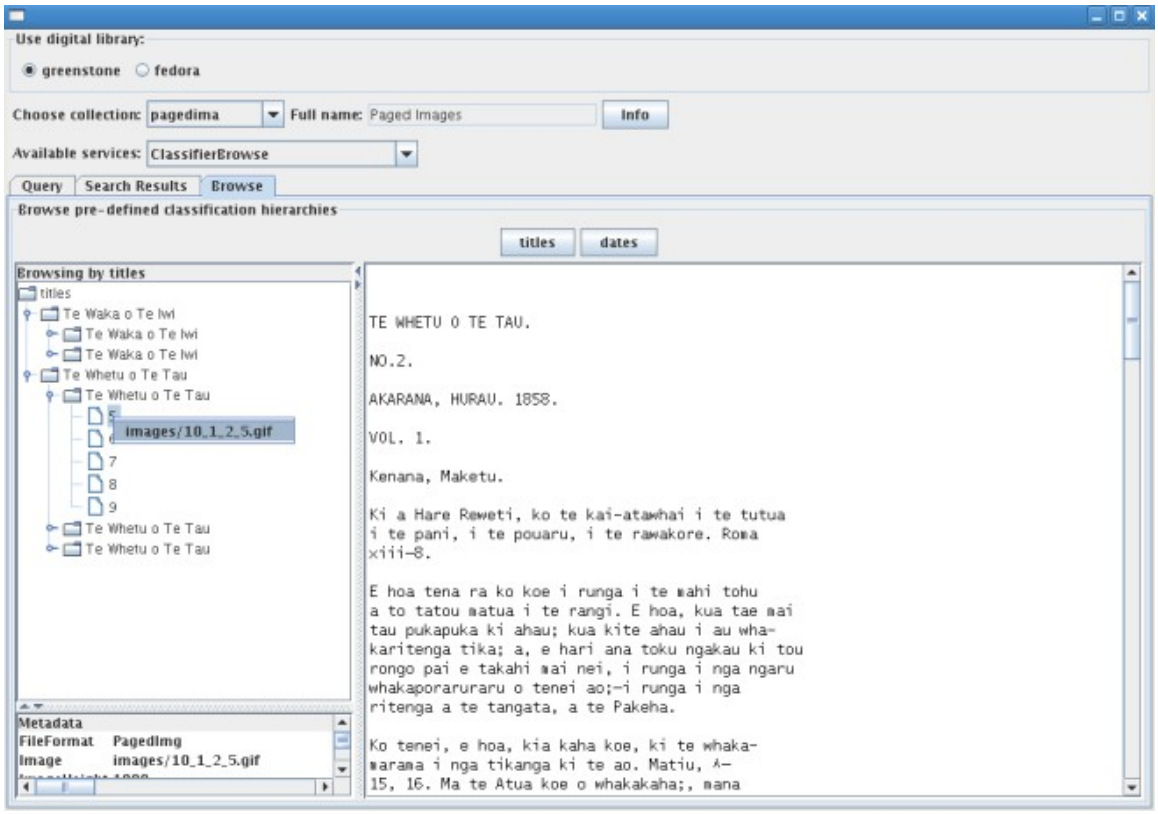


Figure 3.20: For image collections that contain associated text, the text is loaded first.

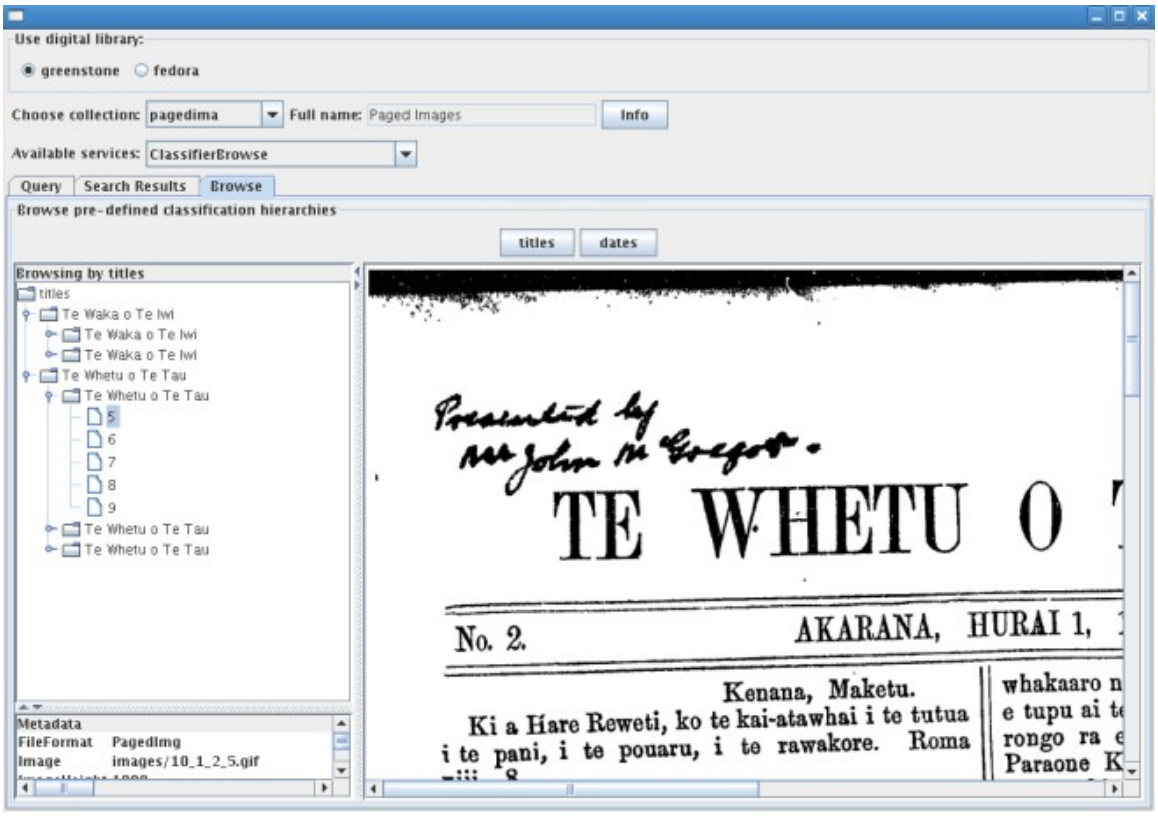


Figure 3.21: The document's popup menu (in the Tree panel) gives access to the image

The demonstration Paged Images collection (a subset of the *Niupepa* collection) is an example of an image collection consisting of newspaper article scans, as well as scanned images with associated OCR text. Where OCR text is included with images, the text of any document selected in the *Browse pane* or *Search results pane* is loaded into the *Contents panel* by default. Right-clicking on a document will list the associated scanned image in the context menu. Clicking on the image file's name in this popup will then load the image. Figures 3.20 and 3.21 show an example interaction with a document from the Paged Images collection, where the document has both a scanned image and OCR text associated with it.

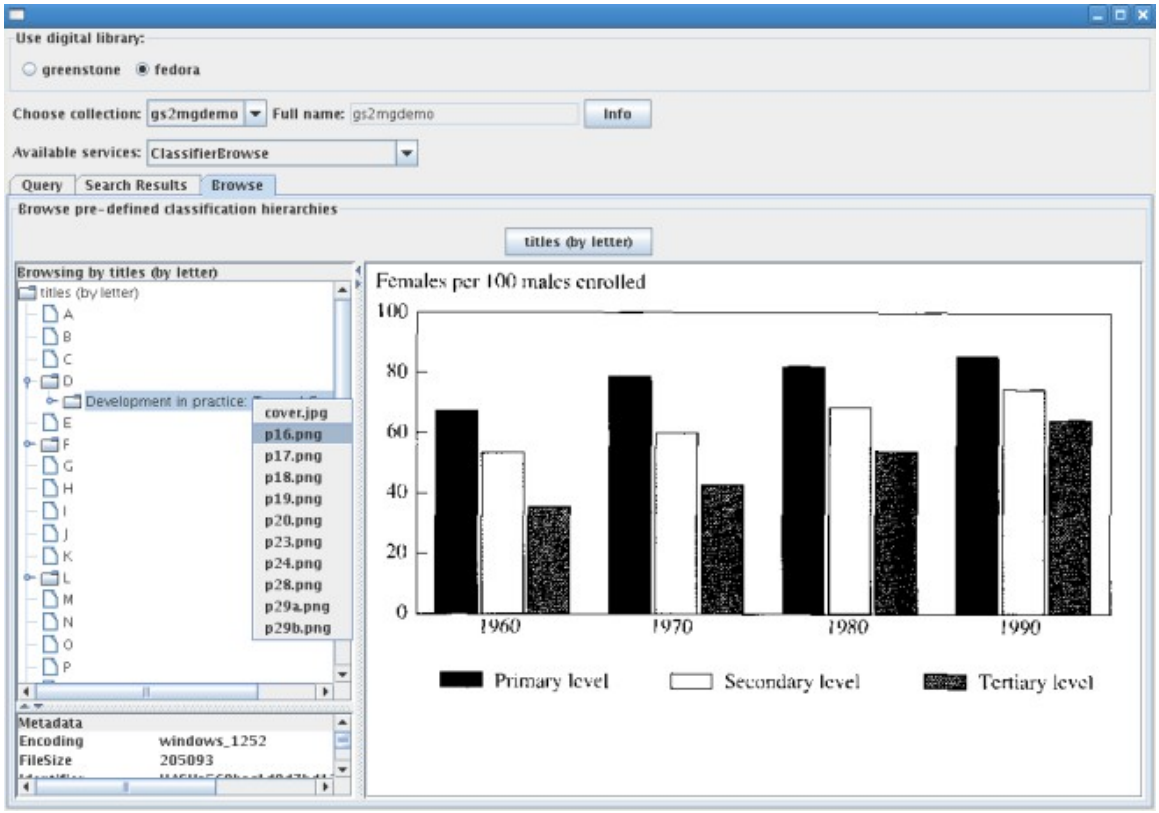


Figure 3.22: Accessing images embedded in the contents of documents stored in Fedora is also accomplished with right-clicking in the Tree panel and selecting the image name

When it comes to documents containing *embedded* images (that is, HTML documents containing references to images in their body), the Java-client is able to display:

- these documents along with their images in its *Contents panel*. This is true for documents from both the Fedora and Greenstone repositories.
- the list of images associated with a document in the context menu that appears when the user right-clicks on the document (in the *Browse pane* and *Search results pane*'s Tree panels that outline structure). If the user clicks on any image name in the popup, the image loads in the *Contents panel*. So far, this works with Greenstone documents retrieved from Fedora, an example of which is shown in Figure 3.22.

However, the latter is not yet possible when accessing documents in Greenstone: where images embedded in text (HTML) documents are concerned, the Java-client is at this point unable to obtain a listing of images associated with such documents. As it is unable

to list them, the context menu is not available upon right-clicking either. This will cease to be a problem when Greenstone is modified such that the Message Router will send the absolute URLs of associated files.

Chapter 4. System design and implementation

This chapter will cover the design and implementation of the three parts of this project:

- web services for Greenstone 3,
- a Java-client application for Greenstone 3 that is to make use of them, and
- a component to act as interface to the Fedora repository of Greenstone documents, which will enable Fedora access to be integrated into the client application.

At various stages of designing the three parts of the project, different design decisions came up and these will be covered here as well.

All parts of the project were implemented in Java 5 on Linux. The project made use of Greenstone 3, Fedora 2.2.1 and Fedora Generic Search[11].

4.1. Web services for Greenstone 3

4.1.1. Considering the functionality to map into web services

All the useful operations which the developers of Greenstone 3 foresaw that clients and end-users would wish to have access to are available through the Message Router class (as well as the Receptionist classes which interface with the Message Router). It was therefore decided to convert those operations that can be performed through the Message Router into web services.

Under consideration, then, are the following Message Router operations (these are explained in detail in the Greenstone 3 Developer's Manual [8], pages 35-52):

- `Describe-type messages`—these can be sent to Greenstone Collections, ServiceRacks and Services as well as the Message Router itself. Describe messages sent to Query services return a response that specifies how the corresponding query form should be presented to the user. The responses to describe messages sent to other services vary.
- `System-type messages`
- `Format-type messages`
- `Status-type messages`—to poll the results of invoking process type services (see below)
- `Process-type messages`, which consist of
  - `Process-type services`—called to carry out some action such as building or importing collections
  - `Query-type services`
  - `Browse-type services`

[11] The Fedora Generic Search installation used was the slightly modified version that was downloaded from the DRAMA project (Digital Repository Authorization Middleware Architecture). Its installation instructions are at http://drama.ramp.org.au/cgi-bin/trac.cgi/wiki/InstallingFedoraGSearch

`Retrieve-type services`—content, structure and metadata retrieval
`Applet-type services`—these process data for an applet
`Enrich-type services`—used to mark up text

- `Page-type requests`—for page generation

It should be noted that web services are not a new *Greenstone* Service, such as querying, browsing and retrieval functionality are. Therefore it is not pertinent to inherit from Greenstone 3's *ServiceRack* class as suggested in the Greenstone 3 Developer's Manual [8] (page 59).

4.1.2. Web service APIs

One way to present the Message Router-accessible Greenstone functionality would be to have a single API of web services for it that would map all the selected Greenstone operations. Another way of organising its web services would be to go by Fedora's example. Fedora has divided its web service operations into two separate APIs: one consisting of those that pertain to managing a repository (the Management API) and one for those concerned with giving access to digital objects already in the repository (the Access API). Greenstone 3 also has operations that fall under the 'access' category. For instance, the query, browse and retrieval services relate to accessing the contents of the Greenstone repository.

It was decided not to split the web service operations for Greenstone into separate API categories after all. Since the Greenstone web service class would instantiate a Message Router object to perform the underlying functions, any clients accessing both APIs would be instantiating *two* separate Message Router instances.

4.1.3. Types of parameters and return values for the web services

Having decided on mapping Greenstone operations that are available through the Message Router into individual web service methods, the next step was to decide on the types of parameters and return values.

General considerations on using data structures as web service parameters or return values

It is possible for web services to return data structures, or for clients to pass data structures as parameters to web services. The data structures that can be passed to and from web services represent only data: objects with multiple fields. They do not have any methods (functionality) associated with them. Functionality is what web service operations themselves ought to provide, while the data is what is passed back and forth with SOAP's encoding format [9]. This must be borne in mind when designing the types of return values and parameters for web service methods.

Custom data structures are considered complex data types in SOAP. Programming languages that have support for web services have the ability to map the single-field types like strings, integers, booleans and arrays into the associated simple data types accepted by SOAP. Custom data structures have multiple fields and require special processing. In Java's case the data structures are allowed to be Javabeans, with an additional restriction on their member fields that requires them to be basic data types or Collection types. If these requirements are satisfied, Apache Axis—which supports web services for Java—can use its default Bean Serializer and Deserializer to pack and unpack Javabean complex types that conform to these specifications. Data types that are more complex than such Javabeans need to have Custom Serializers and Deserializers written for them on the server and client ends. (This is the case regardless of whether such complex data types are used as parameters or as return values.)

Input parameters

There are a number of options for the type of input parameters the web service operations can have. It can consist of:

- a data structure, containing the relevant information
- XML (string) either representing a whole request message or just the portion of a request message that is required by the particular operation being invoked.
- simple types such as strings, integers, arrays (and Apache Axis allows for Java Collection types to be passed with SOAP as well).

While complex data structures can be used as parameters to web services, this need not always be helpful. Parameters to methods can be many in number, while many programming languages restrict the number of return values to only one. Therefore, custom data structures are a necessary consideration when it comes to designing the types of return values, but need not be required as parameters when these can be equally handled by a sequence of simple data types. One disadvantage of using data structures as parameters is that the client would be required to construct a data structure that is recognised and accepted by the web services end. Next to that, if a client were to merely pass simple data types as parameters, instead of first constructing a data structure out of them, it would cut out some extra effort on the client end.

Greenstone 3's original `process` web service took an XML request message as input and returned the same format in response. This was a very practical and elegant choice of format, as XML can encapsulate any kind of well-formed and valid data. As a result, all the kinds of messages (corresponding to Greenstone operations calls) that the Message Router is capable of handling need only be represented by a single input parameter: XML. With XML as input, the contents of messages can evolve in the future to perhaps contain additional pieces of information while the signature of the core Greenstone `process()` method itself need not change (and only the actual processing of the request and response messages might require changes). This means that callers of the `process()` method need not alter the method call itself, even if some of the code constructing requests and parsing response messages may then be modified to deal with

newly-introduced data elements [3]. However, if the new web services were to take XML as input, there is not much advantage gained over the existing `process` web service.

One of the considerations in providing the enhanced web services for Greenstone 3 was to try to make invoking the web service methods somewhat easier and more intuitive. This could be achieved if the client itself need not have to construct the XML request messages for such operations, but could instead delegate that activity to the appropriate web service operation. As such, using simple data types for parameters was thought to be more appropriate. The web service operation would use these parameters to construct the XML request message, which could then be passed on to the Message Router's `process()` method. Each parameter would represent an important piece of the information required by the particular Greenstone operation being invoked through the Message Router.

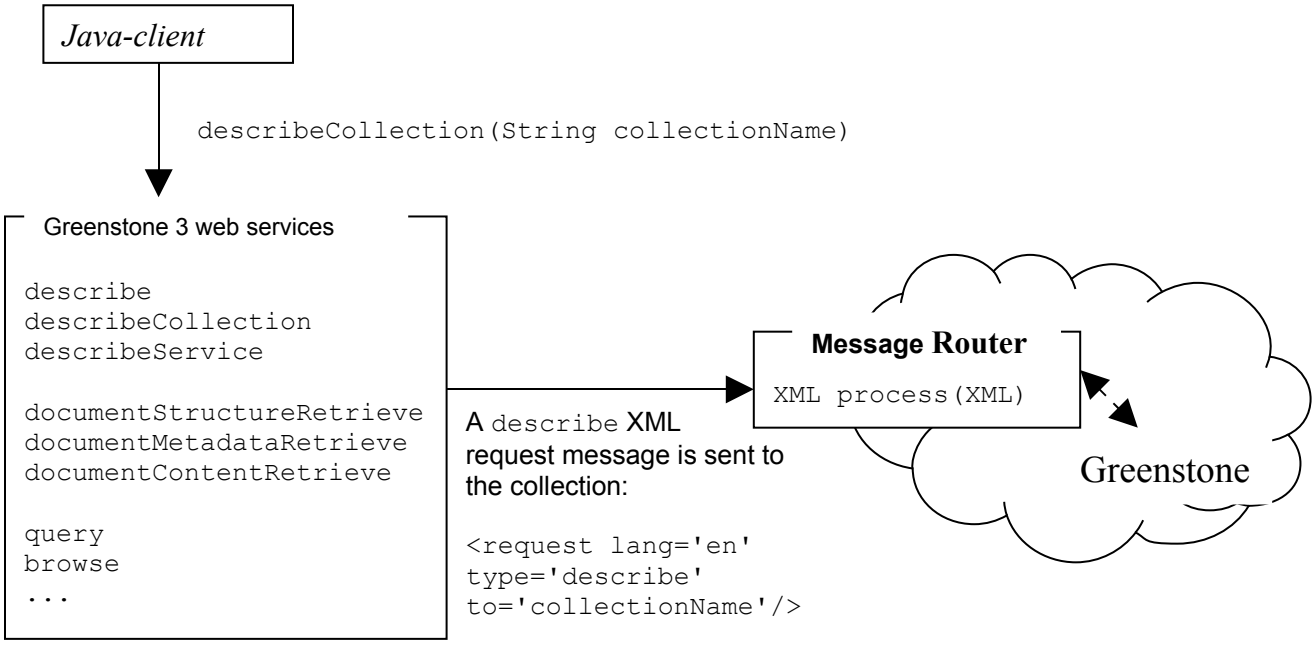


Figure 4.1: The web service operations will construct XML request messages from the basic data types of the input parameters and pass this on to the Message Router's `process` method

For example, when calling Greenstone's `describe` operation on a collection, the corresponding web service could be invoked with a string representing the collection's name. The `describe` web service would then construct the XML `describe request message` and call the Message Router's `process()` method, passing the XML as an argument. The Message Router would thereafter pass on the XML message to the collection that has been asked to describe itself. See Figure 4.1. This last returns an XML response message, which the Message Router would return to the `describe` web service operation, which would in turn pass it back to the client.

Return values

The possibilities for the return type of the web service operations include:

- returning the XML response message as is, or
- returning a custom data structure/object whose fields would contain the same information as held in the response XML.

There are advantages and disadvantages to returning custom data types. The major advantage to returning a custom data structure is that a client need not parse any XML data in order to retrieve the pieces of information it wants. Instead, it need but access the data structure's member fields. This means that clients need not know the format of a response XML emanating from Greenstone. Any Java developer who has perused the Javadocs for the various data structures that could be returned from the web services may know how to start using them. On the other hand, if plain XML response messages were returned, clients would need to be familiar with the XML response message format laid out in the Greenstone 3 Developer's Manual, in order to handle all the data elements a response message may possibly contain.

Since the indexing and searching of full-text in Fedora is provided by the separate Fedora Generic Search (*FedoraGSearch*) installation, the full-text searching capabilities that the latter exposes as web services are not part of the Fedora Access API. While Fedora Generic Search's web services return XML response strings, the web service operations of Fedora's APIs return non-XML data, including convenient-to-use custom objects. They return complex data types where necessary and simple data types otherwise. From the perspective of someone developing an application that makes use of Fedora's web services, the custom Fedora types are very easy to work with since they require no processing.

In Greenstone's case, returning data structures would be somewhat more involved. Some operations can return XML representing different data depending on which Greenstone component the request message was sent to. For example, a `describe` request message sent to a Query service returns XML representing the display data for a query form, whereas a `describe` request sent to a Browse service returns XML with a different internal structure and represents display data meant for top-level browse classifiers. In spite of the operation being the same (a `describe` request sent to a Greenstone Service), the services are different and their XML response messages are consequently not the same: they don't contain all the same elements. Therefore, their response XMLs can not be represented by the same data structures. That is, it is not possible to represent all `describe` responses with a single data structure.

There is a further, if minor, disadvantage to returning custom data types. Using such complex data types for return values would involve parsing the response XML into individual components on the web services side. This would preclude any custom parsing that the client may wish to perform. For instance, a client application may not choose to parse all of the XML and may instead only want to retrieve a single sub-element from a response message. In that case, the web service's parsing to create the custom data

structure from the response XML and passing this back to the client would be an unnecessary overhead.

After much deliberating, we opted for returning the XML response message as a String (which is a simple data type). Although this meant that clients would have to parse the response themselves, this choice was particularly useful when it came to integrating Fedora access into Greenstone 3's Java-client, as will be discussed in Chapter 4, System Design and Implementation.

Even so, a small allowance has been made for Java developers who may want to use Greenstone 3's web services but who do not look forward to parsing the response XML themselves. The Greenstone 3 client application that has been built for this project contains some classes that first parse then encapsulate the information returned from many of the 'Access' operations of Greenstone (like browse, query and retrieve). Others may choose to make use of these classes instead of doing the parsing themselves.

In summary: While Greenstone's existing process web service took an XML request message and returned an XML response message, the new web services will take simple data types as parameters and return the XML response message received from the Message Router back to the client as-is.

4.1.4. The design stages

The web services for Greenstone 3 were designed and implemented a first time (as discussed in the subsection *Design Stage 1* below), after which they were evaluated as covered in Chapter 5, Evaluation. Taking the recommendations from the evaluation into account, the set of web services have been adjusted. The final web services as implemented are presented in subsection *Design Stage 2*.

Design Stage 1

Although all Greenstone 3 operations that can be accomplished through the Message Router were already available through the existing *process* web service, it was thought useful to provide individual web service methods for at least the most frequently used Greenstone operations.

Since creating web services for Greenstone was only part of the project and another major part was implementing a Java-client application to use the same, the main focus during the implementation stage was to first create those web services that the client required. The client application was to provide some of the same functionality that is already available through Greenstone 3's browser interface. The operations under consideration, therefore, were of the "Access" category, such as Greenstone's query, browse and retrieve-type services for document content, metadata and structure. Therefore, providing methods to ease invocation of Greenstone's Access operations were the minimum that the web services needed to offer. Whether it is an acceptable minimum —and whether the remaining (non-Access related) Greenstone operations can be left to

the more general `process` web service operation—is a question considered in Chapter 5, Evaluation.

While mapping the Access operations of Greenstone into web services, it was thought useful to overload web service methods for several Greenstone operations, such that some variants would accept XML parameters after all. (These methods would be available in addition to the default method for each operation that was to take simple data type parameters.) The XML parameter would represent those portions of request messages that were unique to the operation, rather than an entire XML request message. It was thought that methods with such XML parameters could prove useful for any clients that might already have prepared the necessary segment of an XML request message.

Further (different) instances of method overloading were used for some Greenstone operations as well, in the expectation that clients may find invoking one of the variants easier in certain circumstances. Most of these variants were designed to take optional parameters into account.

Table A.1 of the Appendix lists the web services implemented in Stage 1. They correspond to the Greenstone 3 Access operations that are available through the Message Router.

Table 4.1: Listing of the Greenstone 3 web service method definitions. See also the Greenstone 3 Developer's Manual [8], pp.35-52

Methods for sending describe-type requests to the Greenstone's Messagerouter, Collections, ServiceRacks and Services	
String describe(String lang, String subsetOption); String describeCollectionOrServiceRack(String toSC, String lang, String subsetOption); String describeCollService(String collection, String service, String lang, String subsetOption); String describeService(String service, String lang, String subsetOption);	
Query requests for executing queries	
String queryProcess(String collection, String service, String lang, String[] names, String[] values);	The Hashmap parameter requires mappings from query fieldnames to field values. The fieldnames and fixed field values must be those recognised by Greenstone 3.
String queryProcess(String collection, String service, String lang, HashMap nameToValsMap);	
String simplerFieldNameQueryProcess(String collection, String service, String lang, HashMap nameValParamsMap);	This method's Hashmap parameter of query fieldnames and values will accept the user-friendly *fieldnames* returned by the getFieldNameMappings() method as well as the abbreviations used by Greenstone 3. This method also allows the user to provide *field values* like "all fields", "text", "titles", "subjects", "organisations" instead of Greenstone's accepted ZZ, TX, DL, DS, DO respectively. The user can provide "on" or "off" for such fields as casefolding and stemming, instead of the expected "1" or "0".
static HashMap getFieldNameMappings();	Hashmap of mappings from user-intelligible query fieldnames to the argument abbreviations Greenstone 3 uses for them. Users who wish to use the two regular queryProcess methods that take HashMap parameters, may call this method to find out what the query fieldnames are that Greenstone 3 accepts in order to pass the correct fieldname abbreviation.
Retrieve request messages for Content, Structure and Metadata retrieval for Documents and Classifiers	
String retrieveDocContent(String collection, String lang, String[] docNodeIDs);	DocumentContentRetrieve request
String retrieveEntireDocStructure(String collection, String lang, String[] docNodeIDs);	DocumentStructureRetrieve requests
String retrieveDocStructure(String collection, String lang, String[] docNodeIDs, String[] structure, String[] info);	

String retrieveAllDocMetadata(String collection, String lang, String[] docNodeIDs); String retrieveDocMetadata(String collection, String lang, String[] docNodeIDs, String[] metaNames);	DocumentMetadataRetrieve requests
public String browseDescendentsOf(String collection, String browseService, String lang, String classifierNodeID) String browse(String collection, String browseService, String lang, String[] classifierNodeIDs, String[] structureParams)	ClassifierBrowse – Structure retrieve for browsing classification hierarchies
String browseMetadataRetrieveAll(String collection, String categoryName, String lang, String[] nodeIDs); String browseMetadataRetrieve(String collection, String categoryName, String lang, String[] nodeIDs, String[] metaNames);	ClassifierBrowseMetadataRetrieve – Metadataretrieve for browsing classification hierarchies
Operations not related to Accessing the repository	
String status(String to, String language, String pid)	
String reconfigure(String subset) String reconfigureCollectionOrCluster(String name, String subset)	
String activate(String systemModuleName, String SystemModuleType) String deactivate(String systemModuleName, String SystemModuleType)	
String format(String service)	
String processTypeService(String to Hashmap params)	For example, the parameter `to` can be “build/NewCollection”, “build/ImportCollection”. See Manual, Section 3.8.4, p. 49
String appletTypeService(String collection, String service, String requestType, Hashmap params)	
String enrich(String service, Hashmap params, String[] nodeIDs, String[] nodeContents)	
String pageTypeRequest(String action, String subaction, String language, String outputType, HashMap params)	
The core process method (which can process all valid Greenstone 3 messages)	
String process(String requestMessageXML);	A direct mapping of the process method of Greenstone 3's MessageRouter.

Design Stage 2

On evaluating Stage 1 of the Design, certain improvements were suggested as described in Chapter 5, Evaluation. They include:

- expanding the set of web services to cover practically all the major Message Router operations that are possible (not just the Access-related ones), and
- less frequent use of method overloading to reduce the number of similar methods. This was achieved by allowing clients to pass default values for optional parameters, and by removing methods that took partial XML requests which were found to be unnecessary additions.

These changes have been incorporated and the final list of web service operations is presented in Table 4.1.

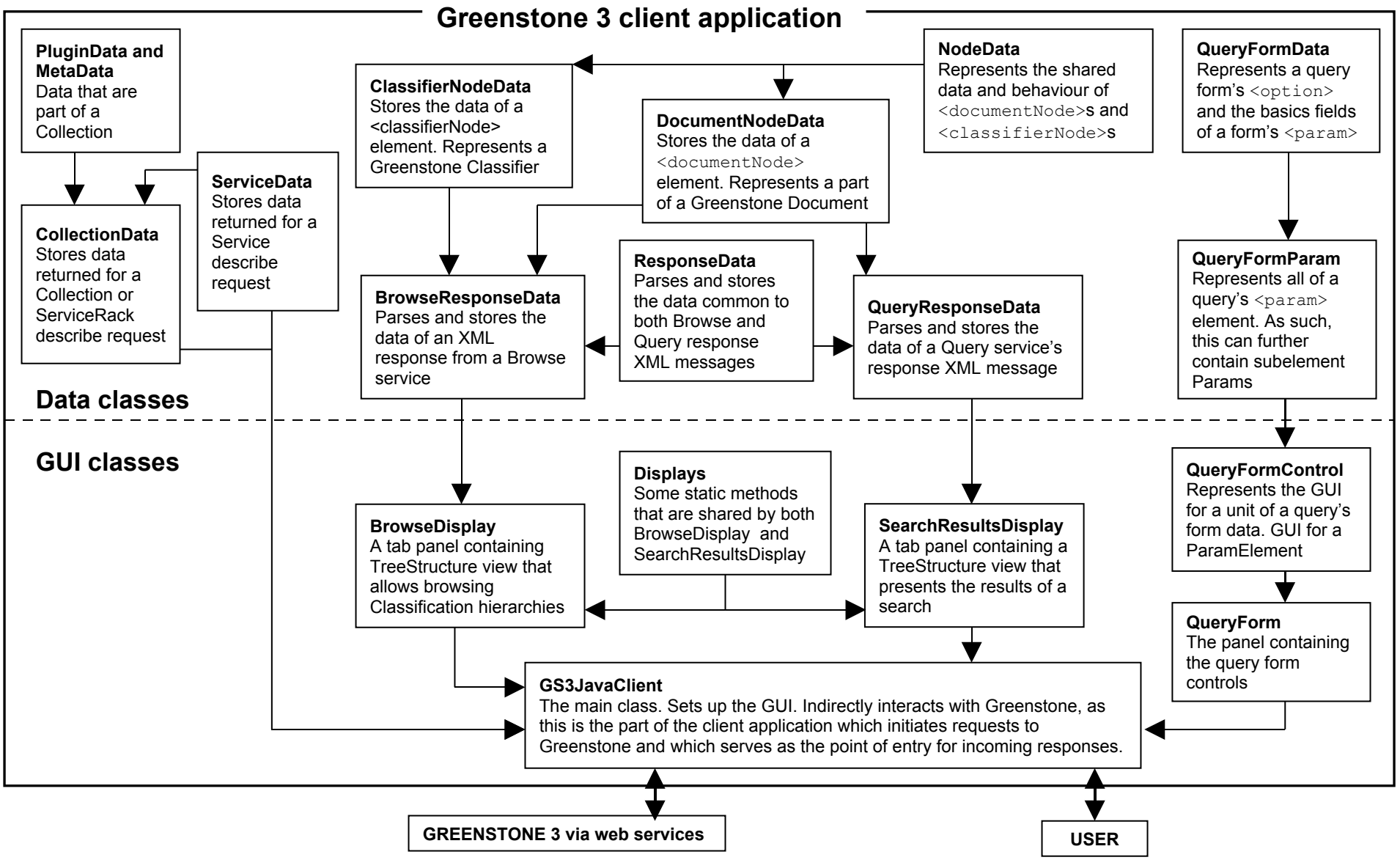


Figure 4.2: The Data and GUI classes of the Greenstone's client application

4.2. The Java-client for Greenstone 3

The browser interface for Greenstone allows end-users to access a repository of collections and documents by providing querying, browsing, and of course document retrieval functionality, among others.

The Greenstone 3 Developer's Manual [8] discusses several alternative methods to build GUI applications for Greenstone 3 that will provide a user-interface equivalent to what is offered by a Greenstone-powered browser (page 60 of the manual, Section 4.3 "New Interfaces"). One of these methods is to have the client application communicate with Greenstone 3's Message Router component in order to access Greenstone's functionality. One of our intentions with this project was to have the client application access Greenstone 3 via the new set of web services, rather than directly communicating with Greenstone's core classes. This would allow the client to simultaneously serve as a means for testing proper implementation of those of the web services that would be invoked (in that it would be an actual scenario of use for those services), and for evaluating the breadth of operations provided. It could also help to identify any missing web services during development so that omissions could then be rectified.

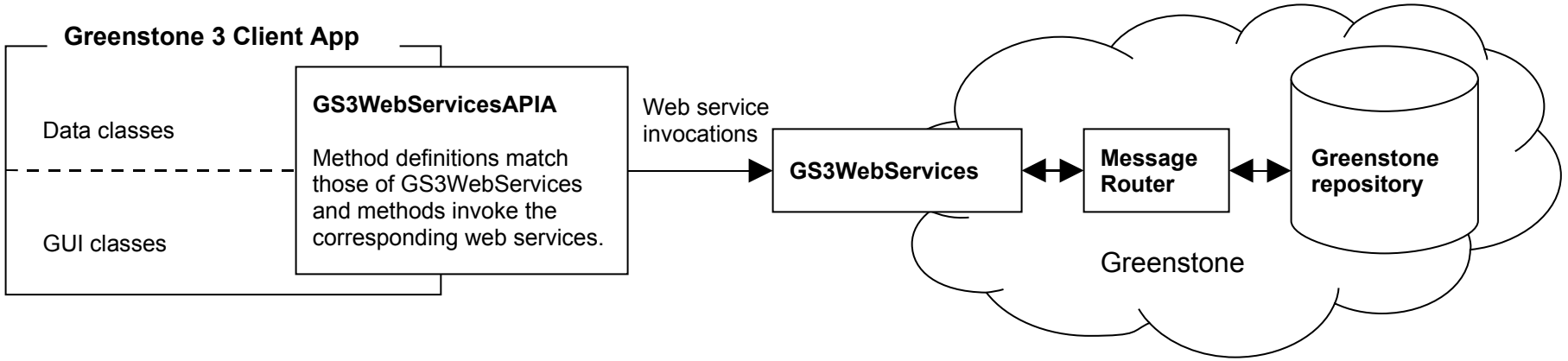
4.2.1. Design and Implementation

Since the web services were going to return the XML response emanating from Greenstone, rather than parsing their elements into data structures and returning these, the parsing and storing of relevant data became part of the implementation of the client application. The client's functionality was first separated into two packages. The *Data classes* were organised according to the XML response data they would parse and store. The *GUI classes* were for visually presenting the various data (such as a listing of the available collections and services, and the display of query forms, browse and search results).

Among the data structures that would deal with processing the XML, the most important were:

- The *QueryResponseData* and *BrowseResponseData* classes that would parse and store the XML response messages sent in response to query and browse requests. These two classes would maintain a list of all the DocumentNodes and/or ClassifierNodes returned in the search results or when browsing.
- *DocumentNodeData* and *ClassifierNodeData*—the classes that would represent a single DocumentNode and ClassifierNode. They would store the data they obtained from parsing the `<documentNode>` and `<classifierNode>` elements respectively.
- The remaining data classes were simpler and represented `<collection>` and `<service>` elements or represented Greenstone-determined form controls for queries.

Figure 4.2 shows how the Data and GUI functionality of the Java-client is organised into Data classes (which parse and store the data in the XML response messages) and the GUI classes that visually present this data. The diagram also shows the main connections between the two.



GS3WebServicesAPIA is the client's interface to Greenstone operations: an intermediate class between the client and Greenstone 3' web services (*GS3WebServices*). It deals with invoking the Greenstone 3 web services.

The cloud indicates that, aside from GS3WebServicesAPIA, the rest of the client application does not know how Greenstone 3 is accessed or how its operations are executed. From the client's point of view, the class GS3WebServicesAPIA is the point of interaction with Greenstone 3. (One could replace the GS3WebServices component and instead make direct calls to the Message Router from within GS3WebServicesAPIA and the client would require no changes.)

Figure 4.3: The GS3WebServicesAPIA part of the Greenstone3 Client application handles the web service invocations

Invoking web services requires some extra overhead on top of what is needed for regular method calls. Some simple optimising techniques were used to keep down the number of web service invocations, the amount of parsing work and, consequently, the amount of data stored in the program's memory. In general, this amounted to no more than delaying the parsing work until required. The decision to process only when necessary was made on the basis that users might not always choose to view every single result returned for a search or every single item available for browsing. For instance, the Browsing pane contains a tree showing the structure of the browsing categories and their subcategories (see Figure 3.16). In their turn, these subcategories contain documents and document sections. The client application will only retrieve the necessary information when a user clicks on a node in the tree. If a user clicks on a category, the tree-view will expand it. It is then that the titles of its subcategories are retrieved by invoking the appropriate web service operation (which itself calls the underlying Greenstone retrieval mechanism). Likewise, it is only when users click on a document that its subsections are retrieved. And only when a section is clicked are its contents retrieved. If a node is never clicked, further data need not be retrieved nor stored. This way, the data structures representing Greenstone's DocumentNodes and ClassifierNodes do not need to parse and store the content, structure or metadata in their member fields until actually required. Likewise, web service calls to perform the retrieval of requested data are issued only when needed.

To make the Java-client work with the web services without knowing that it is specifically dealing with web services, an intermediate class, *GS3WebServicesAPIA*, was created whose operations would handle the details of the web service invocations. Thus, from the point of view of the Java-client's GUI and Data classes, it makes no difference whether the intermediate class' implementation was directly calling Greenstone functionality or whether it was going through web services in order to do so. See Figure 4.3. We revisit this class later, when we incorporate access to Fedora into our client (Section 4.3.3).

4.3. Working with the Fedora repository

4.3.1. Storing Greenstone documents in the Fedora repository

The third aim of this project was to show how it is possible to support access to repositories backed by different digital library softwares (Greenstone and Fedora in our case), through the same client interface. One way in which this can be achieved is for some of the *same* content—collections and their documents already present in Greenstone—to also be stored in Fedora. Fedora provides little in the way of support for ingesting documents in Word, PDF, etc. directly. It only accepts FOXML (Fedora Object XML) and FedoraMETS (a custom METS format). Fortunately, Greenstone has the ability to export its contents into a variety of digital library formats [4], and is able to export to FedoraMETS for ingestion into Fedora. The decision was made to use Greenstone's ability to convert its collections to FedoraMETS to populate the Fedora repository we would be working with. By doing so, the contents of the Fedora repository would contain Greenstone collections, which makes it easier to demonstrate that comparable actions (such as browsing and retrieving a document's structure) can be performed on the digital objects stored in the Fedora-backed repository as can be carried out on Greenstone-backed documents.

Since we were duplicating some of Greenstone's contents as digital objects in Fedora, it was necessary to also have a way to:

- represent the concept of Greenstone *collections* in Fedora;
- store the data of a Greenstone *document*. These include any additional metadata sets associated with a document beyond the default DC (Dublin Core) used by Fedora, the document's hierarchical structure, individual section information including metadata associated with a section and section content, as well as a document's associated files such as images.

Fedora allows users to arrange data that is internal to a digital object in any way they choose. It merely depends on how they design the internal structure (the choice of *datastreams*) of their digital objects.

The construct of a Greenstone collection was simulated in Fedora by making use of the ability to assign prefixes to digital object identifiers (PIDs). Examples of PID prefixes include Fedora's default *demo:* and *test:* (or the *changeme:* prefix for objects that don't have a registered prefix). To make Fedora recognise a custom prefix merely requires it to be added to the list of accepted prefixes in the repository's configuration file.

Tables 4.2 and 4.3 The datastreams associated with Greenstone collection and document digital objects stored in Fedora

Greenstone can export internally stored documents and collections into FedoraMETS format for ingestion into Fedora. The documents are exported with a custom structure, such that a specific set of datastreams become associated with the document when ingested into Fedora. Together, these Greenstone-specific datastreams in Fedora provide the same structure, metadata and content information for the document as it had in the Greenstone repository.
Fedora identifies each datastream by its *itemID*.

***Collection*, PID = greenstone:<collectionName>**	
itemID (datastreamID)	***Notable information stored in this datastream***
DC	Dublin Core metadata (XML). Every digital object in Fedora has a DC datastream. In a Greenstone collection digital object, the DC datastream identifies this digital object as a *collection* and may contain its display title (possibly in several languages).
TOC	Table of contents. Collections don't really have anything useful stored in here. The TOC datastream is only useful for digital *document* objects.
EX	XML file containing the Greenstone-extracted metadata associated with this collection.
Section1	Section content. There is nothing stored in here for collections; this datastream is only of use for documents.

***Document*, PID = greenstone:<collectionName>-<documentID>**	
itemID (datastreamID)	***Notable information stored in this datastream***
DC	Dublin Core metadata that contains the (top-level) document's title.
TOC	Table of contents. XML file that outlines the document's structure and shows how the document is divided into subsections.
EX	Greenstone-extracted metadata for the top-level document. It lists all the names of files associated with the document (which are stored in `<gsdlassocfile>` tags).
DLS	DLS metadata for the top-level/entire document.
Section1	Contains the contents of the top-level document. Note the "1" at the end. Content for all sections and subsections of a document is prefixed with "Section1". (This datastream is actually an XML file that stores html content as regular text—by escaping it—within a `<section>` tag.)
Section1.*n*	Contains the content of subsection n.
DC.*n*	DC metadata for subsection *n*, contains the title of subsection *n*. For instance, DC.5.3 is the DC metadata associated with Section1.5.3.

EX.*n*	Greenstone-extracted metadata for subsection *n*. For example, EX.5 contains the EX metadata for Section1.5.

In this project,

- The prefix for all Greenstone objects ingested into Fedora became *greenstone:*. Since Fedora allows searching on PIDs using wildcards, the use of a custom PID prefix to designate Greenstone digital objects makes it possible to identify and retrieve just the Greenstone digital objects.
- A Greenstone collection would itself be represented as a *digital object* in Fedora. And like all digital objects, it could have associated metadata contained in its datastreams —just as Greenstone's own collections have collection-level metadata.
- All Greenstone collections would be assigned PIDs of the form `greenstone:<collectionName>`.
- Documents within a collection would be ingested with PIDs that included the document's identifier as well: `greenstone:<collectionName>-<documentID>`.

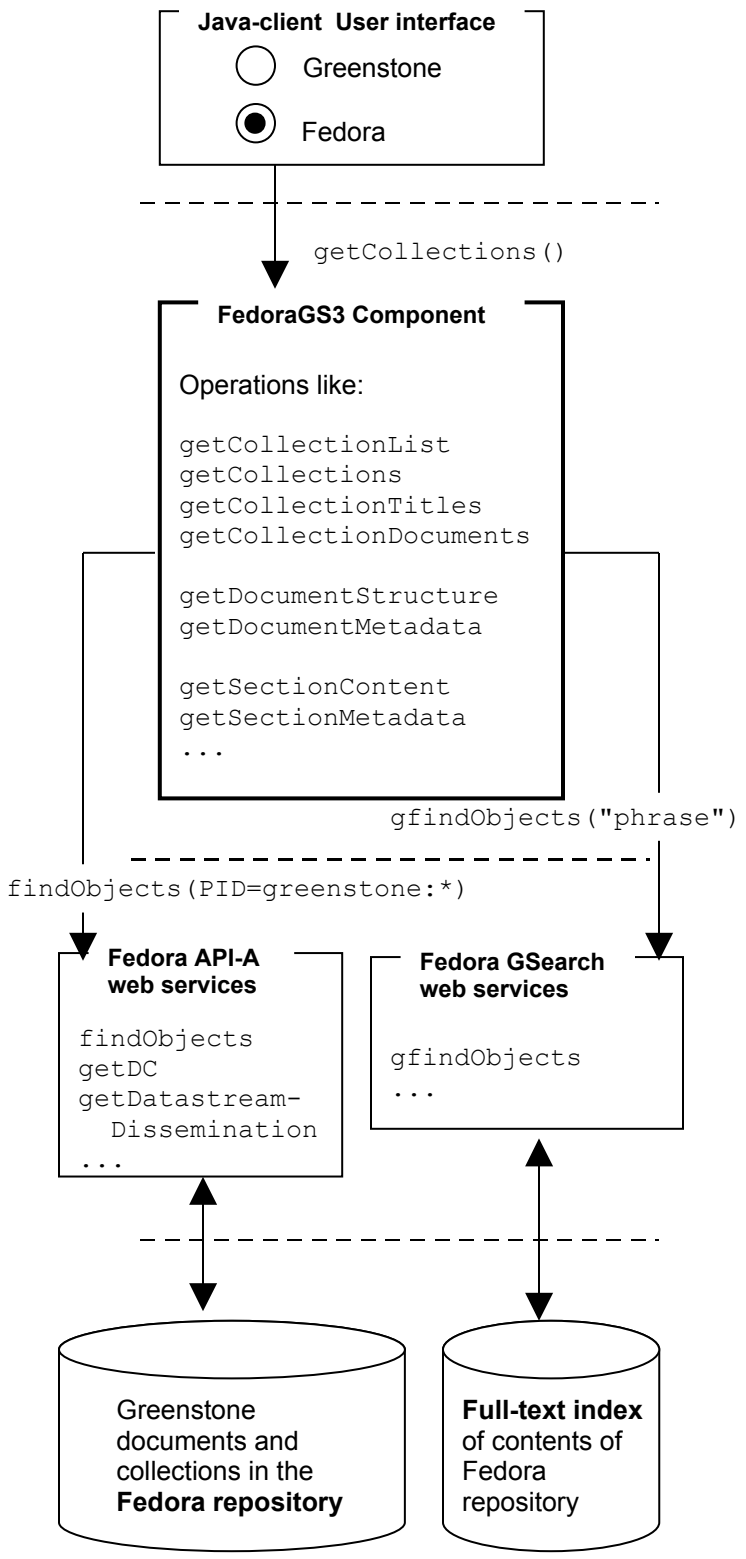
Each Greenstone *document* was also represented as a separate digital object in Fedora and contained datastreams for associated files, document structure, content of sections, section-level metadata, and document-level metadata including the top-level DC datastream that is available in Fedora. The organisation of a Greenstone collection and document in Fedora as a set of datastreams is described in Tables 4.2 and 4.3.

Documents from a sample Greenstone collection were ingested into the Fedora repository in the format described above, and with it, a digital object representing a Greenstone collection. As a result, the local Fedora repository we worked with contained a proper subset of the contents already in the Greenstone repository (in addition to the default *demo:* objects ingested into Fedora after installation). Not all Fedora digital objects can be accessed by our Greenstone client application, but only those that conform to the Greenstone-specific format just discussed. However, using this custom format to store Greenstone collections and documents in Fedora facilitated access to those Greenstone digital objects in a manner similar to how document and collection data are obtained from a Greenstone repository. This meant that the client application that was to connect to the two repositories could be made to access both libraries in the same way. (That is, without having to deal with the specific details of the underlying digital library software, as this work is taken care of by a separate intermediate component nestled between Fedora and the client application which will be discussed in the following section, Section 4.3.2).

4.3.2. Connecting to the local Fedora repository of Greenstone objects

To connect to the local Fedora repository and access digital objects stored therein, clients need to go through the web services of Fedora's Access API (API-A). As was just described in Section 4.3.1, the digital objects we are working with are Greenstone *documents* and *collections* converted into a format suitable for accessing and processing when stored in the Fedora repository. Tables 4.2 and 4.3 contain an overview of the custom formats in which a Greenstone document and collection are stored in Fedora.

Full-text indexing and searching in Fedora is available upon installing the separate Fedora Generic Search facility. Once it has been made to index the Greenstone documents in the Fedora repository, its web services can be used to perform searches on these documents' textual contents (as long as these are stored as plain text, HTML or PDF).



The Greenstone 3 Client application will allow the user to connect to the Fedora repository and access documents stored in there.

The FedoraGS3 component will need to provide operations similar to what is possible with Greenstone.

The component's operations should be tailored to work with the Greenstone documents and collections stored in Fedora's repository (which are items exported from Greenstone into Fedora's internal format). The Greenstone documents in Fedora have the same structural information as they have in Greenstone's own repository. This can be reflected in providing similar operations.

The component's operations will be implemented by invoking:
Fedora's Access API web services to get at the contents of Fedora's repository, and
Fedora Generic Search's web services to search the full-text index of the Greenstone documents.

The Fedora repository's digital objects that the FedoraGS3 component will work with are Greenstone-exported data objects. These have a particular internal structure, which is reflected in the sorts of datastreams available for each Greenstone digital object (TOC, EX, Section, associated files, and EX and DC metadata for each section).

The Lucene index stores the full-text index of documents in the repository.

Figure 4.4: The context of the component that is to provide access to the local Fedora repository of Greenstone documents

In order to access and process the particular kinds of data associated with a Greenstone digital object, we needed to build a component that was tailored to work with the specific structure in which Greenstone documents were stored in Fedora. Figure 4.4 illustrates the context of this component in terms of what it needed to do and how it should fit in the overall processing flow.

The values returned by each of Fedora's web service operations are custom data structures—*Fedora types*. Fedora defines how its complex data types break down into simple types in an XML schema. When Fedora is installed, its web service descriptor file (WSDD) contains a type-mapping section specifying the Javabean serializers and deserializers for the different custom Fedora types (how the various web service return types map to the Java classes that will deal with packing and unpacking them for transport over SOAP).

Since Java was the programming language being used, several choices were available as to how to invoke Fedora's web services. Clients of Fedora can choose to invoke its web services in either of the following ways:

a. by including the stub classes generated by Axis and included in Fedora's installation (though slightly modified by Fedora developers to provide a parameterised constructor). These stub classes act as the GS3WebServicesAPIA of Figure 4.2: they define methods corresponding to each Fedora web service operation, where each method carries out the task of calling the associated web service operation. To clients using the stub classes, it will appear as if they are dealing with regular classes and that the Fedora operations are local, rather than being aware that they are working with web services.
b. through the Fedora API-A web services' WSDL file which specifies the operation names, parameter types and number, and return types. Choosing this option essentially entails that the client side manually writes its own 'stub' class to deal with some of the technicalities of invoking Fedora's web service operations. It is similar to the GS3WebServicesAPIA class in Figure 4.2 that was written for Greenstone's web services. Such a class can define methods for all the Fedora web services that the client will use, where each method will merely handle the details of invoking the corresponding web service. If choosing this option and using Apache Axis to facilitate interaction with web services, the client end may either manually set the names of the operations and types of parameters and return values, or make Apache Axis' Service and Call objects do this work. Either way, the client has do the type-mapping themselves by registering the (de)serializer classes for the various Fedora types used by the web service operations.

Though Fedora's 2.2.1 release contains slight modifications in its method definitions compared to 2.1.1, each installation comes with updated stub classes that match with the corresponding web services. Therefore, since the stub classes will always remain up-to-date and in-sync with the Fedora installation, it was decided to go with the first option of using the included API-A stub class to access Fedora's web services, as opposed to manually writing equivalent classes.

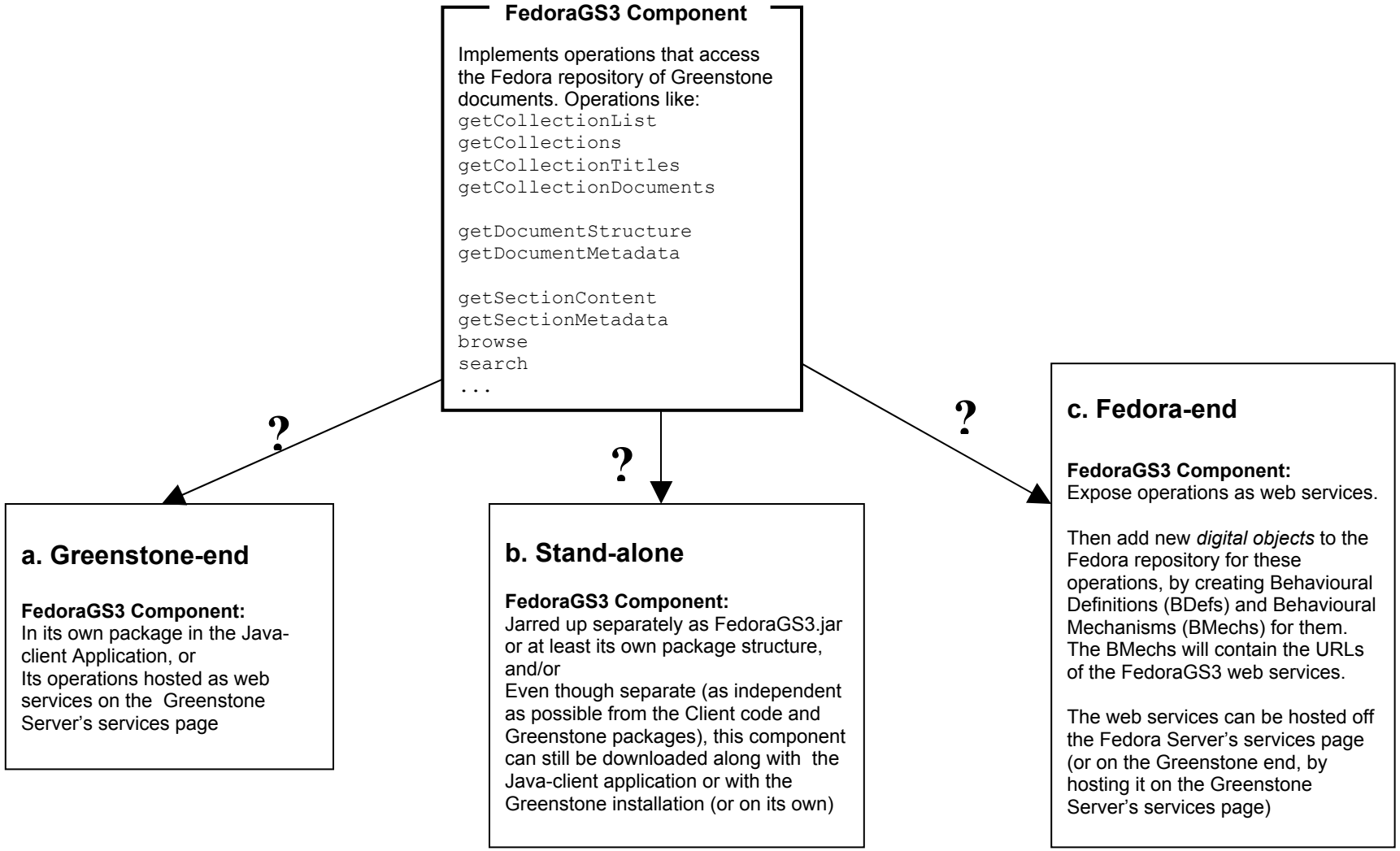


Figure 4.5: Design choices as to where to place the component that will facilitate custom access to the Greenstone documents stored in the Fedora repository

This is where the next significant design crossroads presented itself. There were several places to put this component ("Fedora-GS3") that was to serve as *an interface between Fedora and Greenstone*. See also Figure 4.5, which illustrates the following choices.

a. The component can be placed on the Greenstone 3 client application's end, in which case it merely becomes part of the sets of packages making up the Java-client for Greenstone 3. Choosing this option would involve integrating the calls to Fedora's Access web services into a separate package in the Greenstone 3 client application.

   Disadvantages: Though it is most easy to start implementing the interfacing component as part of the classes constituting the Java-client, it would be better were this component not tied to the Greenstone client application. For instance, if there were other Greenstone developers who might wish to build on it, they would have to download the Java-client in order to make use of the Fedora-Greenstone interface, FedoraGS3.

b. The component can be located on Fedora's end. As discussed in Section 2.1.3 of the System Review Chapter, Fedora not only allows us to store data in its repository, but also enables us to store *operations* that we can perform on the data. These operations need to be made available as web services and two particular kinds of digital objects need to be created for them: Behavioural Mechanisms (BMechs) that provide the implementation of methods defined in the Behavioural Definition digital objects (BDefs).

   Placing our intermediate component on Fedora's end involves:

   1. making the component's operations available as web services, and then
   2. creating Behavioural Definitions in Fedora that define the operations (BDefs which store the service operations' descriptions),
   3. creating Behavioural Mechanisms in the Fedora repository which will reference the web services that implement the Behavioural Definitions. BMechs contain the WSDL file for the web services.

   If going this route, there are some further options relating to where to expose the web services. The straightforward choices are to make them accessible either from the Greenstone 3 web services page or from the local Fedora installation's web services page (the page that links to Fedora's Access and Management APIs).

   Advantages: Next to it being a good experience to learn how to construct Fedora's BDefs and BMechs, it makes sense (conceptually) for the operations that directly work on Fedora's Greenstone digital objects to be available on Fedora's end as well. A client would then invoke the component's operations, which would convert the Fedora digital object data into a Greenstone-recognised format. Thus, from the client's perspective, nothing need be known about the internal datastream structure of the Greenstone digital objects as they are stored in Fedora.

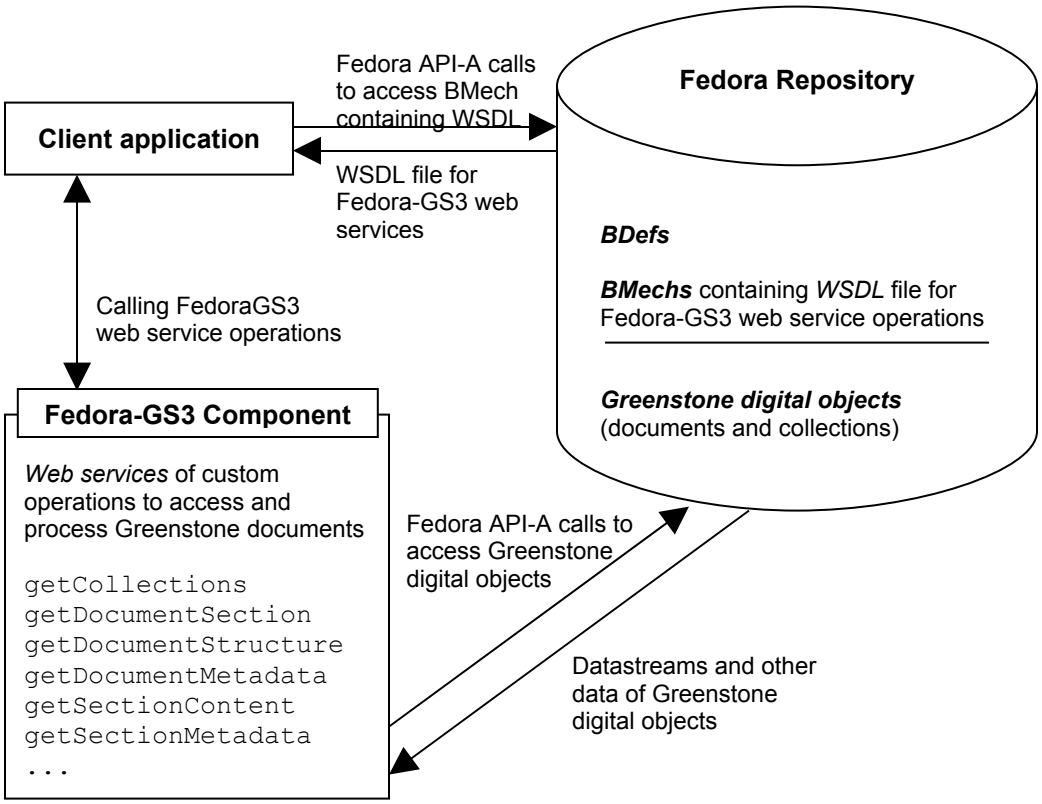


Figure 4.6: The web service invocations required to use the Fedora-GS3 component's operations, were this to be placed on Fedora's end

Disadvantages: Clients who wish to work with the Fedora-Greenstone interface component would be required to first make its operations available as web services. Thereafter, they will need to ingest the component's BDef as well as the BMech referring to its web services into the Fedora repository they wish to access. Ingesting BDefs and BMechs is certainly not hard, as it can be achieved either by using the Fedora Admin-Client application or through calls to Fedora's API-M web services. However, this extra step would require some little familiarity with Fedora itself on the client's behalf.

Furthermore, there would be the overhead of going through an additional set of web services in order to access the Greenstone digital objects in Fedora's repository using the interface component's operations. That is, when a client program wishes to use the FedoraGS3 component's web services, it would first have to invoke Fedora's API-A web services in order to locate and access the BMech associated with the FedoraGS3's web services. This will give the client access to the WSDL file for the component's web services. Once the WSDL file has been obtained, the component's operations are available for use. From this point forward, though, two further sets of web service invocations are required:

1. Because the component's operations would have been exposed as web services, calling any of them involves another web service call.
2. The component's operations work on the Greenstone digital objects stored in Fedora's repository. Thus, executing one such operation will cause the component itself to call Fedora API-A (Access) web service operations to

retrieve the Greenstone digital object's datastreams after which it will process and return them to the client.

Thus, if opting to turn our intended component's functionality into BMech-referenced web services, a call to the Fedora API-A is needed to first access the component, after which two web service invocations are required for each operation of the component —as depicted in Figure 4.6. This is as opposed to just the one web service call to Fedora's Access API per operation that would be required if the component were not placed on Fedora's end.

Figure 4.6 shows how the client application must also interact with the Fedora API-A. This means that the client application has to establish a connection to Fedora, import Fedora's custom data types (which are returned by Fedora's web services) and be familiar with Fedora's web services as well as FedoraGS3's operations.

c. Fedora-GS3 can be considered a somewhat separate component, one that is kept a little distinct from both Fedora and Greenstone. The intended Fedora-Greenstone interface component could, for instance, be made into a Java jar file. Since it is then not bound to the Greenstone 3 Java-client, other clients can use it too. Upon updates to Greenstone 3, this jar file may be modified correspondingly and clients who wish to use it can download it separately.

   This design option does not involve creating or using BDefs or BMechs. Therefore it is no more difficult to accomplish than were the component to be incorporated into the Greenstone 3 client application as in design option (a). Next to that, calling any of the component's operations would only require invoking one set of web services: the Fedora API-A calls necessary to access the Fedora repository's Greenstone digital objects. Finally, the Greenstone 3 client application need not make any invocations to Fedora's web services—in fact, it need not know about Fedora at all, only about FedoraGS3.

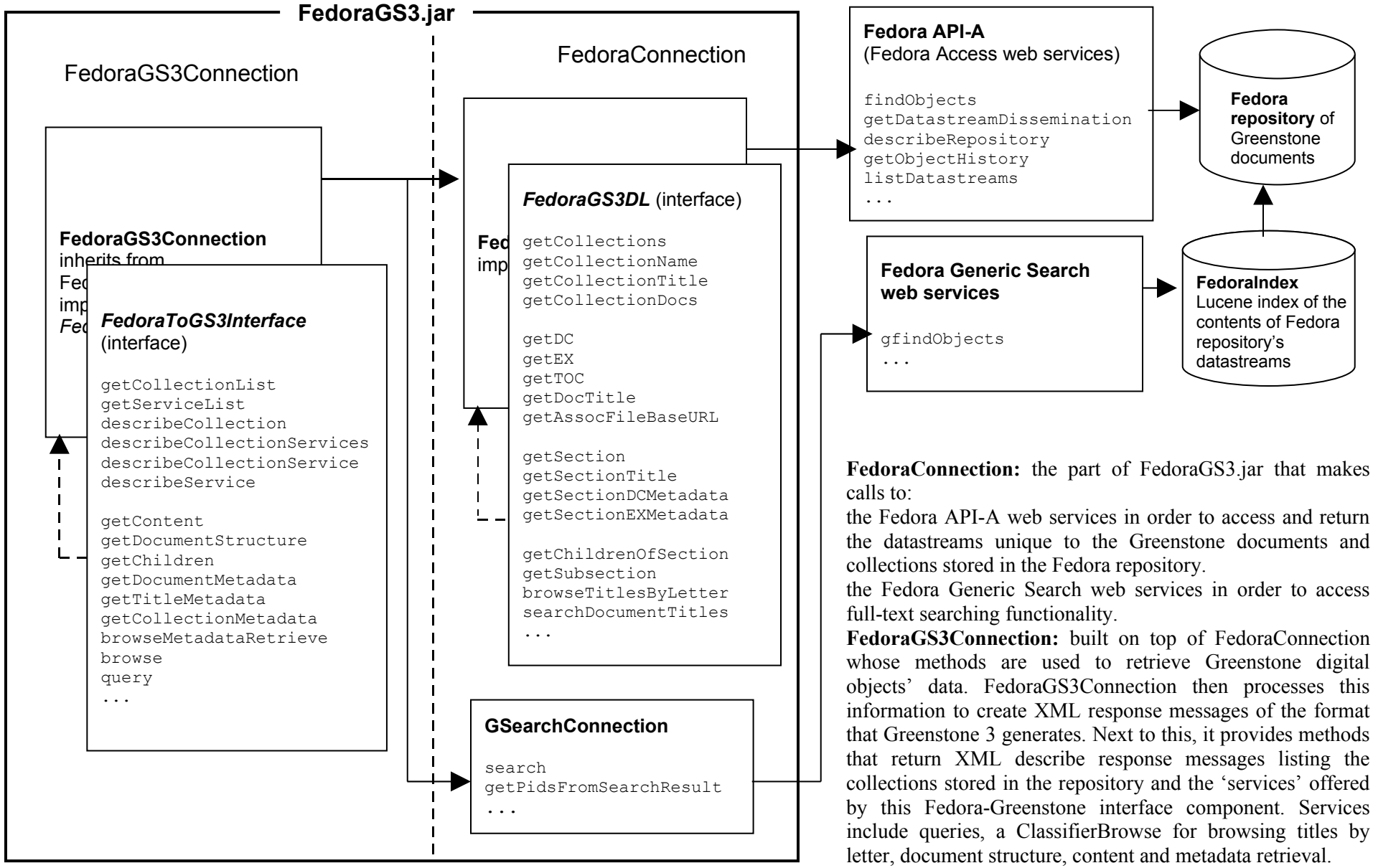


Diagram 4.7 FedoraGS3.jar

FedoraConnection: the part of FedoraGS3.jar that makes calls to:
the Fedora API-A web services in order to access and return the datastreams unique to the Greenstone documents and collections stored in the Fedora repository.
the Fedora Generic Search web services in order to access full-text searching functionality.
FedoraGS3Connection: built on top of FedoraConnection whose methods are used to retrieve Greenstone digital objects' data. FedoraGS3Connection then processes this information to create XML response messages of the format that Greenstone 3 generates. Next to this, it provides methods that return XML describe response messages listing the collections stored in the repository and the 'services' offered by this Fedora-Greenstone interface component. Services include queries, a ClassifierBrowse for browsing titles by letter, document structure, content and metadata retrieval.

We chose the 3rd route for the placement of this Fedora-Greenstone interface component. The stand-alone component was turned into a jar-file and called FedoraGS3.jar. For purposes of clarity and general good programming practises, its functionality was separated into two main parts, as also depicted in Figure 4.7:

1. *FedoraConnection:* this part uses the Fedora Access API to make calls to Fedora's functionality and uses Fedora Generic Search's web services to perform full-text searching.
    - Through Fedora's Access API, it will provide the functions necessary to obtain the various custom datastreams associated with a Greenstone digital object. These include the datastream for a document's table of contents which outlines the document's structure, for the EX, DC and DLS metadata of the document, the datastreams for the various sections, for the associated files, for the EX metadata of each section, and for each section's DC metadata.
    - Through the Fedora Generic Search web services, it will provide the functions necessary to do straightforward text searching and fielded text searching.
2. *FedoraGS3Connection:* this part was to serve as an adapter class—one that would convert the format Fedora uses to communicate the data in, into the Greenstone format that the Java-client understands. It would entail processing the various datastreams obtained through FedoraConnection and piecing them together again into the documentNodes that make up a Greenstone document. It was also to provide services such as for browsing titles by their first letter, and text query and field query services. Finally, FedoraGS3Connection would contain methods that return the kind of XML response messages as are returned by Greenstone 3.

The FedoraGS3Connection part of the component *FedoraGS3.jar* is what our Java-client will be dealing with.

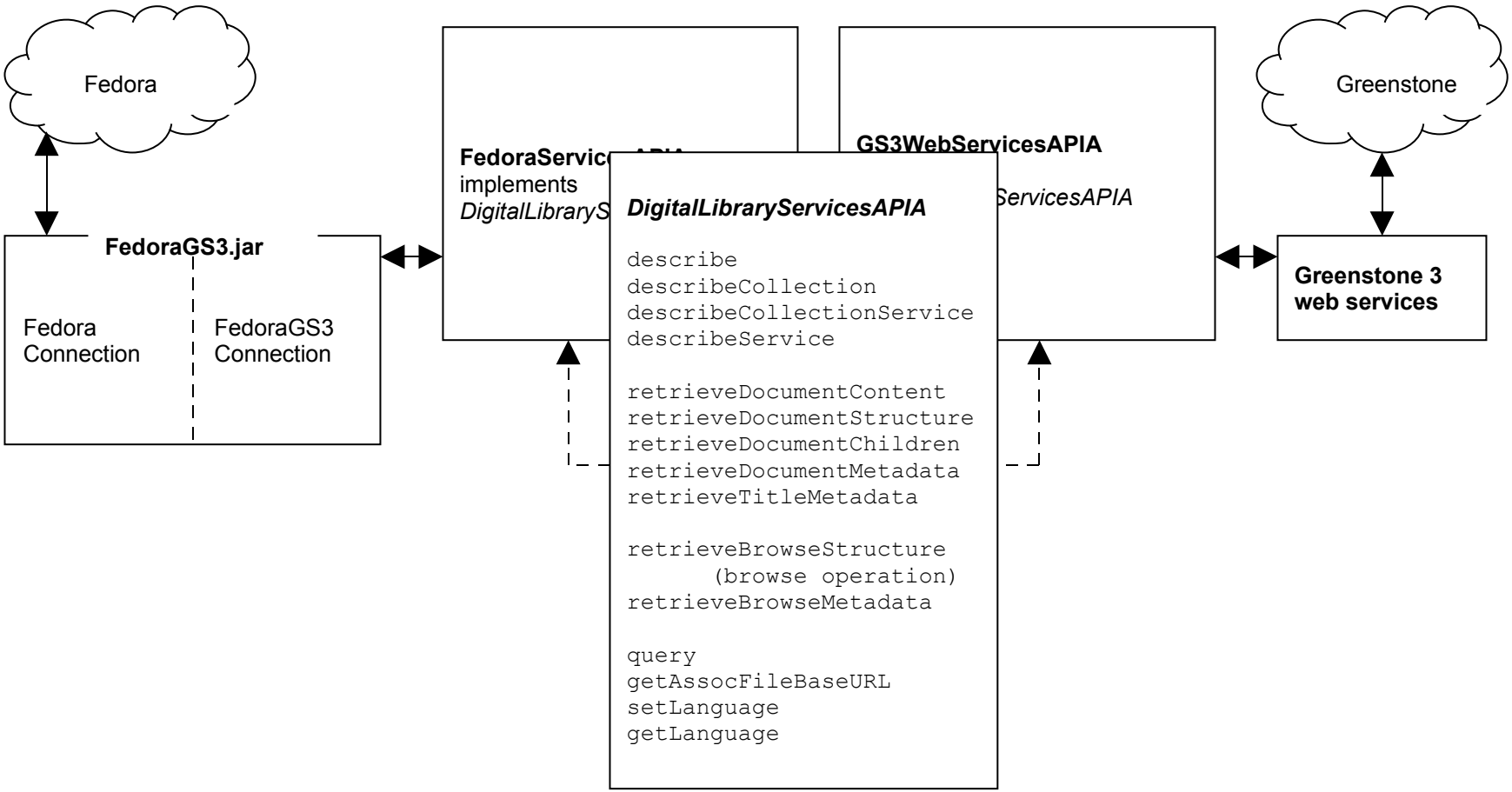


Figure 4.8: The DigitalLibraryServicesAPIA interface: to enable uniform access to both the Fedora- and Greenstone-backed repositories

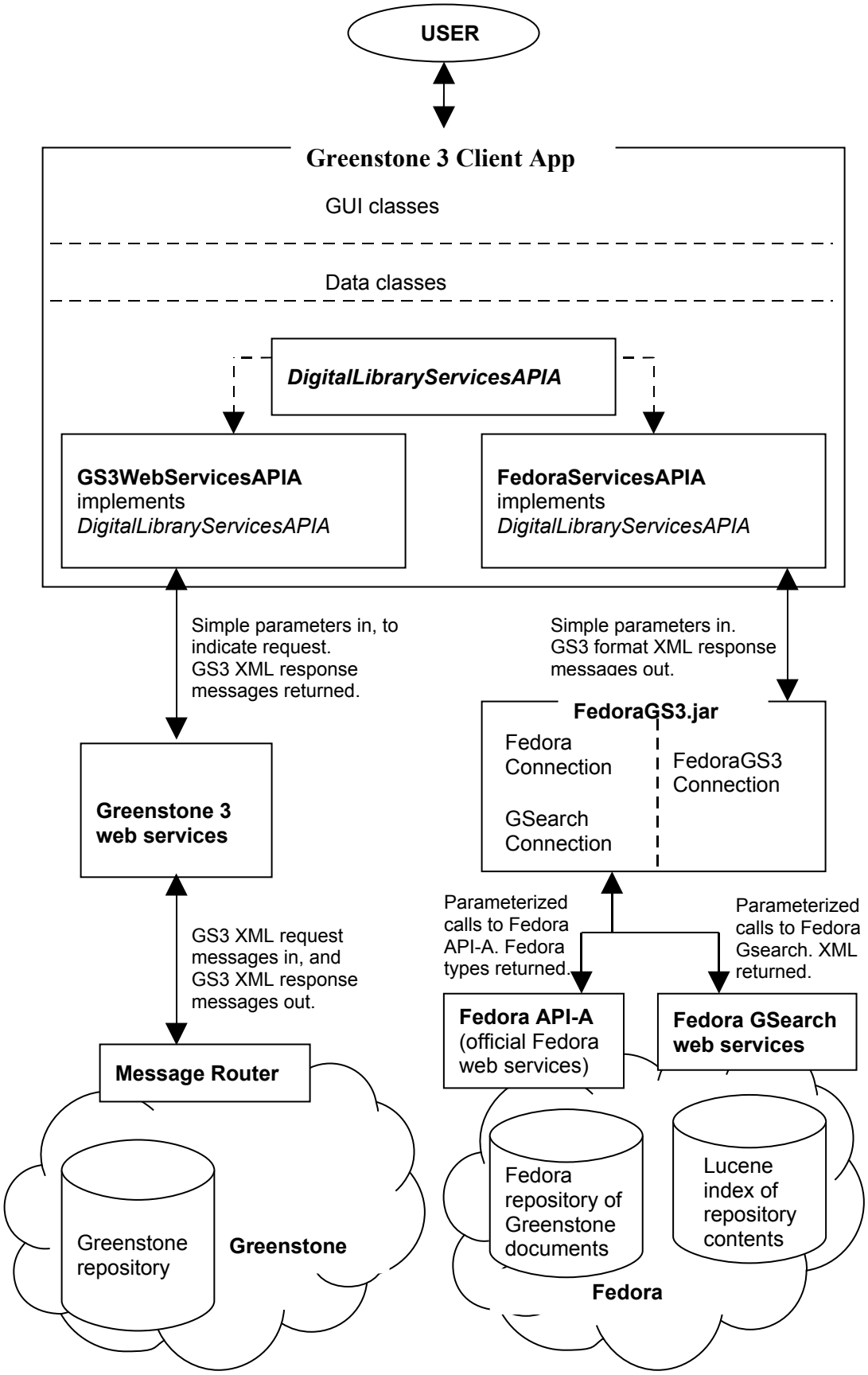


Figure 4.9: How the three parts of this project are connected

4.3.3. Integrating access to the Fedora repository into the Java-client

To facilitate a more general approach to accessing both the Greenstone and Fedora repositories, a Java interface class was created *in the client application* to represent access to a generic digital library. This interface defined methods that would become common to both:

- the FedoraGS3Connection part of the FedoraGS3.jar component shown in Figure 4.7, and
- the GS3WebServicesAPIA class of the Java-client that is depicted in Figure 4.3 (which interfaces with Greenstone 3's web services).

The new Digital Library Services Access API, *DigitalLibraryServicesAPIA*, would act as intermediary between the client application on one end, and the classes/components dealing with the two digital libraries on the other end. As such, the Java-client's GUI and Data classes would not need to know which repository they were dealing with, as long as the operations defined by this interface class were implemented by both. This was accomplished by:

- making the Java-client's *GS3WebServicesAPIA* class implement *DigitalLibraryServicesAPIA*, and
- creating *FedoraServicesAPIA:* a new class in the client application which also implemented the *DigitalLibraryServicesAPIA* interface by making calls to the appropriate operations of the Fedora-Greenstone interface component (*FedoraGS3.jar*).

This is illustrated by Figure 4.8.

When implementing this step, one advantage of having chosen to make the web services for Greenstone 3 return the XML response messages as-is became more apparent. Now the *FedoraGS3Connection* part of the Fedora-Greenstone interface component likewise contained operations that returned XML response messages of the same format that Greenstone 3's Message Router returned. This allowed us to reuse the Java-client's data classes – that is, reuse the existing processing methods that parsed Greenstone 3's XML response messages and stored their data. This meant that the client application was able to present the data obtained from *FedoraGS3.jar* to the user from within its GUI without any changes to the existing code.

By way of summary, Figure 4.9 shows how the 3 parts of this project (Greenstone 3 web services, Java-client and integrated Fedora access) are connected.

Chapter 5. Evaluation

In this project, designing and implementing the Greenstone 3 web services, the Java-client, and the Fedora-GS3 component received much of the focus. Some cursory evaluation has been done, though—usually upon integrating the three parts of this project. For instance,

- The web services implemented for Greenstone 3 mostly covered operations pertaining to accessing the repository. Since the Java-client only made use of the access-oriented Greenstone functionality (like querying, browsing and retrieval of document data), the set of web services that were provided could be partially evaluated upon implementing this client application.
- The FedoraGS3 component's ability to provide the operations required by the Java-client could be assessed when creating the Java-client's FedoraServicesAPIA class. This class, which implemented the *DigitalLibraryServicesAPIA* interface, would call the appropriate FedoraGS3 methods. This meant that FedoraGS3 needed to provide all the functionality defined by the *DigitalLibraryServicesAPIA* interface. The Java-client also implemented this interface for Greenstone's web services in the class GS3WebServicesAPIA. In this way, the interface acted as an indirect check that FedoraGS3 provided comparable functions to Greenstone 3's web services.
- The XML response messages returned by the FedoraGS3 operations could be evaluated by checking whether they smoothly integrated into the Java-client's existing data structures that parsed and stored the data contained in Greenstone's XML response messages.

5.1. Evaluating the web services for Greenstone 3

This section covers the *evaluation of the web services of Design Stage 1* (Section 4.1.4), listed in Table A.1 of the Appendix. The ideas for improvement that came up during this evaluation were incorporated into Design Stage 2 (also Section 4.1.2), so that the final, improved list of web services for Greenstone 3 can be seen in Table 4.1.

In this section we will look at:

- Evaluating some of the design decisions related to the web services. This includes looking at whether the choice of simple data types as parameters and of XML response messages as return values posed any particular difficulties. And whether any drawbacks could be remedied, or the current design improved upon, through use of alternative data types.
- Whether exposing just the access-related operations as web services is enough—while leaving clients to turn to the Message Router's `process` web service for the rest of Greenstone's functionality—or whether creating easy-to-invoke web services for some or all the remaining publicly-accessible Greenstone operations would be better.
- Comparing the web services implemented for Greenstone thus far with the breadth of operations provided by Fedora. Looking at Fedora's own web service

APIs will not only give a more general picture of the sorts of functionality Greenstone's web services could provide, but also allow a comparison of the Greenstone access operations and those exposed by the Fedora Access API.

5.1.1. Evaluating the design of Greenstone 3's web service method definitions

The use of simple data types for method parameters

As mentioned in Chapter 4 System Design and Implementation, the Greenstone web services use basic data types for their parameters. When it came to implementing the Java-client which invoked these web service operations, passing simple values as arguments to the method calls was indeed easier than constructing the XML request messages expected by the Message Router. As anticipated, it made it easier and less complex to have this part of the work taken care of by the web service itself.

Fedora's web service method definitions also have simple data types as do FedoraGSearch's. Calling their web services with simple parameters (as opposed to data structures) from within the FedoraGS3 component was similarly easy.

Returning XML response messages from web service methods

The fact that Greenstone's web services returned XML response messages meant that the implementation of the Java-client involved writing classes to parse and store the data contained in the response XML. This is because the choice of return type forced the parsing work to be done on the client's side instead of by the web services. However, once the necessary Data classes were written for the client, its remaining classes were able to perform their tasks without troubling themselves about the parsing details. One disadvantage that inevitably follows from this is that, in spite of the Data classes taking care of the parsing work for the rest of the Java-client, other potential clients of Greenstone 3's web services will have to either write their own parsing code, or else include the Java-client's Data classes.

Yet, although it would have been easier for the Java-client had the web services returned the required data structures instead of the XML response messages, the main problems mentioned in Section 4.1.3 of the Chapter on System Design and Implementation would still have applied:

- some Greenstone operations can return XML representing differently structured data in different situations. This means that the same operations would have to be able to return quite different data structures at different times, had we chosen to return data structures instead.
- other clients of Greenstone 3's web services may not have much use for the exact data types being returned (which the Greenstone 3 Java-client uses). They might have need for only a subset of the data and might therefore prefer to have direct access to the XML response message in order to parse out just the data they require.

- If custom data types were returned, then the compiler would need to recognise the return types. This would require all client applications to write their own class definitions to map the fields in the complex data types returned. The alternative to this would require that other Java-based client applications include the Data classes of Greenstone 3's Java-client by defining type-mappings to specify which web service return value maps to which data structure defined on the client-end.

In comparison, Fedora's own web services return custom Fedora types, which are certainly very convenient. On writing the FedoraGS3 component's *FedoraConnection* class, which made calls to Fedora API-A, being able to use the Fedora types returned from the Fedora operations made the process of using the return values of web services simpler. However, that is because some of the underlying difficulties involved in doing this were smoothed away behind the scenes by Apache Axis which handled the type-mapping and serialization of SOAP's simple data types into the complex objects that were received by the client (FedoraGS3, in our case).

In fact, there were two specific reasons for why calling the Fedora API-A services was particularly easy:

1. Apache Axis generated stub classes for Fedora web services. These stub classes deal with invoking the actual API-A (and if required, API-M) web services. The stub classes correspond to the one manually written for Greenstone: GS3WebServicesAPIA, which would similarly handle all the details of web service invocations. But whereas the Greenstone 3 web services took simple types as parameters and returned a simple type (always a string representing an XML response message), the situation in Fedora's case was actually more complex. The stub classes Axis generated for Fedora were able to recognise the complex types that the web service operations returned because of the type-mappings specified in Fedora's WSDD (web services descriptor file) which are set on installing Fedora. The type-mappings specified the custom serializer and deserializer classes associated with the various custom Fedora types involved. Fortunately, as all this work was left to Axis and to Fedora's installation process, the FedoraGS3 component—which was a client of Fedora API-A—did not have to be burdened with it.

2. Even though the type-mappings were sorted out and so too the classes handling the custom (de)serialization of the Fedora types, the definitions of these custom Fedora types needed to be included in FedoraGS3. Fedora's administration tool—a GUI application that is a client of Fedora's API-A and API-M web services—defines Java classes for all the required Fedora types. (Since this admin-client application ships with Fedora's installation, it would doubtless be up-to-date with any changes to Fedora's web services.) The Java classes for the Fedora types that were included in the Fedora admin-client were imported in FedoraGS3 in order to convert the values returned from the web services into appropriate data structures. (The alternative would have been to write our own data structures to map all the fields defined by each Fedora type and adjust the type mappings in the WSDD.) Just as our FedoraGS3 component used the Fedora admin-client's Fedora types to map the complex data types returned from Fedora's web services, others may choose to use the data classes

of Greenstone 3's Java-client to process and store the data in the XML response messages returned from Greenstone 3's web services.

Unlike Fedora, and rather like Greenstone, FedoraGSearch returns XML data—in string form. There appear to be two intentions behind using XML return values in their case:

1. Clients can apply an XSLT transformation to convert the XML for display. For instance, FedoraGSearch's XML response for its search operation `gfindObjects` represents the results of a search. By using an XSLT file, this can be turned into a different XML file or into XHTML for displaying the search results formatted for user-viewing.
2. The XML can be dealt with directly by client applications in any other way they see fit (no XSLT need be applied). For instance, clients can do custom parsing to obtain the data they want and then may choose to store or present that data.

(At least, the second explanation above is the way in which FedoraGS3 made use of FedoraGSearch's search operation.)

The advantage of returning an XML response from web services is that developers may immediately see the structure and data contents of the XML. This transparency in the information conveyed means that developers can choose to parse the response XML to retrieve only the elements of data they require. This was certainly the case with us when FedoraGS3 parsed only PIDs out of the long XML response message returned from `gfindObjects`. All the data is there and clearly presented in the response XML—just as it would have been in custom data structures were these used—but with the additional advantage that only the simple data type of string is returned. No custom (de)serialization is required and no type-mapping is necessary.

While it was helpful to receive custom data structures from Fedora API-A operations, it makes equal sense in FedoraGSearch's case and that of Greenstone 3's web services that they return XML to the client.[12]

A small improvement has suggested itself. To help Java-based developers along, the Data classes in Greenstone 3's Java-client can be turned into a separate jar file that all clients (including the Java-client itself) can import and work with if they choose. By making these Data classes available separately as well as in the Java-client installation, they become a stand-alone component just like the FedoraGS3.jar. In this way, developers of any Java-based clients of Greenstone 3's web services who would like to leave the parsing to the existing Data classes, need not download the Greenstone 3 Java-client itself in order to access them.

[12] Personal communication by Eric, Chu-Hsiang in January 2008. Eric is a research student developing an application using Greenstone 3 functionality. He was informed about the way its web services were designed: how various operations available through the Message Router's `process()` method were mapped into individual web service operations, and the fact that these operations would take simple data types as parameters instead of XML request messages, while they would still return the XML responses. He was asked his opinion on whether he would find the simple data types for parameters helpful, and whether he preferred data structures to be returned instead of XML response messages. He approved of the choice of simple data types as parameters, and said that he preferred the XML to be returned. He specifically noted that he would *not* find return values that were complex data types (data structures) helpful.

Method overloading

As can be seen in Table A.1 of the Appendix, which lists the implemented web service methods for Greenstone 3, a lot of method overloading was used in order to provide end-users with alternative ways of calling an operation as may suit their circumstance.
Although having many variants for a single Greenstone operation may simplify some of the work for clients, some operations have as many as five variants (e.g. the query operation) that essentially accomplish the same thing.

The web services for Fedora and FedoraGSearch do not use method overloading. Programmatically specifying default values for parameters, so that a single method definition may suffice instead of many, is not possible in Java as it is in C++. Even so, Fedora and FedoraGSearch overcome this discrepancy by allowing clients to pass *null* or the *empty string* as default values for optional parameters. The web services will then set these internally to some meaningful fixed value or ignore. One disadvantage to this is that developers still need to know beforehand that they can pass null or the empty string for some parameters—something that is certainly not apparent from inspecting the WSDL when other documentation is unavailable—but after some trial and error one can work this out (as was necessary when using FedoraGSearch's web services during this project). Fedora and FedoraGSearch's way of coping with optional parameters may be a way to avoid too many instances of method overloading in the web services for Greenstone, even if not all of the overloaded methods can be dispensed with.

Greenstone's web services of Design Stage 1 (Section 4.1.4) included operations that returned a String enumerating the specific set of values accepted for certain parameters, which developers can resort to when they do not know what is required. Instead, we can have one single help operation that returns a string describing how to invoke certain other web service operations (such as those with optional parameters). This help method can also indicate to developers the occasions where Greenstone clients may pass null, or the empty string, or some other default value as parameters instead.

5.1.2. Using the Java-client to evaluate the design and completeness of Greenstone 3's Access-related web services

The list of intended web services were decided upon beforehand and most of the method definitions (especially the list of parameters) were designed first. The Java-client was written thereafter. Implementation of some of the web service operations was left until the point had been reached where the client application needed those operations, in order to make sure that, at each stage, the web services thus far implemented worked in unison with the evolving client code that worked with them. As such, the client application was mostly built around Greenstone 3's web services. Even so, it was possible to perform some minor evaluation as to whether the available web services and their method definitions were suitable for the client as per their initial design or whether they could be made even easier to invoke. This section will cover such findings.

Initially, the client made direct method calls to the Java class whose methods would soon be exposed as web services (*GS3WebServices*). Whenever that worked fine, that class' public methods were re-exposed as web services and the client was redirected to go through the web services instead. This step afforded some confirmation that the web services were indeed functioning as they should—that is, that they were working just as before, when the local class *GS3WebServices*' methods were called instead.
Next to that, the list of proposed web services proved sufficient for the Java-client (or rather, more than sufficient, as discussed below). The number and types of parameters – of those methods that the client application invoked—were convenient to use.

A couple of improvements did suggest themselves during this early phase of implementing the client application:

- Due to method overloading, there were more method variants available for a single Greenstone operation than the Greenstone 3 client application needed. Some of the variants were only slightly different from the one the client made use of. It took some consideration as to whether to reduce such variants to one method or leave them all in, as it was conceivable that another client might opt for the alternative variant to the one used by the Java-client. In the end, it was considered on a case-by-case basis.
  - For example, the `query` operation is mapped into a web service method that takes two arrays: one for the query field names and the other for the values associated with those fields. Another web service method for the same `query` operation takes a HashMap containing mappings of field names to field values instead. This last version of the operation was used by the Java-client, even though it's possible that another client may find using the first-described version easier.
  - On the other hand, some of the many variants of the describe operation were found to be near duplicates of others, and left out of the redesign of Design Stage 2 (Section 4.1.4).
- For some Greenstone operations, however, there were additional method variants that looked like they might not be required by the Java-client after all—in any situation. These mostly happened to be the kind where one of the web service method's parameters received a proper segment of an XML request that contained just the data particularly required by the underlying Greenstone operation. The web service method that was being invoked would construct the rest of the request message around that XML segment. Methods like this that take parameters in XML format might be superfluous after all. Continuing with the example for the `query` operation, there are method variants for the same that take an XML string. This XML would contain the query field names and field values in a format that can readily be inserted into an XML request message to be sent to the Greenstone Query process-type operation.

After implementing the FedoraGS3 component, the *DigitalLibraryServicesAPIA* Java interface was conceived of and incorporated into the Java-client. This interface was implemented by both the Java-client's FedoraServicesAPIA and GS3WebServicesAPIA classes and defined just the set of methods that the Java-client needed to access both Fedora and Greenstone in a consistent manner. The *DigitalLibraryServicesAPIA*

specified far fewer methods than were designed for Greenstone 3's web services. In many respects, its more concise listing of the minimum operations that were expected from both digital libraries was better (less confusing) than the far larger number offered by Greenstone 3's web services. The method definitions of *DigitalLibraryServicesAPIA* gave ideas on further limiting the set of web services. There would still be some additional methods offered by Greenstone 3's web services beyond those required by the Java-client, while a few listed in *DigitalLibraryServicesAPIA* would not be represented. And some of the parameters may be slightly different between them (for instance, many of the web service methods would still take a `language` parameter, while in *DigitalLibraryServicesAPIA* the `setLanguage()` method would be the only one concerned with language). Overall, though, the *DigitalLibraryServicesAPIA* served as a good guide in redesigning Greenstone 3's web services to a "more general but useful" set.

5.1.3. Comparing Greenstone 3's web services with the breadth of services provided by Fedora

Comparing Fedora's web services with those implemented for Greenstone may prove a useful way to evaluate the latter's web services since both are software used to build digital libraries with. Therefore, in spite of Fedora being more general when it comes to designing digital libraries, the two may be somewhat comparable when one considers the overall functionality each offers. Any web service operations that are present in Fedora and whose equivalents are missing in Greenstone 3's set of web services, can serve as a reason to consider incorporating something similar in the latter (where they are offered by Greenstone 3 itself).

The two sets of web services offered by Fedora are the Management and Access APIs (API-M and API-A). The first deals with managing the repository and includes operations for such processes as ingesting / adding digital objects into the Fedora repository and purging / deleting them from it. The second API is concerned with giving clients access to the existing digital objects in the repository.

Section 5.1.1, which was on evaluating the web service method definitions, had already compared the types of the parameters and return values that Greenstone's web service methods used with those used by the Fedora and Fedora Generic Search APIs. What remains is to compare the breadth of services offered by the last two with those made available for Greenstone 3. First to be considered are the Access-focussed operations of Fedora and Fedora Generic Search, as these are the sorts of operations that have thus far been exposed as web services for Greenstone 3.

Fedora Generic Search provided the built-in search services for full-text and user-added metadata that Fedora lacked. Therefore, Fedora Generic Search's web services will be considered along with those offered by Fedora's Access API, in order make a more complete comparison of the public operations that can be performed on Greenstone and Fedora repositories.

Table 5.1 Comparing the web service methods in Fedora API-A with their equivalents (if any) in Greenstone 3's set of access-related web services

Fedora API-A	Greenstone 3's Access web services

`describeRepository`	`describe` (for the MessageRouter itself)
	`describeCollection`
	`describeServiceRack`
	`getBaseUL`

`findObjects`
`resumeFindObjects`
`gFindObjects` of Fedora Generic Search
`browseIndex` of Fedora Generic Search
	`query`
	`browse`

`getDatastreamDissemination`	`retrieve` operations:
	`documentContentRetrieve`
	`documentStructureRetrieve`
	`documentMetadataRetrieve`
	
	`(getAssocFiles`
	This is meant to return the list of files associated with the document that are specified in the metadata for the document root's. Once Greenstone 3 is made to leave the GSDLAssocFile metadata in the response message, the getAssocFiles web service method should work.)

`getDissemination` This may return a stored (static) datastream or else a dynamically generated one. Either way, a datastream associated with the object is returned. In the second case, the requested dissemination will apply the appropriate BMech associated with the object to the stored datastream which transforms it into another format. The result is then returned.	There is no complete equivalent, but it is partially covered by the `retrieve` operations. Some of the additional functionality might possibly be compared to applying the transformation that is specified by a `format-type` response message to the documents retrieved. `Format-type` request messages are sent to the applicable Greenstone service (such as query) and they return XML defining how to transform a document to present its contents for that service. (The XML may be XSLT or Greenstone Format, GSF. The latter needs to be converted into XSLT first.) This is explained in the Greenstone 3 Developer's Manual, p.42 [8].
`getObjectHistory`	Not applicable, since Greenstone 3 does not do versioning of contents

`getObjectProfile` Basic information on the digital object: some Fedora-specific metadata fields (like PID) and URLs for the object's Dissemination Index and Item Index.	No direct equivalent, but the document's identifier and other Greenstone-specific metadata fields associated with a document can be obtained with `documentMetadataRetrieve`
`listDatastreams`	No real equivalent. However, sending a `describeCollection` to the collection will also list the `retrieve` services provided for it. A `documentStructureRetrieve` message on a document-root requesting *all* the descendants will return a response listing all the internal and leaf document nodes. A `documentStructureRetrieve` will list the associated files, if any.
`listMethods` Lists the method definitions for all the possible disseminations that can be run on the specified digital object	`describeCollectionServices` This lists the functionality (the services) available for all documents in the specified collection. The available behaviours (functionality) for a collection's documents are specified by sending a `describeService` request to all services of the collection.

Table 5.1 lists the access-related web services of Fedora and Fedora Generic Search next to any comparable ones in Greenstone 3's web services. Where none is applicable (for instance, if it does not apply to Greenstone 3's data model), this is stated. The table shows how most of the operations provided by the Fedora API-A have some sort of comparable methods in Greenstone 3's web services, while some have no real equivalents. The table also shows that the functionality provided by Fedora Generic Search web service methods `gFindObjects` and `browseIndex` have corresponding operations in the web services of Greenstone 3.

One discrepancy highlighted by the preceding Table 5.1 is that Greenstone 3's `format-type` message operation had not yet been mapped into a web service.

Table 5.2 Methods of the Fedora Management API (API-M) that have some comparable functionality to what's available in Greenstone

Fedora API-M methods	**Somewhat comparable Greenstone 3 functionality**

`addDatastream`
`addDisseminator`
`ingest`
	`addDocument` Service
	`buildCollection` Service
	`importCollection` Service
	These services are available by sending `process-type` messages to the Message Router.

`export`	`export.pl` Perl script

`purgeDatastream`
`purgeDisseminator`
`purgeObject`
	(The `deactivate` operation on Collections and Service Clusters or individual Services do not delete collections.)
	The Greenstone Librarian interface application gives users the ability to delete a site's collections.

`getObjectXML` XML metadata datastreams associated with a digital object will be included in the data returned, but content datastreams will not be	`DocumentMetadataRetrieve`

`getDatastream`
`getDatastreams`
`getDisseminator`
`getDisseminators`
	`retrieve` operations:
	`documentMetadataRetrieve`
	`documentContentRetrieve`
	`documentStructureRetrieve`

`modifyDatastreamByReference`
`modifyDatastreamByValue`
`modifyDisseminator`
`modifyObject`
	The Greenstone Librarian Interface application gives users the ability to edit collection data

Though many of Fedora's API-A operations offer behaviour comparable to what's possible via Greenstone 3's Message Router, the differences with Fedora's API-M are more numerous. The Fedora API-M web services methods for which there is no equivalent Greenstone functionality are:
`describeUser`, `compareDatastreamChecksum`, `getDatastreamHistory`, `getDisseminatorHistory`, `getNextPID`, `setDatastreamVersionable`, `setDatastreamState` and `setDisseminatorState`.

Many of these concern versioning. The Fedora web services for which some sort of comparable functionality exists in Greenstone (even if not all of them are as yet available through interaction with the Message Router) are shown in Table 5.2.

Is an Access API sufficient for Greenstone 3?

Since this project involved building a Java-client alongside the web services for Greenstone 3, the implementation stage became more focused on the parts of Greenstone 3's functionality that similar client applications would require: the operations for accessing the contents of the repository, such as those dealing with querying, browsing and retrieving data. Though it is likely that the main objective of most clients would concern accessing the documents stored in a Greenstone-backed digital library, it is quite possible that some would want to give end-users the ability to carry out the remaining operations made available by the Message Router. For instance, Table 5.1 showed that the `format-type` message may be useful to provide. Some of the remaining operations are related to managing the repository while there is no particular category for the others. Although no *individual* web services had been implemented in Design Stage 1 for the remaining Message Router-facilitated Greenstone 3 operations, they could still be accessed—but only via the original `process` web service, which requires clients to construct request messages themselves.

The original `process` web service operation was complete when it came to mapping all the Greenstone 3 operations that are offered through the Message Router. In comparison, the web services that were implemented during Design Stage 1 of this project were more focussed on enabling easier *access* to Greenstone's repository. The question that came up is whether it may be useful after all to map more of the remaining Greenstone operations into easy-to-invoke web services, or whether having convenience methods for just the Access-related operations is enough.

The fact that another digital library software like Fedora exposes *all* its major operations as individual web services—rather than just those related to accessing the repository, i.e. the API-A—means that it anticipates that all of them could potentially be useful to clients. Although Greenstone's core functionality was already public due to the Message Router's `process` method being mapped as a web service, the convenience of invoking the distinct methods offered by the Fedora APIs provides a compelling argument in favour of creating individual web service methods for the major Greenstone operations remaining to the Message Router. Even though clients may not need to use some operations as frequently as others, providing stand-alone web service methods for each will ensure that clients can call them if they ever need to, while still having the benefit of invoking them with simple parameters (instead of constructing the request messages themselves).[13]

[13] Personal communication by Eric, Chu-Hsiang in January 2008. This research student—who had earlier provided his view on the design of the web service method definitions—is implementing a statistical tool making use of Greenstone 3's functionality. At one point he wished to reconfigure the Message Router dynamically (without having to shut down the Greenstone server). The `reconfigure` operation is actually available by sending a reconfigure `system-type` request message to the Message Router. The fact that a developer working with Greenstone 3's capabilities did need his program to access a `system-`

5.2. The Java-client for Greenstone 3

The Java-client was not meant to, nor does it, reproduce the complete functionality of the browser interface for Greenstone 3. Its purpose is rather to serve as an example GUI-based application that would demonstrate one of the ways (as indicated in the system's Developer's Manual, [8]) in which a client program for Greenstone 3 could be constructed. As a consequence, usability studies evaluating the intuitiveness of use of the client's interface are for the most part out of scope.

As per its current implementation, the Java-client does enable users to perform the basic operations that are possible through the browser interface, as could be seen in Chapter 3, Extended Worked Example. For instance, the query forms generated by the application are in accordance with what is specified by the `describe response message` returned by the Greenstone 3's Query services. As such, they display the same or similar form controls as are presented by Greenstone 3's browser interface.

The Java-client part of the project has achieved the objective laid out for it, without requiring any significant modifications, except one. As mentioned in Section 5.1.1, one of the ideas that came up when evaluating the web services was to package the Data classes, originally created for the Java-client's use, into their own separate jar file. This way, other developers, if they ever have need for it, can download the jar separately and let it to do the parsing of the response messages returned from Greenstone 3's Access-related operations.

5.3. The FedoraGS3 component

As already mentioned in Section 4.3.3 of the System Design and Implementation Chapter, the decision to make Greenstone 3's web services return the XML response messages emanating from Greenstone's Message Router turned out to be very useful when it came to integrating Fedora access into the client. The data returned from the FedoraGS3 component was of the same specific XML response message format as those returned by Greenstone 3's various services—the message format for query, browse, and document content, structure and metadata retrieve. As such, the existing parsing and data storage classes on the Java-client's end were able to work with Fedora's XML response messages without requiring modification. As a result, it turned out to be quite an easy process to integrate the FedoraGS3 component into the Java-client.

Upon inserting the intermediate *DigitalLibraryServicesAPIA* interface and *FedoraServicesAPIA* class in the Java-client (see Figure 4.8), the prime objective of the third part of the project was accomplished as well: a single user-interface working with repositories backed by two separate digital library systems.

Even though the Greenstone 3 Java-client requires only those parts of FedoraGS3's functionality that are made available through the Java-client's *DigitalLibrary-*

`type` message after all, indicated that such operations might also prove useful to map into stand-alone web service methods (even if they might not always be invoked as frequently as the Access-related ones).

ServicesAPIA interface, the FedoraGS3 component provides a great many more public methods in its *FedoraConnection*, *FedoraGS3Connection* and *GSearchConnection* classes. Being general and useful, these methods were kept public in order to give access to the component's intermediate processing functions. For instance, the public methods of class FedoraConnection return raw values that have not yet been built up into a Greenstone XML response message. Perhaps some of these methods may be useful in the future or for other Java-based developers, in which case the FedoraGS3.jar file can be included.

5.3.1. Limitations

There have been some impediments to achieving complete functional interoperability between Greenstone 3 and Fedora.

The Fedora repository contained Greenstone document and collection data stored in a custom format such as would allow the FedoraGS3 component to easily access and return requested data to clients in the same format as Greenstone 3's XML response messages (see Tables 4.2 and 4.3). However, in spite of representing the internal storage format of Fedora's Greenstone documents in this convenient manner, there was some disparity between how searching Greenstone 3 repositories worked compared to how searching worked in Fedora's case. A Greenstone document in Fedora is a digital object identified by its Fedora PID, while a document's section contents are presented as datastreams within the document digital object where each section is identified by its datastream ID. When performing a search using FedoraGSearch's web services, the PIDs of the digital objects wherein the search term occurred are returned. That is, the FedoraGSearch search operation does not return the section (datastream) IDs of the texts containing the search term, but rather the higher-level document IDs (PIDs). This has the disadvantage that clients will not be able to present the end-user with the exact section of a document that contains their search term, but rather the overall document containing the section where it is to be found. This is unlike Greenstone, where searches can be made to return documentNodes identifying sections in which the search terms are found.

While Greenstone provides search facilities on a collection basis, searching within a 'collection' in Fedora using Fedora Generic Search requires the FedoraGS3 component to filter the PIDs of the Greenstone digital objects returned for a search by the requested collection's name. Fedora Generic Search allows one to search on a PID prefix or limit search results to a PID prefix, but searching on a digital object's PID no longer works if the collection name is appended to the PID prefix. For example, to search for Greenstone documents in the Fedora repository within the gs2mgdemo collection would not work were we to search for PIDs that are prefixed by `greenstone:gsmgdemo(*)` or similar. However, Fedora Generic Search does allow searches to be performed on the recognised PID prefix of `greenstone`. This means that the FedoraGS3 component is forced to restrict the search results afterwards to those PIDs containing the necessary collection name.
Fedora's metadata search also exhibited some unexpected behaviour. Its allows one to search the DC title metadata field, for instance, and one can specify wild cards to match

regular expressions. The Browse service we provided for Fedora would allow users to browse titles in a collection according to their first letter. However, Fedora's metadata search would retrieve all titles containing *any* words that started with the requested letter, rather than returning what we expected: titles where the *first* word started with the letter. Of course, this problem was easily overcome by filtering the result-set that Fedora's metadata search returned to just those that started with the letter we want.

5.4. Conclusion and future work

Two of the project's objectives have clearly been met: a demo-client has been implemented for Greenstone 3 that also integrates access to a Fedora repository.

The main objective of designing and implementing a set of web services for Greenstone 3 that are both *general and useful enough* has at least been partially accomplished, in that web services have been provided that cover the operations available through the Message Router's `process()` method. Detailed evaluation of the web services would help to assess the extent to which this objective has been achieved. Evaluations performed so far in this project on the web services using the Java-client (which we built to make use of those same web services) can not be considered entirely objective. They nevertheless managed to provide some suggestions for improvement. The web services provided for Greenstone 3 were also considered alongside those offered by Fedora and Fedora Generic Search in terms of:

- how they dealt with optional parameters (method overloading versus accepting default values),
- the types of parameters and return values they made use of,
- the ease of invocation and ease of use of return values, and
- the breadth of digital library operations provided.

Though comparisons between Fedora, Fedora Generic Search and Greenstone 3's web services were indeed useful in identifying whether the choice of parameter types and return types was sensible, such a comparison does not give enough feedback on the usefulness of the web services themselves. Fedora and Greenstone are in some respects quite different digital library systems, hence there might not be a great amount of similarity between their functionality and, consequently, between their web services. Even so, this preliminary comparative evaluation did suggest some beneficial changes to the design of the web services, including reasons for exposing more of Greenstone's core functionality as web services than merely the operations pertaining to granting access to the repository. These were taken into account in the re-implementation of the web services. Some variations of overloaded methods were ultimately found to be unnecessary and were removed as a result.

5.4.1. Suggestions for further work in this area

Additional means of evaluating Greenstone 3's web services could include considering existing applications that make use of Greenstone's capabilities behind-the-scenes and try to rewrite those parts of their code that directly invoked Greenstone by replacing them

with calls to appropriate web services instead. If there are no web services to carry out a task, then it would indicate an omission in the set of web services provided.

One way to evaluate a different aspect of the web services would be to implement a very simple client application in another programming language, such as Perl, which also has web services support. The client merely needs to test the web service operations to make sure they work the same as they do in Java.

In this project, we looked at accessing Fedora and tying that back into Greenstone 3's Java-client. It would be interesting to know whether other web service-enabled digital library systems—like EPrints is set to become—can be incorporated in a similar manner as well. Digital libraries written in other programming languages that have their main functionality exposed as web services would enable our Java-based client application to build on their services. It would first require designing a proper format in which to store Greenstone collection and document data in there, and exporting Greenstone contents to this other digital library's format. Trying to include another digital library system into the Greenstone 3 Java-client would also provide a way of evaluating the completeness of the *DigitalServicesAPIA* interface and its helpfulness when it comes to incorporating access to another repository.

References

[1] "The Simple Digital Library Interoperability Protocol (SDLIP-Core)". http://dbpubs.stanford.edu:8091/~testbed/doc2/SDLIP/

[2] Bainbridge, D., Witten, I.H., Buchanan, G., McPherson, J., Jones, S. and Mahoui, A., "Greenstone: A platform for distributed digital library applications." in Proc Fifth European Conference on Research and Advance Technology for Digital Libraries (ECDL'01), LNCS 2163, Constantopoulos, P. and Solvberg, I.T., Eds. Darmstadt, Germany: Springer-Verlag Heidelberg, 2001, pp. 137-148.

[3] Bainbridge, D., Don, K.J., Buchanan, G.R., Witten, I.H., Jones, S.R., Jones, M. and Barr, M.I., "Dynamic digital library construction and configuration", in Proc Eighth European Conference on Research and Advanced Technology for Digital Libraries (ECDL'04), LNCS 3232, Heery, R. and Lyon, L., Eds. Bath, UK: Springer-Verlag, Berlin, 2004, pp. 1-13.

[4] Bainbridge, D., Kaun-Yu, K. and Witten, I.H., "Document level interoperability for collection creators", JCDL, 2006, pp.105-106.

[5] Beach, R., Tansley, R., Harnad, S., Packer, A.L., English, R., Lunau, C.D., Jokitalo, P. and Twiss-Brooks, A., "In Brief", D-Lib Magazine, vol. 6(10), 2000.

[6] Carr, L., "Eprints version 3: Repository Walkthrough", http://www.eprints.org/software/v3/EPrintsv3Presentation_small.pdf, 2007

[7] Deitel, H.M., Deitel, P.M., DuWaldt, B. and Trees, L.K., Web Services: A Technical Introduction (Deitel Developer Series). Prentice Hall PTR, 2002.

[8] Don, K., Buchanan, G. and Witten, I.H., "Greenstone3: A modular digital library", Department of Computer Science, University of Waikato, http://www.greenstone.org/docs/greenstone3/manual.pdf, 2006

[9] Englander, R., Java and SOAP. O' Reilly, 2002.

[10] Lagoze, C., Payette, S., Shin, E. and Wilper, C., "Fedora: An Architecture for Complex Objects and their Relationships", International Journal on Digital Libraries, vol. 6(2), pp. 124-138, 2006.

[11] Pedersen, G.S., "Fedora Generic Search Service", Technical University of Denmark, Fedora Project, http://defxws2006.cvt.dk/fedoragsearch/, 2006

[12] Phillips, S., Green, C., Maslov, A., Mikeal, A. and Leggett, J., "Manakin: A New Face for DSpace", D-Lib Magazine, vol. 13(11/12), 2007.

[13] Smith, M., Bass, M., McClellan, G., Tansley, R., Barton, M., Branschofsky, M., Stuve, D. and Walker, J.H., "DSpace: an open source dynamic digital repository", D-Lib Magazine, vol. 9(1), 2003.

[14] Tansley, R. and Harnad, S., "Eprints.org software for creating institutional and individual open archives", D-Lib Magazine, vol. 6(10), 2000.

[15] The Fedora Development Team, "Fedora Tutorial 1: Introduction to Fedora", http://www.fedora.info/download/2.2.1/userdocs/tutorials/tutorial1.pdf, 2005

[16] Witten, I.H., Bainbridge, D., Tansley, R., Huang, C. and Don, K.J., "StoneD. A Bridge between Greenstone and DSpace", D-Lib Magazine, vol. 11(9), 2005.

[17] Witten, I.H. and Bainbridge, D., "A retrospective look at Greenstone: lessons from the first decade", in Proceedings of the 2007 conference on Digital libraries. Vancouver, BC, Canada: ACM, 2007, pp. 147-156.

Appendix

Table A.1: The Greenstone 3 web services Access API. Listing of web service method definitions.

Methods for sending describe-type requests to the Greenstone's Messagerouter, Collections, ServiceRacks and Services ***(See Describe-type request messages, Greenstone 3 Developer's Manual, Section 3.4, pp.35-41)***	
String describe(); String describe(String lang, String subsetOption); String describeServiceRack(String toSC, String lang, String subsetOption); String describeCollection(String toColl, String lang, String subsetOption); String describeCollService(String toColl, String toService, String lang, String subsetOption); String describeService(String toService, String lang, String subsetOption);	
String getMessageRouterSubsetOptions(); String getCSSubsetOptions(); String getServiceSubsetOptions();	Returns the values accepted for the SubsetOption parameter of describe requests sent to the MessageRouter, Collections and ServiceRacks and Services (respectively).
Query requests for executing queries ***(See process-type messages: query-type services, Greenstone 3 Developer's Manual, Section 3.8.1, pp.45, 46)***	
String queryProcess(String toColl, String toService, String lang, String[] names, String[] values); String queryProcess(String to, String lang, String[] names, String[] values);	
String queryProcess(String to, String lang, String paramListElement);	This method can be used if the user has already prepared the (proper) XML specifying the list of parameters required by the Query service being invoked
String queryProcess(String toColl, String toService, String lang,	The Hashmap parameter requires mappings from query

HashMap nameToValsMap); String queryProcess(String to, String lang, HashMap nameToValsMap);	fieldnames to field values. The fieldnames and fixed field values must be those recognised by Greenstone 3.
String simplerFieldNameQueryProcess(String collection, String service, String lang, HashMap nameValParamsMap);	This method's Hashmap parameter of query fieldnames and values will accept the user-friendly *fieldnames* returned by the getFieldNameMappings() method as well as the abbreviations used by Greenstone 3. Next to that, this method allows the user to provide *field values* like "all fields", "text", "titles", "subjects", "organisations" instead of Greenstone's accepted ZZ, TX, DL, DS, DO respectively. The user can provide "on" or "off" for such fields as casefolding and stemming, instead of the expected "1" or "0".
static HashMap getFieldNameMappings();	Hashmap of mappings from user-intelligible query fieldnames to the argument abbreviations Greenstone 3 uses for them. Users who wish to use the two regular queryProcess methods that take HashMap parameters, may call this method to find out what the query fieldnames are that Greenstone 3 accepts in order to pass the correct fieldname abbreviation.
Retrieve request messages for Content, Structure and Metadata retrieval for Documents and Classifiers ***(See process-type messages: retrieve-type services, Greenstone 3 Developer's Manual, Section 3.8.3, pp.47-49)***	
DocumentContentRetrieve requests	
String retrieveDocContent(String toColl, String lang, String[] docNodeIDs); String retrieveDocContent(String toColl, String lang, String docNodeListElement);	If the user has already prepared the proper XML specifying the list of DocumentNodes for which they want to retrieve the content, then they can use this method.
DocumentStructureRetrieve requests	

String retrieveEntireDocStructure(String toColl, String lang, String[] docNodeIDs); String retrieveDocStructure(String toColl, String lang, String[] docNodeIDs, String[] structure, String[] info); String retrieveEntireDocStructure(String toColl, String lang, String docNodeListElement); String retrieveDocStructure(String toColl, String lang, String docNodeListElement, String[] structure, String[] info); String documentStructureOptions(); String browseStructureOptions(); String documentStructureInfo();	If the user has already prepared the proper XML specifying the list of DocumentNodes or ClassifierNodes for which they want to retrieve the structure, then they can use these two methods. Returns the values Greenstone accepts for the parameters structure and info when sending a DocumentStructureRetrieve request. For structure, these can be ancestors, parent, siblings, children, descendants, entire. For info, this can be numSiblings, siblingPosition, numChildren.
DocumentMetadataRetrieve requests	
String retrieveAllDocMetadata(String toColl, String lang, String[] docNodeIDs); String retrieveDocMetadata(String toColl, String lang, String[] docNodeIDs, String metaName); String retrieveDocMetadata(String toColl, String lang, String[] docNodeIDs, String[] metaNames); String retrieveAllDocMetadata(String toColl, String lang, String docNodeListElement); String retrieveDocMetadata(String toColl, String lang, String docNodeListElement, String metaName); String retrieveDocMetadata(String toColl, String lang, String docNodeListElement, String[] metaNames);	If the user has already prepared the proper XML specifying the list of DocumentNodes for which they want to retrieve the metadata, then they can use these three methods.
ClassifierBrowseMetadataRetrieve - Metadataretrieve for browsing classification hierarchies	
String browseMetadataRetrieveAll(String toColl, String categoryName, String lang, String[] nodeIDs);	

String browseMetadataRetrieve(String toColl, String categoryName, String lang, String[] nodeIDs, String metaName); String browseMetadataRetrieveAll(String toColl, String categoryName, String lang, String nodeListElement); String browseMetadataRetrieve(String toColl, String categoryName, String lang, String nodeListElement, String metaName);	If the user has already prepared the proper XML specifying the list of ClassifierNodes for which they want to retrieve the metadata, then they can use these two methods.
The core process method (which can process all valid Greenstone 3 messages)	
String process(String toColl, String toService, String lang, String id, String paramList);	A direct mapping of the process method of Greenstone 3's MessageRouter.