O'REILLY®

eXist

A NOSQL DOCUMENT DATABASE AND APPLICATION PLATFORM

Erik Siegel & Adam Retter

O'REILLY®

eXist

Get a head start with eXist, the open source NoSQL database and application development platform built entirely around XML technologies. With this hands-on guide, you'll learn eXist from the ground up, from using this feature-rich database to work with millions of documents to building complex web applications that take advantage of eXist's many extensions.

If you're familiar with XML—as a student, professor, publisher, or developer—you'll find that eXist is ideal for all kinds of documents. This book shows you how to store, query, and search documents with XQuery and other XML technologies, and how to construct applications on top of the database with tools such as eXide and eXist's built-in development environment.

- Manage both data-oriented and text-oriented markup documents securely
- Build a sample application that analyzes and searches Shakespeare's plays
- Go inside the architecture and learn how eXist processes documents
- Learn how to work with eXist's internal development environment
- Choose among various indexes, including a full-text index based on Apache Lucene
- Dive into eXist's APIs for integrating or interacting with the database
- Extend eXist by building your own Triggers, Scheduled Tasks, and XQuery extension modules

"This book tells you everything you need to know to implement eXist, all the way from writing your first queries to building sophisticated web applications."

—**Priscilla Walmsley**
XML consultant and author of *XQuery*

"Erik Siegel and Adam Retter have written a technical tour de force about the database in their eponymous book eXist, one that's both eminently readable while still digging deep into the inner workings of the database and how to use it."

—**Kurt Cagle**
Principal Evangelist at Avalon Consulting

Erik Siegel is a content engineer and XML specialist who runs Xatapult consultancy in The Netherlands. He specializes in content design and conversion, XML schemas and transformations, and eXist and XProc applications.

Adam Retter, Director of Evolved Binary in the UK and a cofounder of eXist Solutions in Germany, has been a member of the eXist Core Development Team since 2005. He is also a member of the XML Guild and an Invited Expert to the W3C XML Query Working Group.

XML/DATABASES

US $44.99 CAN $47.99
ISBN: 978-1-449-33710-0
9 781449 337100 54499

Twitter: @oreillymedia
facebook.com/oreilly

eXist

Erik Siegel and Adam Retter

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo
O'REILLY®

eXist
by Erik Siegel and Adam Retter

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editors: Simon St. Laurent and Allyson MacDonald
Production Editor: Matthew Hacker
Copyeditor: Rachel Monaghan
Proofreader: Rachel Head
Indexer: Lucie Haskins
Interior Designer: David Futato
Cover Designer: Ellie Volckhausen
Illustrator: Rebecca Demarest

December 2014: First Edition

Revision History for the First Edition
2014-12-04: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449337100* for release details.

978-1-449-33710-0

[LSI]

Table of Contents

Preface

Welcome

Welcome, dear reader, to our book on eXist. Whether you have purchased, begged, borrowed, or stolen this book, we hope that you find its contents of great use when applied to solving your information management problems.

While it's true that eXist has been around for some years now—in fact, for longer than many of the now popular NoSQL platforms—eXist has continued to innovate and evolve. eXist, while stable and widely used for many years, has now hit a milestone in its history where it can be considered "battle-worn"—a veteran, if you like (or as we like to say in software engineering, "mature"). We have considered writing a book on eXist for the past few years, but we now know that the time is right to share our knowledge with the world. Welcome eXist 2.0.

Who Is This Book For?

Perhaps we should first answer this question with another question: *Who is eXist for?*

eXist aims to meet the requirements of a wide user base, and therefore is probably the most feature-rich product in its class. eXist has been engineered over the years to meet the needs of users ranging from humanities students and professors undertaking interesting linguistic projects, to large international publishers working with millions of documents, to developers wishing to rapidly create document- and data-driven web applications, and most cases in between.

This book aims to meet the needs of a wide audience: from tinkerers, students, professors, and information managers right up to software engineers. This book assumes that you wish to learn and use eXist; if not, you may have bought the wrong book! No familiarity with eXist is assumed; we start with the basics and progresses to more complicated topics. This book does not set out to teach XML, XPath, XQuery, XSLT, XForms, or any of the other XML technologies. While of course you may gain an understanding of them from this book, there are other books and online resources available that focus on these topics as their *raison d'être*. We assume that you have a working knowledge of, or access to learning resources for, XML technologies.

As always, beginners should start at the beginning, while those who already have some experience with eXist may find new insights in Chapters 4 to 6 onward. We hope you will find the book an excellent reference resource.

Should you be looking for books on XML technologies, in our experience and from the feedback of colleagues and beginners we have met, it is a good idea to have a copy of *XQuery* by Priscilla Walmsley (O'Reilly) at hand, as XQuery is the predominant language used for working with eXist. For further useful resources, see "Additional Resources" on page 16.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, file- and pathnames, database collections, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, module names, data types, environment variables, statements, and keywords. Also used for commands and command-line output, database user and group names, and permission modes.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

$EXIST_HOME
: While *$EXIST_HOME* typically follows the Unix-like syntactical expression of an environment variable, it is used throughout the book to refer to the location where you have installed eXist, whether that be on a Windows/Linux/Mac or any other type of system. The corresponding expression for referencing the equivalent environment variable on Windows platforms would be *%EXIST_HOME%*.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

XQuery Filename Conventions

The XQuery specification as published by the W3C does not define a particular filename extension for XQuery files. The specification, however, does define two different types of XQuery module:

XQuery main module
: A *main module* is defined as having a *query body*. Simply put, this means that an XQuery processor can directly evaluate the XQuery code in this file.

XQuery library module
: A *library module* does not have a query body and must start with a *module declaration*. Again, simply put, this means that an XQuery processor cannot directly evaluate a library module; rather, the library module must be directly or indirectly imported into a main module.

As a result, there has been a proliferation of different filename extensions used for XQuery files, including *.xq*, *.xql*, *.xqm*, *.xqy*, *.xql*, *.xqws*, and *.xquery*. Each XQuery implementation vendor, and even individual XQuery developers, seem to have their own ideas about XQuery file naming. Some projects differentiate between main and library modules by using two different file extensions, but which two is entirely inconsistent across projects. Other projects opt to use a single file extension and apply it to both main and library modules. This proliferation of different file extensions can be disorienting and leads to confusion when you're approaching an existing code base.

eXist recognizes and supports XQuery files with any of the aforementioned file extensions, and will load and store them correctly into its database as XQuery. However, we believe that such an accumulation of different file extensions for what is effectively one or two (main and library) types of file is ridiculous and raises the barrier to truly reusable and portable XQuery code within projects, between projects, and across XQuery implementations.

This book takes the strong opinion that the following XQuery file extension convention should be used by at least all users of eXist, if not all XQuery developers:

.xq
: The *.xq* filename extension is to be used for all main modules.

.xqm
: The *.xqm* extension is to be used for all library modules. The *m* suffix in the file extension indicates that the XQuery module starts with a module declaration and is therefore a library module.

This convention is justified by the following points:

- The ability to differentiate between main modules and library modules at the file level proves very useful within a large project. Especially if you are new to the project, you can easily and quickly locate the main entry points of the application.
- This is not yet another new convention (standard); this is already the convention in at least one other project outside of eXist.
- It is backward compatible with various approaches that have been adopted by eXist community members in the past.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*eXist* by Erik Siegel and Adam Retter (O'Reilly). Copyright 2015 Erik Siegel and Adam Retter, 978-1-449-33710-0."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Accompanying Source Code

Many of the code examples provided in the book and example programs that are discussed in the book are publicly available from GitHub at *https://github.com/eXist-book*, where we currently provide two repositories:

https://github.com/eXist-book/book-code
: This encompasses all of the code that accompanies the book (i.e., XQuery, XSL-FO, XSLT, XForms, XML, Java, and Python), except for the examples discussed in Chapter 3.

  For convenience, build scripts are included so that the majority of examples can be compiled into an EXPath Package file (see "Packaging" on page 227) that can be easily deployed into eXist, and the Java projects can be compiled into JAR files for use with eXist or from the command line.

https://github.com/eXist-book/using-exist-101
: This is provided as a reference for the tutorials set out in Chapter 3. It is deliberately kept separate from the other code examples, as we felt that you would benefit more from following the tutorials and entering the code manually while considering each line of code that you are writing.

  This repository is structured as an eXist backup. To restore the backup, see "Backup and Restore" on page 396.

Getting the Source Code

With either of our two GitHub repositories, to get a copy of the source code you need to ideally have Git installed. If you do not wish to install Git, it is also possible from the GitHub repositories to download a ZIP or compressed TAR file of the source code. However, using Git is recommended, as it will allow you to easily update the source code in the future, should we make any corrections or additions.

Assuming that you have Git installed (if you are on a Windows platform, we will assume that you are using Git Shell), from your Unix/Linux/Mac terminal (or your Windows Git Shell), you can run the following to *clone* (make a copy of) our repositories:

```
$ mkdir exist-book
$ cd exist-book
$ git clone https://github.com/eXist-book/book-code
$ git clone https://github.com/eXist-book/using-exist-101
```

You now have a clone of each repository. In the future, should you wish to pull in any updates we have made, you can simply run:

```
$ cd exist-book/book-code
$ git pull
$ cd ../using-exist-101
$ git pull
```

Building and Deploying

Now let's look at how you build and deploy the code from the *book-code* repository.

The *book-code* repository contains the following top-level folders:

build-parent
: This folder contains the build configuration that is inherited by each project.

build-parent-java
: This folder contains the build configuration that is inherited by each of the Java projects.

chapters
: This folder contains subfolders for each chapter of the book where example code is provided.

xml-examples-xar
: This folder contains the build configuration for building an EXPath package.

Building everything

We use the Apache Maven build tool for compiling all of the projects that accompany the book. Therefore, to make the most of the example code that goes along with the book, you will also need to download and install Maven. Maven, like eXist, requires Java; if you do not already have Java installed you can download either Java 6 or 7 from *http://java.oracle.com*. Each *pom.xml* file that you see in the code is a Maven project file that describes how to build the code and resolves any dependencies that are required.

If you wish to build all of the code projects that accompany the book in one step, you can simply run the following commands from your terminal (or Git Shell on Windows):

```
$ cd book-code
$ mvn package
```

Building the EXPath package

If you wish to build just the EXPath package of the example XQuery, XSLT, XForms, and XML code that accompanies the book, you can simply enter the *xml-examples-*

xar subfolder and run `mvn package`. To achieve this, we have used the excellent EXPath package Maven plug-in written by Claudius Teodorescu, which allows us to easily create a XAR file from a manifest (see the file *xml-examples-xar/expath-pkg.assembly.xml*) that describes the EXPath package.

The result of the Maven build process is the file *exist-book-1.0.xar* in the *target* subfolder of *xml-examples-xar*. You can then deploy the package by either copying it to *$EXIST_HOME/autodeploy*, or using the dashboard app as follows:

1. Open up the eXist dashboard in your web browser, log in as `admin`, and click on the Package Manager tile.
2. Click on the upload application icon (in the top left of the screen; it looks like a stack of disks).
3. Browse to and select the *exist-book-1.0.xar* file and press the Submit button.

After installation, the sample code is available as another tile in the dashboard. It runs as a simple application that allows you quick access to running the examples.

See "The Dashboard" on page 24 and "Packaging" on page 227 for further information on working with the dashboard and EXPath packages.

Compiling the Java examples

The Java examples that accompany the book will also be built if you build everything, and the resultant artifacts will be placed into the *target* subfolders of each project. Each Java project example is discussed in detail in the relevant chapter later in the book. You can also compile the Java projects individually by running `mvn package` in the folder of each Java project. For example, if you wanted to build just the REST Server client examples, you would run:

```
$ cd book-code/chapters/integration/restserver-client
$ mvn package
```

Each Java example is designed to both educate and potentially serve as a skeleton for your own Java projects. By simply changing the `groupId` and `artifactId` of the project's *pom.xml* file and including any additional required dependencies, you have a very quick mechanism to start building your own projects.

It is also worth mentioning that a ZIP or fat JAR file assembly is also created for many of the Java project examples, and this can be found in the appropriate *target* subfolder. A fat JAR file assembly is simply a JAR file that also contains all of the dependencies of the project, to allow you to have a single file artifact. So, for example, when you are compiling the *restserver-client* examples, the following assemblies are created:

- *restserver-client-query/target/restserver-client-query-1.0-example.jar*

- *restserver-client-remove/target/restserver-client-remove-1.0-example.jar*
- *restserver-client-retrieve/target/restserver-client-retrieve-1.0-example.jar*
- *restserver-client-store/target/restserver-client-store-1.0-example.jar*

Safari® Books Online

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/eXist*.

To comment or ask technical questions about this book, send email to: *bookquestions@oreilly.com*

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

The authors would like to especially thank Dan McCreary, Chris Wallace, and Dannes Wessels for reviewing their work.

In addition, they would like to offer their thanks to Ron Van den Branden, Martin Holmes, Casey Jordan, Kurt Cagle, Paul Kelly, Tobi Krebs, Brois Lehečka, Wolfgang Meier, Chris Misztur, Dave Pawson, Jens Østergaard Petersen, Phill Ramey, Dmitriy Shabanov, Luis Tavera, Claudius Teodorescu, Chris Tomlinson, Joern Turner, David Voňka, Priscilla Walmsley, Michael Westbay, Joe Wicentowski, and Lars Windauer for their support and feedback.

CHAPTER 1
Introduction

What Is eXist?

As it turns out, this is quite a difficult question to answer. The problem lies in the wide audience that eXist serves. eXist is many things to many people, and thus there is no single succinct answer.

eXist is an open source piece of software written in Java that is freely available in both source code and binary form. eXist has always been made available under the Lesser GNU Public License (LGPL), version 2.1. While eXist makes use of many other open source libraries itself, all of these are compatible with the LGPL, and eXist eschews the GPL license in favor of freedom of choice for its users.

eXist was conceived as a native XML database. As a database, its unit of atomicity is the document, so we could very easily brand it a *NoSQL document database*. However, to do so would be to do an injustice to the software, and worse, to all of those who have contributed to making eXist much more than just a NoSQL database over the years.

Unlike most NoSQL databases, which each have their own proprietary database query language, eXist makes use of a standardized query language developed by the W3C: XML Query Language (XQuery). With a standard query language, you have the ability to write code that can be used not just on eXist, but on any platform or processor that supports XQuery. Some of the benefits of XQuery are that it is:

Synergistic
: XQuery was carefully designed and evolved over a six-year period by an open working group with many contributors, meaning that many real industry use cases were considered during its construction. XQuery has been influenced by

several previous languages and concepts, such as Perl, Lisp, Haskell, SQL, and many more.

Easy to use
: XQuery was designed to be simple to use and debug, meaning that nonprogrammers (given an understanding of their documents) should be able to easily construct queries. Many eXist users work in the humanities and have no formal computer science background, but are quite comfortable writing complex XQueries to query their documents.

Easy to optimize
: The XQuery specification does not detail how an implementation should perform query processing, and its developers have given great thought to ensuring that any implementation can optimize processing of query operations. Likewise, as a moderate user of XQuery, you can often easily understand why a particular query is slow and what you may do to improve it.

Easy to index
: Join operations in XQuery (e.g., predicates and `where` clauses) lend themselves well to index optimization, which eXist exploits to speed up XQuery execution. eXist provides multiple indexing schemes that the user may configure.

Turing complete
: XQuery is more than just a query language: it is in a class of languages known as *Turing complete*, which means that it is a complete programming language and any program can be expressed in it. XQuery is also a functional programming language, as opposed to a procedural one, meaning that it is generally easier to construct programs that you can easily understand and that ultimately contain fewer bugs. In eXist you can build entire applications in just XQuery!

Query first
: While XQuery is a programming language, it is designed primarily as a query language. Therefore, it is much easier to extract just a few elements from large data collections with XQuery than, say, with XSLT.

More than you realize!
: While XQuery is easy to get started with, its functional nature can make it tricky to work with if you have only procedural programming experience. XQuery can be incredibly elegant, and we are frequently surprised at how very complex problems may be solved quite simply in XQuery when considered in a different light.

We should also perhaps mention that eXist is not just for XML documents. You can, in fact, store *any* file into the database, and eXist can do some very clever things with content extraction and metadata with non-XML documents to help you query and manage those binary formats too.

We could stop there and focus the rest of the book on the database, but you would really miss out on the good stuff. eXist is also a web server: you can make web requests directly to the database to store, retrieve, or update XML documents. eXist achieves this by providing an HTTP REST API that describes the database. It also provides a WebDAV interface so that your users can easily drag and drop documents from their desktops into the database, or open a document for editing.

But wait, there's more! As eXist evolved over the years it became clear that being able to store, retrieve, and edit documents via the Web was neat, but also being able to store XQuery into the database and execute it via a web request from your web browser meant you could easily construct very powerful web applications directly on top of the database. eXist of course continued to evolve here, providing new features for forms, web application packaging, improved security, SQL queries, SSL, and support for producing and consuming JSON and HTML 5, among other offerings.

So, in summary, what is eXist?

- A NoSQL document database for XML and binary (including text)
- A web server for consuming and serving documents
- A document search engine
- A web application platform
- A document creation and capture platform (XForms)
- A data mashup and integration platform
- An embeddable set of libraries for use in your own applications
- And much, much more

eXist Compared to Other Database Systems

Let's take a moment to discuss some of the main differences between eXist and other SQL and NoSQL database systems. eXist is:

Document oriented

Unlike traditional RDBMSs (Relational Database Management Systems) such as Oracle, MySQL, and SQL Server, which are table oriented, eXist is a NoSQL document-oriented database.

Many other NoSQL document databases (including MongoDB and Apache Cassandra) store JSON documents, whereas eXist stores XML documents. One of the key advantages of XML over JSON is the ability to handle complex document structures using *mixed content*. JSON easily handles data-oriented documents, while XML easily handles both data-oriented markup and text-oriented documents. Another key advantage of XML over JSON is that you can adopt *namespaces* to cleanly model different business domains.

Schemaless
: RDBMSs and even several NoSQL databases require you to define your data schema before you can start storing your data. eXist is entirely flexible, and allows you to store your documents without specifying any schema whatsoever. It is ideal for business problems that have high-variability data and also helps developers rapidly prototype and evolve applications. However, *schemaless* should not be confused with providing *validation of documents*. Should you wish, you can also define a schema in eXist and have eXist enforce that only documents meeting your schema requirements are stored or updated.

Portable queries
: RDBMSs typically use a standardized SQL query language; however, in practice, apart from the most basic queries it can be hard to run the same SQL queries across different RDBMS database products. Likewise, most NoSQL systems have their own proprietary query languages, which are entirely product-specific. eXist takes a very different approach and provides XQuery and XSLT, which are W3C standardized query and transformation languages, meaning that with very little effort you can execute your eXist queries on any other product that provides an XQuery and/or XSLT processor.

Structured search
: Like many database systems, eXist allows you to define different indexes for your searches. However, combined with the ability to search based on the document structure, this makes eXist search results more precise than those of almost any other database when dealing with structured documents such as TEI, DocBook, and DITA. eXist will consistently have better search metrics (precision and recall) than search systems that ignore document structure. If findability is high on your list of desired attributes, then eXist is a great choice.

Forms
: Oracle provides Oracle forms for use with its RDBMS. We are not aware of any NoSQL databases that provide form support for constructing end-user interfaces that can feed directly into the database. eXist supports XForms (another W3C standard), which allows you to easily capture user input into XML documents in the database. Some organizations find that eXist is ideal not just for managing data collection with forms, but also for entire backend workflows around the content publishing process.

Application development
: Like some RDBMSs and NoSQL databases, eXist is embeddable into your own applications. However, when you are using eXist as a server, it really becomes an application platform and offers more than almost any other database system.

When running eXist as a server, you can develop entire applications in eXist's high-level query languages (XQuery and XSLT) without necessarily having to be a computer scientist or programmer.

Transaction management
: Most RDBMSs and many NoSQL databases allow you to control your transactions from within your database queries or API calls. Unfortunately, eXist does not currently support database-level transaction control. eXist does have transactions internally and uses a database journal to ensure the durability and consistency of your data, but these are not exposed to the user. Transaction control is high on the list of desirable features for eXist, and some options have already been explored for the future.

Horizontal scalability and replication
: A feature of many RDBMSs and NoSQL database systems is the ability to cluster database nodes to increase database scalability and capacity. eXist is currently mostly deployed on single servers. As of this writing, it has no support for automatic sharding of data. eXist has recently gained support for replication in version 2.1 through JMS (Java Message Service) message queueing. The replication feature allows you to have a *master/slave* database, which is highly available for reads that may occur from any node, but it is still currently recommended to send writes to the master node. For further details of the emerging replication support in eXist, see *https://github.com/eXist-db/messaging-replication/wiki*.

History

Once upon a time, around the turn of the 21st century, there was a researcher named Wolfgang Meier working at the Technical University of Darmstadt. He was in need of a system to analyze and query XML data, and since there was nothing around that satisfied his needs, he decided to write something himself: eXist.

Starting out in C++, Meier quickly turned to Java, and by the beginning of 2001, a first version was available. It was based on a relational database backend and, compared to where we are now, very primitive. The functionality was basic and it was slow on indexing, but yes, it already had some XPath on board. Immediately, some dictionary research projects started using it.

The next stage was replacing the relational backend with native XML storage. While this was happening, more and more people started using eXist, and around 2004 the first commercial projects arrived. The development of eXist has since then mostly been financed by its users, who needed new functionality and were willing to pay for it.

Implementing XQuery met some resistance. At that stage, eXist was still mostly an XML database only. Why would you need something like XQuery if you already have

XPath? Luckily (for us), a professor of literature really needed XQuery support and paid for its implementation. It was embedded in the product by 2005.

During 2005 eXist was going so well that Meier was able to quit his university job and concentrate on eXist projects only. By that time some other programmers had come on board, and they constitute what we now know as the original "core programmer team." In alphabetical order, they were Pierrick Brihaye, Leif-Jöran Olsson, Adam Retter, and Dannes Wessels.

Up to 2006 the version number was kept to v0.xx, but in 2006 a real v1.0 was released!

By this time, having previously only communicated via the Internet, the core programmer team met live for the first time in 2007 in Versailles. One of the first things they did was to check eXist against the official XQuery test suite, which subsequently resulted in the current 99%+ compliance score.

The product kept evolving. A major improvement was replacing the existing scheme for node identifiers with a much better one. As a result of that, limitations on XML size and structure disappeared. Stability and transaction management were improved and the Lucene full-text search engine added. From the original research/retrieval tool, eXist evolved into something we really could call a *native XML database*.

The XRX (XForms, REST, XQuery) paradigm popped up as a way to create fully XML-driven applications. eXist was among the first engines that made this possible. It turned, slowly but surely, into a full-blown *application platform*.

With version 1.4 of eXist released in 2009, suddenly many more organizations were using eXist in their production systems day to day. More development effort went into stabilizing, fixing bugs, and improving reliability. With this, the first "settled application" problems arrived: it became harder and harder to change anything without breaking backward compatibility. Consider, for instance, eXist's XQuery update support: an implementation of a draft version of the standard for writing XQuery statements to update XML. Switching to the final standardized version is virtually impossible because it would break backward compatibility and existing applications would stop working.

However, the development team did not stop working, and gradually the 2.0 version as we know it came to life. Release candidates were made available throughout 2012, containing a large number of major changes and additions to the previous versions:

- Behind the scenes, the XQuery engine and optimizer were improved.
- Support for (large parts of) XQuery 3.0 was added.
- The way the indexes work was redesigned to reduce lock contention, offer modularity, and improve performance.

- Security was reorganized and now works not only a lot faster, but also in a way most developers are comfortable with (i.e., similar to Unix-like systems).
- The repository manager was added, opening the way to a more modular eXist.
- RESTXQ, a standard for coupling function invocations to URLs, was added.
- And, of course, numerous other small improvements were made.

The final version 2.0, released in February 2013, was a massive leap forward from 1.4, representing the culmination of more than three years' worth of sustained development effort. As such, it was not completely without backward compatibility problems. For instance, existing XQuery applications will have to do something about their security settings before they can run on the new version. However, that's not too hard and is well worthwhile.

Version 2.1 was released shortly after, in July 2013, and consisted mostly of bug fixes and a new version of eXide. In February 2014, a release candidate of eXist 2.2 was made available, which—along with the usual bug fixes—included a completely new range index based on Lucene that offered much improved query times.

It is expected that eXist will keep evolving. The plans are to move more and more toward a core product with separate modules, enabling adding à la carte functionality as needed.

Competitors

Now, obviously we are passionate about eXist; otherwise, you would not be reading a book we have written on the subject. More importantly, though, we are passionate about open source, and even more so we are concerned with quality software and using the right tool for the job. Like any other product, eXist has both strengths and weaknesses, and it would be somewhat misleading if we were not to share the whole story with you. Pointing out the weaknesses of a software product for which you have bought a book may not help us sell more books, but we do hope it will help you make informed decisions.

As eXist has such a wide scope, it is impossible to compare it directly to other products; so, we compare it instead against other native XML databases that also couple web server and application platform capabilities.

eXist's competitors can be split into two categories: those that are open source and freely available, and the closed source, commercial offerings. By no means is what follows a complete list, but it contains the offerings that we believe are popular and frequently encounter when talking to others.

A further independent comparison is available in the XML database article on Wikipedia.

Open Source Competitors

Let's begin with the open source eXist competitors.

BaseX

BaseX is a lightweight native XML database with some application server facilities written in Java. The project was started in 2005 by Christian Grün at the University of Konstanz, and BaseX was released as open source in 2007. BaseX promotes ease of use and provides an easy-to-use GUI frontend also written in Java.

Compared to eXist, BaseX adheres more closely to the W3C XQuery specifications, achieving 99.9% compliance with the W3C XQuery 1.0 specification (eXist has 99.4%) and implementing the specifications for XQuery Update 1.0 and XQuery Full-Text. eXist has an older draft implementation of XQuery Update and its own proprietary full-text search. eXist, however, has been available for significantly longer, and thus benefits from many more features, such as XSLT.

BaseX is released under the more liberal BSD license. Commercial support is available for BaseX from BaseX GmbH, which was founded to support commercial applications of BaseX.

Sedna

Sedna is a lightweight native XML database without application server capability, written in C and C++. The origins of Sedna are not well documented, but it appears to have started around 2003 as a project of the Institute for System Programming at the Russian Academy of Sciences. Sedna seems to focus on providing core database services and little more. While it has no REST Server of its own, it can be configured to work as a module within the Apache HTTP Server.

Compared to eXist, Sedna supports more APIs for different programming languages directly; eXist mostly assumes that developers will use its REST or RPC APIs, and leaves language APIs to third-party providers. Sedna reports 98.8% compliance with the W3C XQuery 1.0 specification; as mentioned previously, eXist has 99.4% compliance. Sedna, like eXist, implements its own proprietary full-text search, and a draft version of XQuery Update.

Sedna is released under the Apache 2.0 license. There does not appear to be a commercial support offering for Sedna.

Closed Source, Commercial Competitors

Now we'll take a look at eXist competitors that are closed source and commercially available.

28.io

28.io is a PaaS (Platform as a Service) for the Zorba open source XQuery processor. 28.io integrates Zorba with MongoDB, supporting the storage and indexing of XML into MongoDB as its main datastore. The query optimizer leverages the full capabilities of MQL (Mongo Query Language), enabling developers to leverage the expressiveness and productivity of XQuery atop a highly scalable store.

In comparison with eXist, 28.io focuses on the cloud and manages its database entirely using XQuery, whereas eXist provides a number of Admin GUI tools and additional APIs. 28.io, much like eXist, provides an application server platform enabling you to build entire apps using XQuery. 28.io (through Zorba) has similar compliance to the W3C XQuery 1.0 specification as eXist, but also supports XQuery Update, XQuery Full-Text, and XQuery Scripting. 28.io's main advantage over eXist is its cloud scaling; eXist's main advantage is XSLT, XRX, and XForms support.

28.io is developed by 28msec. 28msec is based in Zurich, Switzerland, and has strong research links with ETH Zürich.

MarkLogic Server

MarkLogic Server is a standalone native XML database server, written in C++, that provides XQuery and XSLT query and transform capabilities. MarkLogic Server also has the capability to cluster nodes to scale horizontally, with the additional capability to pass large batch processing jobs off to Hadoop. MarkLogic distances itself from the technical marketing of XML and XQuery and instead identifies itself as a NoSQL database solution for the enterprise.

Compared to eXist, MarkLogic markets itself as being able to handle petabytes of XML data. eXist can currently scale to hundreds of gigabytes, but this is very much dependent on the dataset and queries made. MarkLogic lacks a document-representative REST API, but does provide a REST API for application development. MarkLogic's main advantage over eXist is scaling to huge datasets, while eXist's advantage is its fast innovation and rich feature set. Both support XSLT, but MarkLogic does not support XQuery Update; rather, it provides its own proprietary functions.

MarkLogic Server is developed by MarkLogic Corporation, based in San Carlos, California.

Who Is Using eXist, and for What?

The problem with giving something away for free with *no questions asked* is that you can never quite be sure:

- How many people are using it

- Who the people using it are
- What it is being used for

From the support channels available to eXist users, and as a member of the community, you can see that eXist is used by many people for many different purposes, but their end goals and projects are not always disclosed or clear.

Here we have pulled together a few descriptions of various projects using eXist from the developers of those projects themselves:

> The Tibetan Buddhist Resource Center (TBRC) holds the world's single largest collection of Tibetan texts—nearly 10 million scanned pages and over 11,000 Unicode Tibetan texts. TBRC.org provides online access to over 4,000 users via an Ajax client written in Google Web Toolkit as a front-end to the eXist-db. TBRC has used eXist-db since 2004 to store the catalog for the texts in the library as well as a knowledge-base of persons and places that provide a cultural context for Tibetan literature. The integration of eXist-db with the Lucene full-text indexing has created a powerful framework with which TBRC.org is able to provide searchable access to the library via comprehensive tables of contents in Tibetan and a large collection of texts that have been input in Unicode in centers around the world. Our production system currently runs eXist-db 2.1.
>
> —Chris Tomlinson,
> Senior Technical Staff Member,
> Tibetan Buddhist Resource Center,
> Cambridge, Massachusetts

> ScoutDragon initially started as a baseball research project by a group of baseball enthusiasts including writers, agents, scouts, fans, fantasy owners, and even former players. This group realized a need for original English content, data, and research on baseball players in Asia.
>
> All data for multiple sports covering multiple sporting leagues is stored in XML documents within eXist in a schema derived from IPTC's SportsML, most extensions having to do with providing multi-lingual support of players so that information may be displayed in English, Japanese, Korean, and/or Chinese. XQuery has proven to be a fantastic language for not just transforming the vast quantities of data to web pages, but also for data analysis and the generation of sabermetrics-based statistics.
>
> —Michael Westbay,
> Lead Programmer/System Administrator,
> ScoutDragon.com,
> Japan

> Semanta's core business is metadata in business intelligence. Part of our concern is parsing metadata from reporting platforms. Many of these reporting platforms supply their metadata in large XML chunks, which we then need to further process efficiently. A typical example is our IBM Cognos connector, where we use eXist heavily to extract details of report structures and data sources. Originally we thought we would only use eXist for prototyping, but ultimately, we have used embedded eXist in the production

system; re-writing the connector without eXist's XQuery turned out to be just too complicated!

—David Voňka,
Programmer,
Semanta,
Czech Republic

The Centre for Document Studies and Scholarly Editing of the Royal Academy of Dutch Language and Literature (Ghent, Belgium) develops rich scholarly collections of textual data, and publishes them as digital text editions and language corpora. From the start, we have fully embraced open standards and publication technologies. At first, we started out with the Cocoon XML publication framework, which back then nicely integrated with eXist (or the other way round) for efficient querying of XML content. Since the introduction of eXist's MVC framework, we have extended our use of eXist as a full application server, not only for querying the indexed data, but also driving the entire application and presentation logic.

The texts we're querying (or rather, processing) with eXist are mostly document-centered XML documents that are conformant to the schemas developed by the Text Encoding Initiative (TEI). Depending on the specific edition project, they are enriched with metadata such as named entities, editorial annotations, and sometimes highly specific textual documentation (such as critical apparatuses documenting variation among text versions). Though our texts are mostly in Dutch, we try to connect and contribute to methodological good practice emerging in the interesting field that is Digital Humanities. Some of our exemplar projects include a collection of letters in relation to the Belgian literary journal *Van Nu en Straks*; a digital edition comparing 20 versions of *De trein der traagheid*, a novel by the Belgian novelist Johan Daisne; and a digital edition of the first Dutch dialect survey in the Flemish region by Pieter Willems (developed between 1885–1890).

—Ron Van den Branden,
Centre for Scholarly Editing and Document Studies of the Royal Academy of Dutch Language and Literature,
Ghent, Belgium

At the Cluster of Excellence "Asia and Europe in a Global Context," we use eXist-db to store our collections of MODS (bibliographical) and VRA (image metadata) records. We have developed two open source applications for this, Tamboti and Ziziphus, where our records can be searched and edited. Both applications are built entirely in XML technologies (XQuery and XForms) using eXist-db and make use of LDAP integration and detailed user rights management.

—Heidelberg Research Architecture,
Cluster of Excellence "Asia and Europe in a Global Context,"
The University of Heidelberg,
Heidelberg, Germany

Haptix Games is a video game and interactive application development and publishing studio, and we have been a Microsoft shop for as long as I can remember. We have leveraged C#, MVC, and IIS for user experience; WCF, OData, and BizTalk for message exchanges; MSSQL and Entity Framework for storage. That is a lot of acronyms and even more complexity under the hood. Prototyping a concept usually involved all above-mentioned technologies, while the final solution release was either expensive, inflexible, or did not meet client expectations.

With the adoption of eXist-db our development and release workflows have become highly agile and more competitive. Utilizing eXist-db as a dark-data solution platform and not just another XML database allowed us to eliminate 80% of our Microsoft code base just by taking advantage of the built-in web server, low-level data manipulation using XQuery 3.0, restful data exchange, and native storage capabilities.

—Chris Misztur,
CTO,
Haptix Games,
Chicago, Illinois

easyDITA is an end-to-end solution for collaboratively authoring, managing, and publishing content using the DITA XML standard. Companies utilize easyDITA to reduce the cost and time to market to deliver content in a variety of formats and languages. By leveraging eXist, easyDITA is able to deliver customers exceptional ability to search, manage, localize, and publish content. eXist's schemaless design and flexible indexing system makes it easy to support customizations like reporting, analytics, and new content models without sacrificing performance or doing major redesigns.

—Casey Jordan,
cofounder,
easyDITA, Jorsek LLC,
Rochester, New York

We [at XML Team Solutions] help media and entertainment companies integrate sports news and data feeds. These feeds are predominantly XML. We use eXist for two things:

1. Regulating and preparing vendor web service XML for transmission to clients. Scheduled jobs access remote web services, preprocess, and pass on XML via HTTP Client to our main feed processor.
2. API to drive graphics for live television broadcast. API built from RESTXQ provides live updates of results to broadcaster clients. Currently uses JMS to sync from one write DB to two load-balanced readers. Also has an XForms "beat the feed" live score updater which mimics the incoming feed in case feed vendor is delayed.

We recently delivered a project for BBC Sports to deliver live broadcast information. eXist met all the requirements for speed, cost, and reliability for an API to deliver up-to-date scores and statistics to BBC television.

—Paul Kelly,
Director of Software Development,
XML Team Solutions Corp,
Canada

eXist-db is at the core of [the Office of the Historian's] open government and digital history initiatives. It powers our public website, allowing visitors to search and browse instantly through nearly a hundred thousand archival government documents. On the fly, it transforms our XML documents and query results into web pages, PDFs, ebooks, and APIs and data feeds. Its support of the high-level XQuery programming language and its elegant suite of development tools empower me and my fellow historians to analyze data and answer research questions.

The open source nature of eXist-db has delivered far more value to us than its simply being "free"; its active, welcoming, expert collaborative user community has helped us learn, discover eXist-db's plethora of capabilities, and find the best solutions to our research and publishing challenges. eXist-db belongs in the toolkit of all digital humanities, open government, and publishing projects.

—Joe Wicentowski,
Historian,
Office of the Historian,
U.S. Department of State

Contributing to the Community

There is a vibrant and supportive community around the eXist software, whose goal it is to make using eXist easy for beginners and as painless as possible for advanced developers. The eXist community prides itself on the agility and quality of its responses to support requests.

There are many ways to contribute to eXist and the community. You need not be a crack software engineer; even beginners asking questions on the mailing list can help others learn from their issues and encourage the developers to simplify or consider new approaches.

To get in touch with the eXist community, you have several channels available to you:

Email: the eXist-open mailing list

This is the official preferred mechanism, and your best bet for getting a quick answer. There are also the *eXist-development* and *eXist-commits* mailing lists; the former is used for technical discussion of features and fixes that go into eXist, and the latter is a feed of any changes made to the source code of eXist.

For further details, see *http://sourceforge.net/p/exist/mailman/*, *http://exist-open.markmail.org*, and *http://www.exist-db.org/exist/apps/doc/getting-help.xml*.

Stack Overflow: the exist-db tag

While the mailing lists should currently be considered the primary support mechanism, Stack Overflow is also becoming popular for asking eXist questions. You can find eXist questions and answers under the exist-db tag.

Twitter: @existdb
: The Twitter channel is monitored by the core developers of eXist. It's mostly used for announcements about eXist, as providing tech support in 140 characters is tough!

IRC: #existdb on irc.freenode.net
: The eXist IRC chat room is for the community, by the community. It can be a mix of users and developers and is often worth a visit, but getting a response can really depend on who is awake and logged in.

Individuals Using eXist

As a user of eXist, be sure to do the following:

Ask questions
: There are no stupid questions. Importantly, all questions and answers on the mailing list are archived so that others may learn from them also. Sensibly, you should search the archive first, to see if your question has already been answered; if not, or if the answer's not clear, then ask away!

Report bugs
: All software has bugs! eXist is no exception. If you think you have found a bug, it is probably best to discuss it on the *eXist-open* mailing list first, and then, if it's confirmed, log it in the eXist issue tracker on GitHub.

  If you want to report a bug, it's very important that the developers of eXist understand how to reproduce what you are seeing; if you can't describe the problem and its cause, it's very hard for the community to help you! Ideally you should provide a reproducible test case of the absolute minimum steps required to cause the issue. See "Getting Support" on page 413 for more detail.

Answer questions
: As you begin to use eXist, you will start to learn more and more things that other users may not know. Why not get some good karma back by answering some questions on the mailing list? After all, it's a community!

Evangelize
: If you're having a great time using eXist, or you are enthralled by some neat feature, tell your friends, and let us and everyone else know by writing a blog entry or article.

Organizations Using eXist

Open source developers are often working on a project for "the love of it," and many of the developers of eXist contribute much of their time to the project completely unpaid. Sadly, love for developing open source code with your friends does not necessarily equate to food or shelter. If you're part of an organization making free use of eXist, there are a number of ways that you can contribute back to the community:

Sponsor features or bug fixes
: Perhaps there is some feature that you wish that eXist had that would really help your project, or there is a bug that sometimes upsets your system. Your organization could financially sponsor a developer from the eXist community to add this feature or resolve that issue. Sponsoring eXist developers for small or large projects helps support them in their work on eXist and could provide new or improved functionality to the community. If you want to give something back financially but have no specific features or bug fixes in mind, just get in touch via *eXist-open* and the community will helpfully propose a project to meet your budget.

Friday afternoon eXist
: If you have developers in your organization, empowering them to spend a small amount of their paid work time contributing to the development of eXist can also be a great way to give back to the community. For example, Google allows its developers to work on open source projects on Friday afternoons and has realized various benefits from this.

Contracts and jobs
: Are you looking for someone who is an expert in eXist and XQuery and/or XML technologies? The *eXist-open* mailing list can be a great place to advertise. You will more than likely end up sponsoring one of the contributors to eXist, as they tend to be the people who really know it inside and out.

Support and maintenance contracts
: If you're serious about using eXist in your projects and running production systems on it, you will more than likely want the support and operational security afforded by purchasing a support contract for it. eXist Solutions provides a variety of support contracts and consultancy services for eXist. It was founded by core developers of eXist and contributes almost all of its resources back into developing the software. By working with eXist Solutions, you are closely supporting and funding eXist's development.

Authors Using eXist

If you're an author using eXist, here's how you can contribute:

Documentation
: eXist has a large set of documentation that accompanies it, but it is by no means complete or exhaustive. You do not have to be a developer to write documentation for eXist, and all improvements to the documentation are warmly accepted.

Developers Using eXist

Developers using eXist can give back in the following ways:

Bugs, patches, and new features

Found a bug? Want to submit a patch or new feature? Why not roll up your sleeves and get your hands dirty? In the beginning the eXist code base may seem intimidating in its size, but it's fairly modular and easy to get around. And if you have the skills, there is often no quicker way to get something fixed than to do it yourself, while hopefully learning a few new and interesting things along the way. Bug reports should be posted to the *eXist-development* mailing list first, and then logged in the issue tracker on GitHub. Patches can be submitted by means of a pull request to the eXist GitHub repository.

For further information about developing eXist, see "Developing eXist" on page 483.

Additional Resources

This section contains additional informational resources. It's compiled from our personal preferences and bookshelves, meaning there are many other good sources of information around. However, this list is a good place to start:

General

- W3C (World Wide Web Consortium)

  The W3C is the body that manages, among other things, the XML standards. The website is surprisingly easy to use, yet informative.

- W3 Schools

  For a quick high-level overview of any W3C standard with practical examples, try the W3 Schools.

XQuery

- *XQuery*, by Priscilla Walmsley (O'Reilly, 2007)

  This is probably the best XQuery book available in our opinion.

- *XQuery* wikibook, edited by Dan McCreary et al.

  The *XQuery* wikibook is an excellent resource for XQuery and eXist ,with the majority of the examples developed for eXist.

- XRX wikibook, edited by Dan McCreary et al.

  The XRX wikibook, like the XQuery wikibook, is an excellent resource when you're building applications atop eXist using REST and XForms.

- XPath and XQuery Functions and Operators 3.0 (W3C, 2014)

  The F+O specification is a great resource for quickly looking up the available functions and their specification for XQuery, XPath, and even XSLT.

- *XQuery: The XML Query Language*, by Michael Brundage (Addison-Wesley, 2004)

  This was a great book at the time it was published; while still relevant, it was released before the final XQuery 1.0 specification.

- *XQuery from the Experts: A Guide to the W3C XML Query Language*, by Don Chamberlin et al. (Addison-Wesley, 2004)

  Again, this book was released before the final XQuery 1.0 specification, but it is useful for those who want to know the nitty-gritty details like the formal underpinnings of the language.

- The XQuery Talk mailing list

  The *xquery-talk* mailing list is a great place to ask XQuery questions that are not specific to eXist-db.

XSLT

- *XSLT 2.0 and XPath 2.0: Programmer's Reference*, 4th Edition, by Michael Kay (Wiley, 2008)

  This is *the* book to have beside your keyboard if you ever want to do any serious XSLT programming.

- XSL-List

  The XSL-List is the best place to ask XSL questions and receive help. Note that it has an excellent archive; we suggest that you search that first for an answer before asking!

- XSLT Questions and Answers—FAQ, curated by Dave Pawson

  The XSLT FAQ is an incredible resource that has many answers from those who were involved in specifying XSLT and those recognized as subject experts.

- *XSLT*, 2nd Edition, by Doug Tidwell (O'Reilly, 2008)
- *XSLT Cookbook*, 2nd Edition, by Sal Mangano (O'Reilly, 2006).

  Also a very handy book to have when you only sporadically program XSLT; it contains many useful "recipes."

XForms

- XForms Tutorial and Cookbook wikibook, edited by Dan McCreary

The XForms wikibook is an excellent resource for XForms examples, especially as many of the articles are developed against eXist.

- *XForms Essentials*, by Micah Dubinko (O'Reilly, 2003)

  An excellent reference guide to have when you're working with XForms, with some good explanations of the W3C XForms specification.

XML Schema

- *Definitive XML Schema*, by Priscilla Walmsley (Prentice Hall, 2013)
- *RELAX NG*, by Eric van der Vlist (O'Reilly, 2003)

XSL-FO

- XSL, XSL-FO FAQ, curated by Dave Pawson

  The XSL-FO FAQ is in a similar vein to the XSLT FAQ and likewise is an invaluable resource, with many questions answered by subject experts.
- *XSL Formatting Objects: Developer's Handbook*, by Doug Lovell (Sams, 2003)

CHAPTER 2

Getting Started

This chapter takes you through the first steps in using eXist. It handles subjects like downloading and installing, starting and stopping, running the examples, and demonstrates some of eXist's capabilities on a "Hello World" level. In other words, like the chapter title says, it will get you started.

If you have used eXist before, you may like to skip over this chapter.

Downloading and Installing eXist

This section takes you through the steps necessary for getting eXist up and running on a *standalone development* system. That is to say, we keep things simple and don't spend time on more advanced subjects such as database security, tuning, performance, embedded mode operation, and the like. Those subjects and more are covered in the chapters to come.

Be aware that installing eXist for *production* purposes (e.g., as the engine behind a public website) requires much more thought and planning. Security, especially, requires attention in those kinds of more public situations. Also, if you plan to use eXist with some really huge datasets, you probably need a different setup than that described here. For information on installing eXist in a server environment, see "Installing eXist as a Service" on page 405.

Preconditions

eXist can be installed on almost all versions of Linux, Windows, and Mac OS X. The deciding factor is whether or not your OS (operating system) supports at least Java version 1.6 (1.7 is recommended). If it does, then eXist should run.

In order to run the eXist installer, you must have a working JRE (Java Runtime Environment) or JDK (Java Development Kit), version 1.6 or newer. The eXist team regularly tests eXist with the Oracle and OpenJDK JRE and JDKs, but the community reports that the IBM JDK (among others) also works.

You can download the Oracle JDK from *http://www.oracle.com/technetwork/java/javase/downloads/index.html*.

To check whether you have the right Java version (and have installed it correctly), open a terminal/command-line window and type `java -version`. You should see a message telling you which version of Java you're running.

Downloading eXist

Downloading eXist is easy. Go to *http://www.exist-db.org*, navigate to the download section, pick the right distribution, and download it. For getting started, pick the latest stable distribution. The filename will probably look like *eXist-db-setup-<version>-rev<XXXXX>.jar*.

This book was based on the 2.1 release of eXist (*eXist-db-setup-2.1-rev18721.jar*), but by the time you read this, a newer version may be available.

Things to Decide Before Installing

Of course, you can go ahead now and run the installer using the defaults provided. However, there are probably a number of things you want to decide *before* firing up the installer:

Installation directory

Where are you going to install eXist? For a "getting started" installation, this is not extremely important; you can use the default suggested by the installer or any other location you like (provided it is writable by the installer).

However, there are a number of reasons why the installation directory matters more than is usual for a software installation. Firstly, the default for the data directory (where eXist stores its data) is *inside* the installation directory, as described shortly. Secondly, logging and temporary directories are also inside the installation directory.

Having frequently written locations inside a software installation might be problematic because security sometimes does not allow this, or it causes performance degradation. When you start to do more serious work, make sure the important locations are included in your backup.

We will refer to the installation directory as *$EXIST_HOME* throughout this book.

Data directory

This is the directory where eXist stores the content of its database. The installer will propose a default that's *inside* eXist's installation directory (*$EXIST_HOME/webapp/WEB-INF/data*, to be precise). If you just plan to play around a bit or do some development work, keep the default. You can always change it later.

However, if things get serious, like on a production server, make sure that this directory is writable, located on a volume that is sufficiently fast for updates, and backed up (which is not always the case for program file directories).

Administrator password

The installer will ask you to provide an administrator password. This is *not* your operating system's administrator password, but the initial password used for eXist's administrator's account (called `admin`). You are strongly encouraged to set an administrator password on all installations of eXist. If you don't, eXist will use an empty password, so anyone who tries an empty password would have full access to your eXist instance.

Memory settings

The installer allows you to set the amount of memory reserved for eXist's JVM and its internal cache. Common settings are shown in Table 2-1.

Table 2-1. eXist installation memory settings

Max memory	Cache memory	Remarks
512 MB	64 MB	Don't go any lower than this, or eXist will not run properly.
1,024 MB	128 MB	This is the default setting and is fine for small development use.
2,048 MB	256 MB	If your machine has enough memory to spare, then use at least this.

Packages/apps to install

For getting started purposes, we recommend keeping everything checked (this book assumes that you did!).

Installing eXist

Start the installer in one of the following ways:

For desktop-driven systems
: If Java is set up correctly, on many systems that provide a GUI, double-clicking the downloaded *eXist-db-setup-<version>-rev<XXXXX>.jar* file will fire up the graphical installer.

On all GUI systems, from the command line
: Open a terminal/command-line window and enter the following command: `java -jar eXist-db-setup-`*`<version>`*`-rev`*`<XXXXX>`*`.jar` (of course, the name of the file you just downloaded). This will launch the graphical installer.

On non-GUI systems, from the command line
: If you are on a system that does not provide a GUI environment—for example, a remote server—you can entirely install eXist from the terminal. At the terminal/command-line window, enter the following command: `java -jar eXist-db-setup-`*`<version>`*`-rev`*`<XXXXX>`*`.jar -console` (using the name of the file you just downloaded).

As usual with installers, follow the instructions on the screen to complete the installation. You'll be asked to enter the information prepared in the previous section. Let the installer run its course, and that's it!

Post-Installation Checks

By default, eXist uses two TCP ports:

Port 8080
: This port is used for all the normal HTTP communication.

Port 8443
: This port is used for the confidential HTTPS communication. By default, eXist uses a *self-signed certificate*, which, while more secure than using no certificate, should not be considered for production use. You may also see a warning about the self-signed certificate when accessing this from a web browser.

If one of these ports is used by another application on your system, you either have to make this other application change its ports or change the port settings for eXist.

The easiest way to find out if something is using these ports is, before starting eXist, to visit *http://localhost:8080/* and *https://localhost:8443/*. If nothing happens, the ports are probably free and you can go ahead.

Changing eXist's TCP port usage is explained in "Changing Jetty Settings: Port Number and URL Prefix" on page 205.

This book assumes eXist is running on *localhost* using the standard TCP port numbers 8080 and 8443, so you'll see URLs like *http://localhost:8080/...* throughout the book.

Starting and Stopping eXist with a GUI

If you're on a system with a GUI, the installer will have created a menu entry and/or a desktop icon called *eXist database*. If you're on a command-line-only system, go to "Starting and Stopping eXist from the Command Line" on page 24.

Clicking the eXist database icon starts eXist and also fires up a little control application that should be visible in the system tray (or its equivalent on your system) as a dotted X. For instance, on a Windows 7 machine it looks like Figure 2-1.

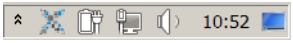


Figure 2-1. The eXist control application in a Windows 7 system tray

Clicking it opens a little menu that gives you further control of eXist (like stopping the server) and lets you do a few other useful things, as shown in Figure 2-2.

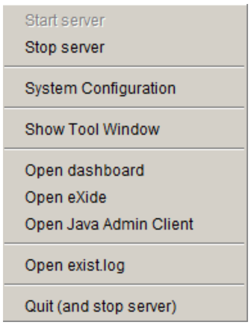


Figure 2-2. The menu of the eXist control application

If for some reason this doesn't work, open a command window in *$EXIST_HOME* and type `java -jar start.jar`. This should fire up the control application and the database. If this works, you're probably best off creating a shortcut or menu entry for it manually. If still nothing happens, read the next section.

After starting the database, open your browser and visit *http://localhost:8080/exist*. If a nicely tiled screen appears (like in "The Dashboard" on page 24), you've succeeded!

Starting and Stopping eXist from the Command Line

If you don't want to or can't work with the GUI niceties, you can also start eXist from the command line. For this, open up a command-line window and navigate to *$EXIST_HOME/bin*. There you'll find several command files in both the Windows (**.bat* versions) and Unix/Linux/Mac (**.sh* versions) variants. For starting and stopping, do the following:

`startup`
: This will fire up eXist.

`shutdown -p` *`adminpassword`*
: This will stop the running eXist instance. It needs the administrator password.

A First Tour Around Town

This section will give you a quick tour of eXist's highlights and attractions, including the user interface and what's on your disk.

The Dashboard

The home screen of eXist since 2.1, *http://localhost:8080/exist*, is called the *dashboard*; it is a set of tiles linking to various applications and utilities. The initial set shows the default tiles provided with eXist. You may install additional ones via the Package Manager, or if you start developing applications of your own with the Packaging System (see "Packaging" on page 227), those can appear here too.

Now, most of the functionality provided through the dashboard—stuff like eXide and the function documentation—is important, and you will probably use it often. It is therefore well worth your time to familiarize yourself with the smorgasbord offered (see Figure 2-3).

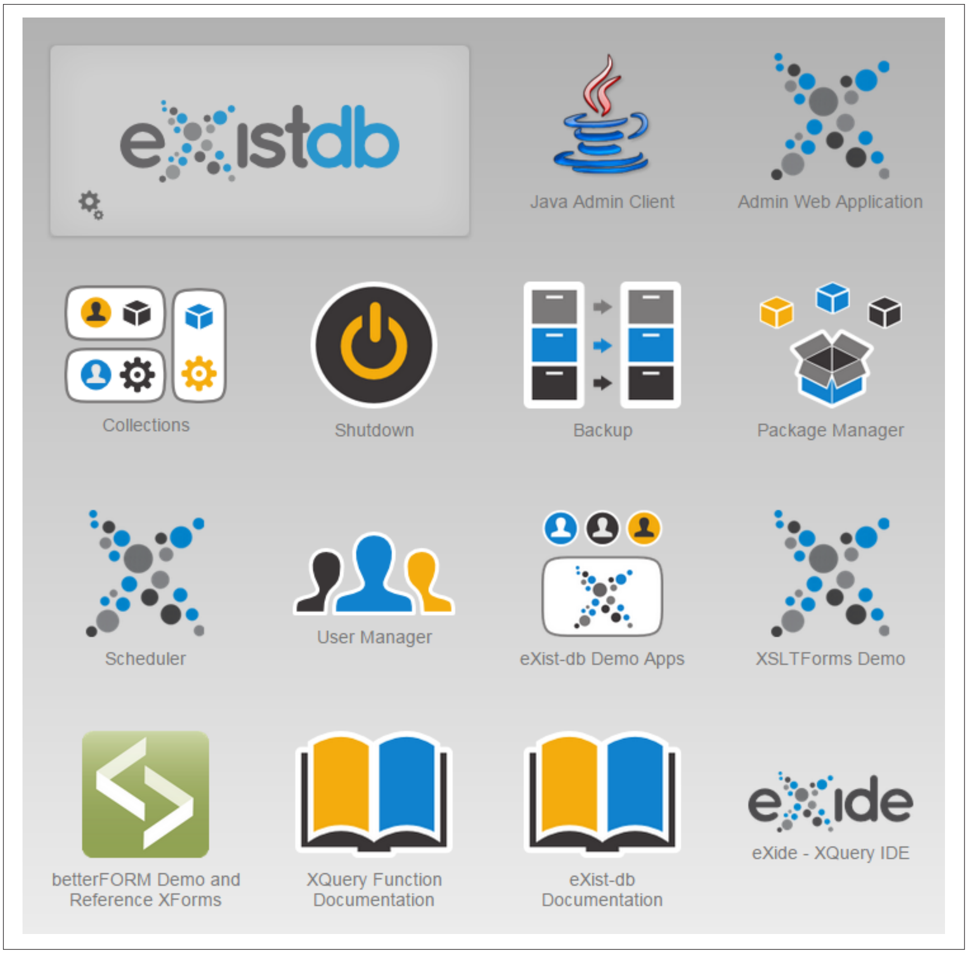


Figure 2-3. The eXist dashboard

The tiles provided by default are:

Java Admin Client
: This tile provides a Java Web Start, a.k.a. JNLP (Java Network Launching Protocol), link to eXist's Java Admin Client application. Use this if you want to access an eXist installation remotely, from a system that does not have eXist installed. For local use you're better off starting the Java Admin Client directly (e.g., through the control application's menu, as shown in Figure 2-2). Read more about the Java Admin Client in "The Java Admin Client" on page 29.

JNLP does not work well with all browsers. You might just get a "Save downloaded file" dialog when pressing this tile.

Admin Web Application
: This tile gives access to the original (pre-2.x) administrator web client application. There is still some functionality there that has not yet appeared in the new interface, such as profiling queries and index overview.

Collections
: This tile starts a collection browser that enables you to control the contents of the database.

Shutdown, Backup
: These applications do what their titles suggest.

Package Manager
: A *package* is a set of related files that together provide some kind of functionality—for instance, an application or library. The Package Manager allows you to manage (view, install, and uninstall) packages in your eXist database. When you open it, you can see that most of the functionality behind the tiles of the dashboard is provided by separate packages.

  Packages can come from the eXist public repository—you can see the packages available there by selecting the *available* option at the top of the Package Manager—or they can be distributed as separate package files with the extension *.xar*.

  It is also possible (and even advised!) to design your own applications for use with the Package Manager and distribute them using *.xar* package files. You'll learn how to do this in "Packaging" on page 227.

User Manager
: This tile allows you to control the user population of the eXist database. You can create, edit, and delete users and groups.

betterFORM Feature Explorer, betterFORM Demos, XSLTForms Demo
: eXist has two built-in ways of doing XForms: betterFORM™ and XSLTForms™. These applications provide you with demos and overviews. Find more information in "XForms" on page 254.

eXist-db Demo Apps
: This tile is a collection of applications that demonstrate some of eXist's capabilities.

XQuery Function Documentation
: This is an application you'll probably use very often. It provides an overview of all the functions available and their documentation in both the standard eXist extension modules and your own XQuery modules. There's more about modules in Chapter 7.

eXist-db Documentation App
: This app provides access to the eXist documentation.

eXide
: eXide is a cool, handy, fully integrated editor for working with XQuery, XML, and other resources stored in eXist. You can use it for a multitude of activities, from writing complete applications to fiddling around and experimenting. Don't miss it. Find more information in "eXide" on page 374.

Playing Around

If you're like us, at this point you'll want to play around, try some XQuery, store some XML, and perform other familiarization rituals. Get your feet wet and splash around (without going into the deep end). Here is the quick recipe:

1. Open the dashboard: *http://localhost:8080/exist*.
2. Click on the eXide tile.
3. Click Login and log in as `admin` with the password set during installation.
4. Directly type some XQuery and run it.
5. If you want to see what's in the database, click File→Manage (or press Ctrl-Shift-M).
6. If you want to save your work or put some related files together, create a collection for this underneath */db*.

What's in Your Database

You can look inside the database using, among others, the Collections app in the dashboard or the eXist Java Admin Client (see "The Java Admin Client" on page 29). You'll see something that looks like a disk directory structure (but of course isn't). To explain the terminology, what you might think of inside your database as a *directory* is called a *collection* in XML database geek speak (more about this in "Terminology" on page 88). Here are the most important collections:

/db
: The root collection in the database is always */db*. You can't change this.

/db/system

This is where eXist stores important configuration information (e.g., about users, groups, and versioning). You shouldn't change any of this information by hand or programmatically, with the exception of what's inside */db/system/config*.

/db/system/config

This collection is used to store the collection-specific configuration for eXist, like validation, indexes, and triggers. If you look underneath, you'll find a (partial) copy of the existing database structure with *collection.xconf* files here and there. These (XML) files contain the collection configuration. Read more about this in "Implicit Validation" on page 246 and "Configuring Indexes" on page 275.

*/db/apps/**

These are the root collections for the packages, installed during installation and manually later. Underneath these is their code and data. If you're ever going to write applications yourself (Chapter 9), you'll create your own subcollections here.

What's on Your Disk

Now let's look at some interesting and/or important locations on your disk for eXist.

> There are rumors on the grapevine that the basic file structure will change in future versions, so be aware if you use this book with a later version than 2.1.

$EXIST_HOME/

This is eXist's home directory.

$EXIST_HOME/conf.xml

This is eXist's main configuration file. If you peek inside (it's well documented), you'll find entries for, for instance, all kinds of default behavior, the location of the database (in *db-connection/@files*), cache sizes, the indexer, and the built-in XQuery modules.

$EXIST_HOME/tools/jetty/etc/jetty.xml

This is the Jetty web server's configuration file (eXist uses Jetty to communicate with the world). There are several interesting things you might want to change using this file, like the TCP port numbers and the default URL prefix *exist/*.

$EXIST_HOME/webapp/WEB-INF/

This location defines the eXist web application. It holds several important configuration files and is the default base location for the database and the logfiles.

$EXIST_HOME/webapp/WEB-INF/controller-config.xml
: This tells eXist what to do when a request with a certain URL is entered. There's more information in "The controller-config.xml Configuration File" on page 206.

$EXIST_HOME/webapp/WEB-INF/data/
: This is the default location for eXist's database (unless you specified somewhere else during the installation process).

: If you peek inside this directory, you'll find underneath the *fs* subdirectory all the non-XML files stored in the database. However, your XML files are not there; they have seemingly disappeared. Don't despair: they're absorbed into the **.dbx* files you see in the root of the database directory. You'll find more information about this in "Help: Where Is My XML?" on page 87.

> You might be tempted to change the non-XML content underneath the *fs* subdirectory directly. *Don't do this*. It will ruin the database's internal administration. Use only the normal mechanisms for this, like WebDAV, the dashboard, or the Java Admin Client tool.

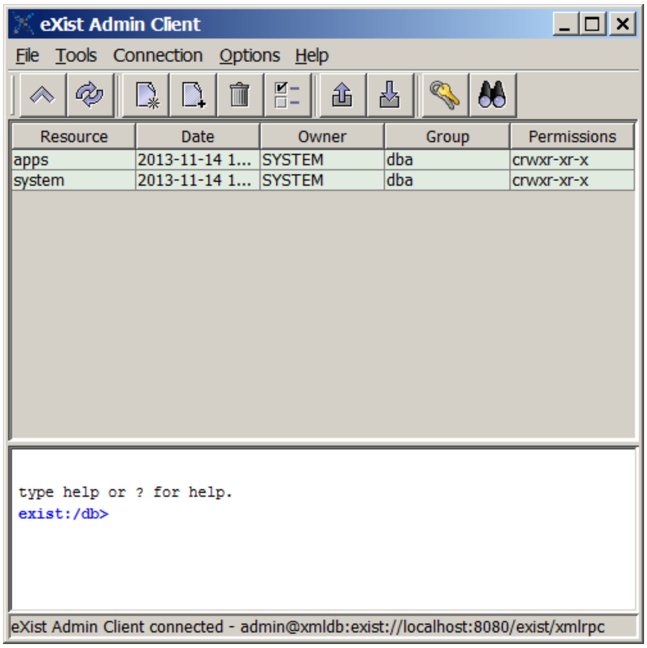
$EXIST_HOME/webapp/WEB-INF/logs/
: Here you'll find several logfiles that can help you find out what's going on underneath eXist's hood.

The Java Admin Client

Through the eXist controller application (visible in the system tray), you can start the *Java Admin Client*. This pops up a small and, admittedly, rather old-fashioned-looking program. It allows you to do maintenance work on the database like backups and restores, imports and exports, checking and setting properties, and creating collections. Figure 2-4 shows how it looks on a freshly installed database.

The eXist Client tool is a standard GUI application, and its functionality speaks for itself.

Most of this tool's functionality is also present in the new dashboard application, so there's a good chance you'll never need it. However, there are circumstances in which it can be useful, such as when you're working on a production server where you don't want the dashboard to be present.



Resource	Date	Owner	Group	Permissions
apps	2013-11-14 1...	SYSTEM	dba	crwxr-xr-x
system	2013-11-14 1...	SYSTEM	dba	crwxr-xr-x

```
type help or ? for help.
exist:/db>
```

eXist Admin Client connected - admin@xmldb:exist://localhost:8080/exist/xmlrpc

Figure 2-4. The main screen of the eXist Java Admin Client tool

Getting Files into and out of the Database

eXist is an XML database. Its primary storage concern is XML documents. It can also hold your XQuery files, and any other resources needed by your application. So how do you get files in and out of it?

Collections app
: Browse to eXist's Collections application (available through the dashboard). This allows you to look through the contents of your database and maintain the collections and resources.

eXide
: eXist's built-in native IDE, called eXide, has facilities for uploading and downloading files. Click File→Manage (or press Ctrl-Shift-M).

WebDAV
: eXist's WebDAV (Web-based Distributed Authoring and Versioning protocol) interface allows you to access the contents of the database just like it was any other file store available to your OS. The address to use is *http://localhost:8080/exist/webdav/db/* or, when your OS requires safe URLs (like Windows 7), *https://localhost:8443/exist/webdav/db/*.

Exactly how to work with a WebDAV server and which client tool to use is platform-specific. Some operating systems, like Windows, will allow you to integrate it more or less into the normal file browsing capabilities, while others need special client tools. Read more about this in "WebDAV" on page 305.

Java Admin Client
: eXist's Java Admin Client tool (see "The Java Admin Client" on page 29) also has some basic facilities for getting files into and out of the database.

External IDE
: Some external IDEs, such as oXygen, provide you with the option to work with eXist natively. This includes importing/exporting files. Find more information in "oXygen" on page 375.

Programmatically
: Of course, you can import and export files programmatically by writing some XQuery code that performs what you want on the database. That's okay within applications, but a bit cumbersome for now. For more information, see "Controlling the Database from Code" on page 107 and Chapter 13.

Ant
: eXist provides a library for the Ant build tool to automate common tasks like backup/restore or importing a bunch of files. This method is recommended if you need to repeat batch tasks on your database. There's more information in "Ant and eXist" on page 379.

Hello eXist!

This section performs a first exploration of the fundamental mechanisms in eXist; that is, how you get it to actually do something—store/retrieve/filter information, show a web page, transform XML, and more. In other words, this section is an extended "Hello world" example in which, in a (very) shallow way, we touch upon the important processing features of the platform.

For most of the examples, the output is not shown because we want to encourage you to try this yourselves using the provided example code. We assume you've installed the example code and know how to access it, as described in "Accompanying Source Code" on page 15.

Hello Data

In the example code for this book, there is an XML file in */db/apps/exist-book/getting-started/xml-example.xml* that looks like Example 2-1.

Example 2-1. XML file

```
<?xml version="1.0" encoding="UTF-8"?>
<Items>
    <Item name="Bogus item">This is a complete bogus item</Item>
    <Item name="Funny item">Ha, ha, very funny indeed!</Item>
</Items>
```

Accessing data (and also scripts) is done through the eXist *REST interface*. To see it in action, fire up your browser and visit *http://localhost:8080/exist/rest/apps/exist-book/getting-started/xml-example.xml*.

The result is that you see exactly the file from Example 2-1. Not impressive, maybe, but hey, this is only the beginning.

The REST interface allows you to directly query this file. For instance, assume you're interested in the first item only. You can access it by adding a `_query` parameter:

```
    http://localhost:8080/exist/rest/apps/exist-book/getting-started/
      xml-example.xml?_query=//Item[1]
```

The result will be:

```
    <exist:result xmlns:exist="http://exist.sourceforge.net/NS/exist" exist:hits="1"
        exist:start="1" exist:count="1">
      <Item name="Bogus item">This is a complete bogus item</Item>
    </exist:result>
```

Because it's a query, eXist wraps the result in an `exist:result` element with additional information in its attributes. There are other query parameters that will let you limit the size of the result set and even retrieve the results block by block. More information about the REST interface can be found in "Querying the Database Using REST" on page 94.

Hello XQuery

Of course, the main language when you are dealing with eXist is XQuery, which is *the* language to access XML databases. Put your XQuery script in a file (or database document) with the extension *.xq*. Example 2-2 shows you a basic way to output some XML.

Example 2-2. Basic XQuery code returning XML

```
xquery version "3.0";

let $msg := 'Hello XQuery'
return
    <results timestamp="{current-dateTime()}">
        <message>{$msg}</message>
    </results>
```

Now what if you want to show the result as an HTML page instead? That's called *serialization*, and Example 2-3 shows one of the ways to do it.

Example 2-3. Basic XQuery code returning HTML

```
xquery version "3.0";

declare option exist:serialize "method=html media-type=text/html";

let $msg := 'Hello XQuery'
return
    <html>
        <head>
            <title>Hello XQuery</title>
        </head>
        <body>
            <h3>It is now {current-dateTime()} and so {$msg}!</h3>
        </body>
    </html>
```

XQuery-initiated readers might have noticed that we did not declare the `exist` namespace prefix. eXist has most eXist-specific namespace prefixes predeclared for you, so you don't have to explicitly mention them in your code.

Hello XSLT

XSLT is built into eXist using (by default) the Saxon XSLT processor. The examples contain a simple stylesheet to show you how this works. The stylesheet in Example 2-4 takes the XML from Example 2-1 and turns it into an HTML page.

Example 2-4. Transformation of the example XML into HTML

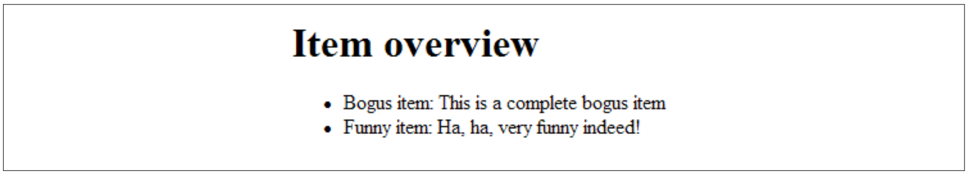
```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
    <xsl:template match="/">
        <html>
            <head>
                <title>Hello XSLT</title>
            </head>
            <body>
                <h1>Item overview</h1>
                <ul>
                    <xsl:for-each select="//Item">
                        <li>
                            <xsl:value-of select="@name"/>:
                            <xsl:value-of select="."/>
                        </li>
                    </xsl:for-each>
                </ul>
            </body>
        </html>
    </xsl:template>
</xsl:stylesheet>
```

To run an XSLT stylesheet over some XML from within XQuery, you need to use an *extension module*. Extension modules are, well, extensions to the basic XQuery capabilities. eXist has lots of them, and we devote all of Chapter 7 to the subject. An overview (and all function documentation) is accessible through the XQuery Function Documentation app, available through the dashboard.

Transforming documents with XSLT is done with the `transform` extension module. A little XQuery script that performs this transformation is shown in Example 2-5, and its result in Figure 2-5.

Example 2-5. Using XSLT with the transform extension module

```
xquery version "3.0";

declare option exist:serialize "method=html media-type=text/html";

transform:transform(
    doc("/db/apps/exist-book/getting-started/xml-example.xml"),
    doc("/db/apps/exist-book/getting-started/convert-items.xslt"),
    ()
)
```


Item overview

- Bogus item: This is a complete bogus item
- Funny item: Ha, ha, very funny indeed!

Figure 2-5. Result of the XSLT transformation

Notice that the `transform` extension module was not explicitly declared in the XQuery script. eXist does this implicitly for you. The third parameter of `transform:transform`, which here is passed an empty sequence, can contain parameters for the stylesheet.

More about using XSLT transformations within eXist can be found in "XSLT" on page 238.

Hello XInclude

eXist can also do XInclude processing for you. This means that on the way out, when the final results of an XQuery operation are serialized, they are inspected for `xi:include` elements. When found, these references are expanded.

An interesting feature of the XInclude processing is that you can also refer to XQuery scripts. The script is executed and the result included. Example 2-6 demonstrates this.

Example 2-6. XInclude

```
<?xml version="1.0" encoding="UTF-8"?>
<XIncludeEnvelope xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="xinclude-content.xml"/>
  <xi:include href="hello-world-1.xq"/>
</XIncludeEnvelope>
```

hello-world-1.xq is the XQuery script presented in Example 2-3. The included XML file contains the fragment shown in Example 2-7.

Example 2-7. XML fragment to include with XInclude

```
<XIncludeContent>This element was included by the XInclude processing
    in eXist. Yes!</XIncludeContent>
```

Now if you retrieve *xinclude-envelope.xml* from the database, the XInclude references are resolved, resulting in Example 2-8.

Example 2-8. The result of the XInclude processing

```
<XIncludeEnvelope xmlns:xi="http://www.w3.org/2001/XInclude">
  <XIncludeContent>
    This element was included by the XInclude processing in eXist. Yes!
  </XIncludeContent>
  <results timestamp="2013-02-21T13:12:21.399+01:00">
    <message>Hello XQuery</message>
  </results>
</XIncludeEnvelope>
```

There are more features to XInclude processing, like `fallback` instructions and the ability to pass parameters to XInclude-d XQuery scripts. Read more about this in "XInclude" on page 243.

Hello XForms

XForms is a W3C standard that defines declaratively the contents of a form on a web page, its behavior, and its result. It's neither a thick nor a complicated standard.

However, trying to fully understand what's going on, and all the details (like forms submission), can be challenging!

eXist has two third-party XForms processors built in that you may choose between: betterFORM and XSLTForms. They allow you to create pages that contain XForms logic and have them rendered and executed as the XForms specification describes. To see this in action, take a look at Example 2-9, which will be rendered using betterForm.

Example 2-9. A simple XForms example

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ev="http://www.w3.org/2001/xml-events"
    xmlns:xf="http://www.w3.org/2002/xforms">
    <head>
        <title>Hello XForms</title>
    <!-- The XForms data model: -->
        <xf:model id="xforms-data-model">
            <xf:submission action="hello-xforms-submit.xq" id="submit-id"
              method="post"/>
            <xf:instance xmlns="">
                <Data>
                    <Name/>
                    <Date/>
                </Data>
            </xf:instance>
            <xf:bind id="NameBind" nodeset="/Data/Name" required="true()"
              type="xs:string"/>
            <xf:bind id="DateBind" nodeset="/Data/Date" required="true()"
              type="xs:date"/>
        </xf:model>
    </head>
  <!-- -->
    <body>
        <h1>Hello XForms</h1>
        <xf:group>
            <xf:input bind="NameBind">
                <xf:label>Name</xf:label>
            </xf:input>
            <xf:input bind="DateBind">
                <xf:label>Date</xf:label>
            </xf:input>
            <xf:submit submission="submit-id">
                <xf:label>Submit</xf:label>
            </xf:submit>
        </xf:group>
    </body>
</html>
```

This example will let you fill in a simple form. Notice that because we bound the `/Data/Date` field to the data type `xs:date`, the form will automatically show a date picker for the date input field! Have a look at the underlying HTML code. As you can see, betterForm adds lots of functionality to make all this happen.

When you press the submit button (after filling in the values), the posted XML will show through the *hello-xforms-submit.xq* page, as Example 2-10 demonstrates.

Example 2-10. Getting the results of an XForm

```
xquery version "1.0" encoding "UTF-8";

<XFormsResult>
    {request:get-data()}
</XFormsResult>
```

You can find more on XForms in "XForms" on page 254.

CHAPTER 3
Using eXist 101

This chapter contains an introduction to actually *using eXist* to build software. It will take you by the hand and guide you step by step along the winding XPath roads, through the XQuery meadows, along some RESTful paths, over the index mountain, and to many, many other wonderful places—all in a quest for the golden ring of an understanding that binds it all (which will definitely not be thrown into some volcano by Gollum).

This chapter guides you through building a simple web application that uses, analyzes, and enables searching of some Shakespeare plays. We start at the very beginning (creating the application's collections, adding data), and end with a simple but usable piece of software. Along the way, we will tell you where to find more information about the subjects covered. Everything is done natively through eXist; no external IDE is required.

To make ourselves clear: the goal of this chapter is not to teach you XQuery. We assume you have at least a basic understanding of this language, including some experience with XPath. However, we do try to explain what we're doing with some of the XQuery, so if you don't know any XQuery yet, do not despair!

Preparations and Basic Application Setup

The application we're going to build needs a set of data to work with. eXist comes with an example application (one of the eXist-db demo apps, available through the dashboard) that contains a nice, consistent, and large enough set of example data to be useful for us. So we're going to reuse this and, in doing so, show you how to export and import files.

The tools provided alongside eXist for working with the database's content (see "The Dashboard" on page 24) have the ability to copy content from one database collection to another. For instance, if you look at the database through the dashboard's collection browser, you'll find copy and paste facilities. However, sometimes you have to export and import content from the filesystem, so that's a good place to start our quest.

eXist Terminology

To prevent you from getting confused, let's introduce some eXist terms to you first:

Collections
: What you may typically think of as a *directory* or *folder* in the filesystem world, is actually called a *collection* inside eXist's database. There are some subtle and important differences with collections that you will learn about shortly.

Resources
: What is called a *file* in the filesystem world is called a *resource* inside eXist's database. Resources can be anything you usually store in a file: images, CSS files, XQuery scripts, and, of course, XML documents.

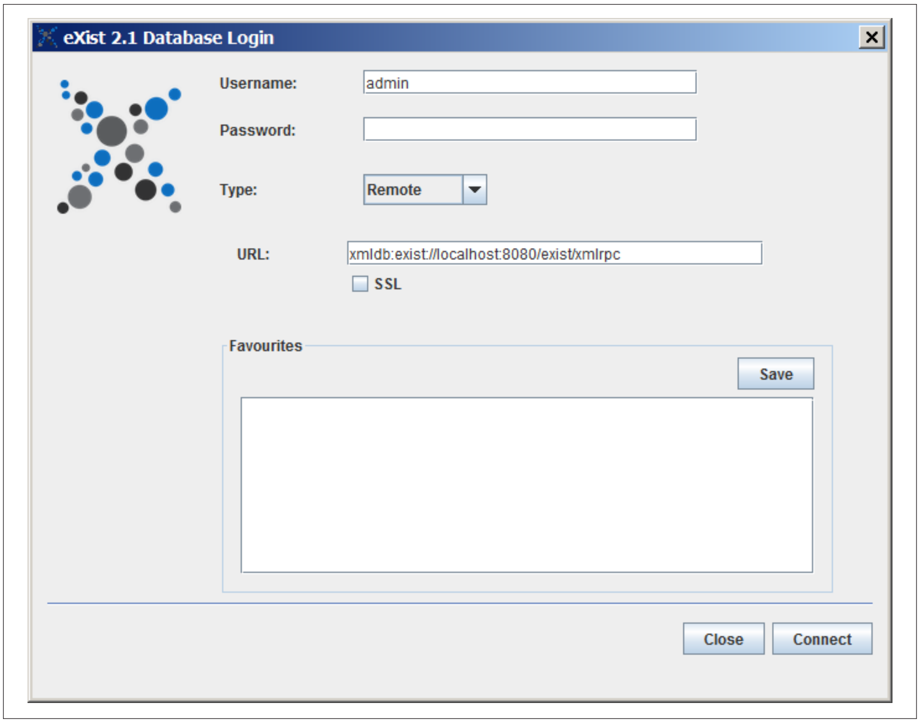
Documents
: A resource containing well-formed XML is called a *document*.

See also "Terminology" on page 88.

Exporting Documents from eXist

First, we're going to export the documents from eXist (afterward, we'll pretend they came from somewhere else). If you have a WebDAV connection set up (see "Getting Files into and out of the Database" on page 30), this is really easy: just drag and drop the necessary files from the eXist collection to somewhere on your filesystem. If you don't (yet) have WebDAV working, you can use the Java Admin Client as follows:

1. Start the Java Admin Client by clicking the eXist application icon in your system tray (see "Starting and Stopping eXist with a GUI" on page 23) and choose *Open Java Admin Client*. You'll see the screen shown in Figure 3-1.



Username: admin
Password:
Type: Remote
URL: xmldb:exist://localhost:8080/exist/xmlrpc
SSL

Figure 3-1. The opening screen of the Java Admin Client

2. Log in as `admin` (using the password you set when you installed eXist, as described in "Downloading and Installing eXist" on page 19) with the URL *xmldb:exist://localhost:8080/exist/xmlrpc*.
3. After a successful login, the screen shown in Figure 2-4 appears. Navigate to the collection *apps/demo/data*.
4. Select *hamlet.xml* and click the menu command File→Export a resource to a file.... Dump it on your disk somewhere. Repeat this for *macbeth.xml* and *r_and_j.xml* (guess what: *Romeo and Juliet*!).
5. Change the filename of the *Romeo and Juliet* play from *r_and_j.xml* to *r and j.xml* (replace the underscores with spaces). This will illustrate an important property of eXist later on.

So now we have three Shakespeare plays in XML markup on our disk. Let's pretend these files came from somewhere else and import them into our database.

Designing an Application's Collection Structure and Importing Data

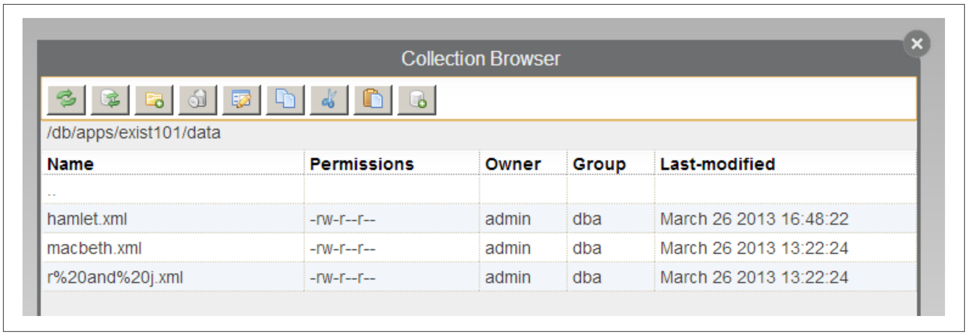
When you write an application in eXist (or anywhere else), you need a place to store your code and the accompanying data. Now, in eXist you can design any collection structure you like, but it is customary to store applications underneath the */db/apps* collection. So, that is where we're going to put our 101 application.

In this example, we'll consider our data, the plays, *static* (immutable) data. However, most applications also have *dynamic* data (data your application creates, updates, uploads, etc.). There is a debate as to whether (a subcollection of) */db/apps/<yourapp>* is a good location for this data. Some application designers argue that you should be consistent and keep everything in one place. But storing your dynamic data somewhere else (e.g., in */db/data/<yourapp>*) has benefits of its own. For instance, you can more easily update your application's code without losing the accumulated dynamic data. We won't worry about this in our 101 course, but make sure to give it some thought if you're going to build a real application.

You could use the Java Admin Client again to create the necessary collections and import the plays (there's an Import Files button in the toolbar), but let's check out another useful tool. Close the Java Admin Client, open a web browser, and follow these steps:

1. Browse to the dashboard (*http://localhost:8080/exist/*).
2. Log in as `admin` (click on "Not logged in" in the upper-left corner).
3. Click on the Collections tile.
4. Navigate to */db/apps*.
5. Create a fresh */db/apps/exist101* collection (the New Collection command is in the toolbar).
6. Navigate into the */db/apps/exist101* collection and create the */db/apps/exist101/ data collection*.
7. Navigate to the */db/apps/exist101/data* collection and click "Upload resources" in the toolbar.
8. Upload the plays we just downloaded into the collection.

The collection browser should now look like Figure 3-2.



/db/apps/exist101/data

Name	Permissions	Owner	Group	Last-modified
..				
hamlet.xml	-rw-r--r--	admin	dba	March 26 2013 16:48:22
macbeth.xml	-rw-r--r--	admin	dba	March 26 2013 13:22:24
r%20and%20j.xml	-rw-r--r--	admin	dba	March 26 2013 13:22:24

Figure 3-2. The collection browser after we upload the plays

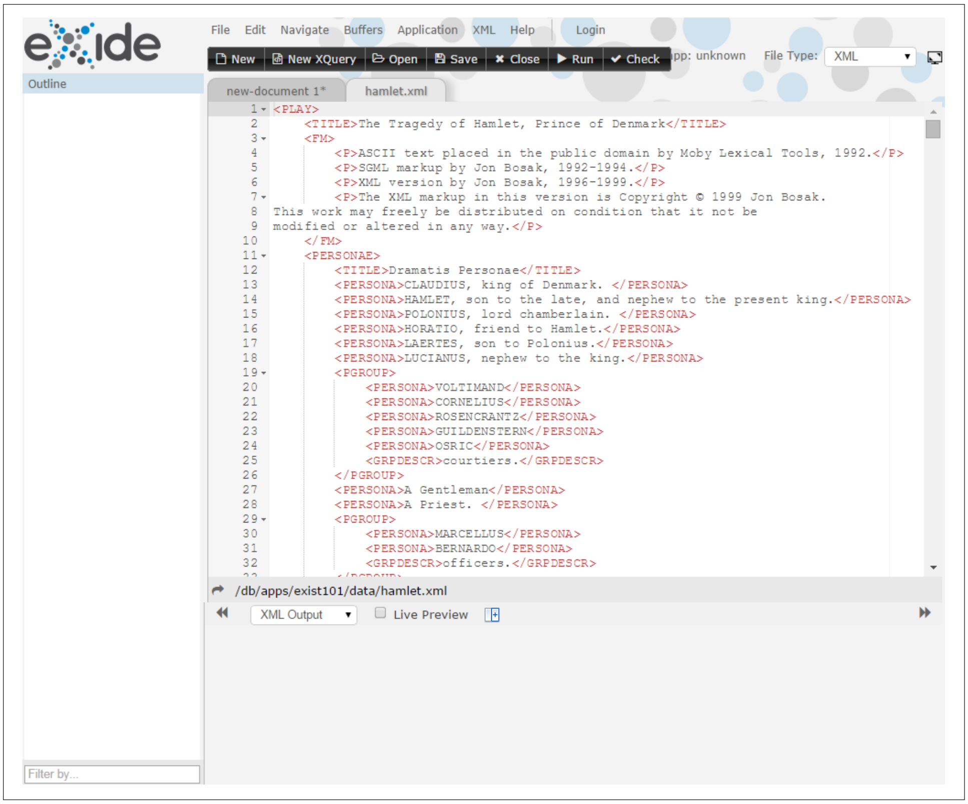
Look at the name of the *Romeo and Juliet* file. Instead of *r and j.xml*, it is now called *r %20and%20j.xml*. What happened? Well, names of collections and resources inside eXist are always *URL-encoded URIs*. Reserved characters, according to the URL encoding rules, are percent-encoded. A space character is one of these, so that explains the *%20* codes. More about this can be found in "Use URIs" on page 91. We'll come back to how to handle these names later.

Viewing the Data

Let's pretend for a moment we did not have the data on our disk before we imported it into the database. Instead of viewing a file on disk, how can we view XML (and other) resources stored in eXist?

The easiest and most versatile way of working with stored data is through an editor that is connected with eXist. Luckily for us, eXist has a built-in IDE, *eXide*, which we can use to view and edit files as follows:

1. Browse to the dashboard (*http://localhost:8080/exist/*).
2. If you're not already logged in, log in as `admin` (click on "Not logged in" in the upper-left corner).
3. Click on the "eXide - XQuery IDE" tile. eXide will open in a new browser window or tab.
4. Click Open, navigate to */db/apps/exist101/data*, and open *hamlet.xml* (Figure 3-3).


eXide
Outline
File Edit Navigate Buffers Application XML Help Login
New New XQuery Open Save Close Run Check
App: unknown File Type: XML
new-document 1* hamlet.xml
/db/apps/exist101/data/hamlet.xml
XML Output Live Preview
Filter by...

```
<PLAY>
    <TITLE>The Tragedy of Hamlet, Prince of Denmark</TITLE>
    <FM>
        <P>ASCII text placed in the public domain by Moby Lexical Tools, 1992.</P>
        <P>SGML markup by Jon Bosak, 1992-1994.</P>
        <P>XML version by Jon Bosak, 1996-1999.</P>
        <P>The XML markup in this version is Copyright © 1999 Jon Bosak.
This work may freely be distributed on condition that it not be
modified or altered in any way.</P>
    </FM>
    <PERSONAE>
        <TITLE>Dramatis Personae</TITLE>
        <PERSONA>CLAUDIUS, king of Denmark. </PERSONA>
        <PERSONA>HAMLET, son to the late, and nephew to the present king.</PERSONA>
        <PERSONA>POLONIUS, lord chamberlain. </PERSONA>
        <PERSONA>HORATIO, friend to Hamlet.</PERSONA>
        <PERSONA>LAERTES, son to Polonius.</PERSONA>
        <PERSONA>LUCIANUS, nephew to the king.</PERSONA>
        <PGROUP>
            <PERSONA>VOLTIMAND</PERSONA>
            <PERSONA>CORNELIUS</PERSONA>
            <PERSONA>ROSENCRANTZ</PERSONA>
            <PERSONA>GUILDENSTERN</PERSONA>
            <PERSONA>OSRIC</PERSONA>
            <GRPDESCR>courtiers.</GRPDESCR>
        </PGROUP>
        <PERSONA>A Gentleman</PERSONA>
        <PERSONA>A Priest. </PERSONA>
        <PGROUP>
            <PERSONA>MARCELLUS</PERSONA>
            <PERSONA>BERNARDO</PERSONA>
            <GRPDESCR>officers.</GRPDESCR>
```

Figure 3-3. Hamlet opened in eXide

Alternatively, you can view the XML file in a browser through eXist's REST interface (see "Querying the Database Using REST" on page 94). Simply visit *http://localhost:8080/exist/rest/db/apps/exist101/data/hamlet.xml*.

But wait, it's not working! You should get an error message about a *shakes.xsl* stylesheet not being found. Crime doesn't pay: it's our punishment for being a data thief! The problem is that the XML files we started with were part of an application and were coupled to an XSL stylesheet by a processing instruction. This stylesheet, meant to create an HTML version of the play, was not copied by us and therefore, alas, could not be found. There's more about using XSLT via processing instructions in "Invoking XSLT by Processing Instruction" on page 242.

Fear not; the problem is easily solved. Open *hamlet.xml* (again) in eXide and look at the first line. It begins with:

```
<?xml-stylesheet href="shakes.xsl" type="text/xsl"?>
```

Remove this processing instruction using eXide, save the file, and try the URL again. You should now see *hamlet.xml* in all its XML glory.

If you feel bold enough, you could also try this: do not remove the processing instruction (or use one of the other files) and copy the *shakes.xsl* file from */db/apps/demo/data* to */db/apps/exist101/data*, like we did with the plays. When you now open the document in your browser (with the URL given before), you'll see a nicely rendered HTML page.

Listing the Plays (XML)

Let's write our first XQuery script and have it find out which plays we have in the */db/apps/exist101/data* collection. For now we'll return the result as an XML fragment, and in the next section we'll create a nice-looking HTML page from this. There is more than one way to do this (where have I heard that line before?), and we'll show you a few.

First, perform the following preparations:

1. Browse to the dashboard (*http://localhost:8080/exist/*).
2. If you're not already logged in, log in as `admin` (click on "Not logged in" in the upper-left corner).
3. Click on the "eXide - XQuery IDE" tile. eXide will open in a new browser window or tab.
4. Click on New XQuery. A new tab opens with an empty XQuery script.
5. Click Save and save the script as */db/apps/exist101/playlist.xq*.

Listing with the collection Function

In our first version of the solution, we will use the XPath `collection` function to iterate over all the resources in our *data* collection. Enter the following code and press Run:

```
for $resource in collection("/db/apps/exist101/data")
return
  base-uri($resource)
```

The `collection` function returns a sequence of documents (`document-node` items) for all the resources found in the given collection and its subcollections (for details, see "The collection Function" on page 93). The XPath `base-uri` function returns the URI for a `document-node`, which in eXist is the path leading to the resource.

Running this should return (in the bottom window) a list with the plays, including each one's full path in the database.

Turning this list of strings into a well-formed XML fragment is simple:

```
<plays>
{
  for $resource in collection("/db/apps/exist101/data")
    return <play uri="{base-uri($resource)}"/>
}
</plays>
```

This should return:

```
<plays>
  <play uri="/db/apps/exist101/data/hamlet.xml"/>
  <play uri="/db/apps/exist101/data/macbeth.xml"/>
  <play uri="/db/apps/exist101/data/r%20and%20j.xml"/>
</plays>
```

Now let's tweak this a little more. Later on we're going to present a list of available plays to the user, and it would be nice if we could display the name of the document (without a collection path) and, more importantly, the name of the play.

To get the name of the file, we have to do some string munging on the URI we already have. For this, XPath's regular expressions come in handy. Regular expressions are expressions to parse and work with strings. We won't explain them here because that would take too much ink (in fact, there are whole books devoted to them). For now, just accept that the following expression returns the part of a filename after the last / character:

```
replace(base-uri($resource), '.+/(.+)$', '$1')
```

There is one more thing about the name: we don't want the URI encodings, like *%20*, to show up in them. For this, we use a native eXist function from one of its extension modules (extension modules are covered in Chapter 7): `util:unescape-uri`. It needs two parameters: the name to unescape and the character encoding, which is nowadays almost always `UTF-8`. With this, the full code to get a nicely formatted filename becomes:

```
util:unescape-uri(
  replace(base-uri($resource), '.+/(.+)$', '$1'),
  'UTF-8'
)
```

To get the name of the play, we have to dive into the XML of the play itself. It's always in the `/PLAY/TITLE` element, and since we already have the root `document-node`, retrieving this is a piece of cake:

```
$resource/PLAY/TITLE/text()
```

Using the `/text` function will ensure we get the result as a text node. If we didn't use this, the `TITLE` element itself would be returned (try it out if you want to).

With all this, our full code becomes what you see in Example 3-1.

Example 3-1. The full code to get the play information

```
xquery version "3.0";
<plays>
{
    for $resource in collection("/db/apps/exist101/data")
    return
        <play uri="{base-uri($resource)}"
          name="{util:unescape-uri(replace(base-uri($resource),
            ".+/(.+)$", "$1"), "UTF-8")}">
        {
            $resource/PLAY/TITLE/text()
        }
        </play>
}
</plays>
```

And, with some slight improvements (using variables to store repeating data), it looks like Example 3-2.

Example 3-2. Improved version of the code that gets the play information

```
xquery version "3.0";
<plays>
{
    let $data-collection := "/db/apps/exist101/data"
    for $resource in collection($data-collection)
    let $uri := base-uri($resource)
    return
        <play uri="{$uri}"
          name="{util:unescape-uri(replace($uri, ".+/(.+)$",
            "$1"), "UTF-8")}">
        {
           $resource/PLAY/TITLE/text()
        }
        </play>
}
</plays>
```

Don't forget to save your code. After that, you can also try it from the browser. Try the URL *http://localhost:8080/exist/rest/db/apps/exist101/playlist.xq*. You should see the following:

```
<plays>
  <play uri="/db/apps/exist101/data/hamlet.xml" name="hamlet.xml">
    The Tragedy of Hamlet, Prince of Denmark</play>
  <play uri="/db/apps/exist101/data/macbeth.xml" name="macbeth.xml">
    The Tragedy of Macbeth</play>
  <play uri="/db/apps/exist101/data/r%20and%20j.xml" name="r and j.xml">
    The Tragedy of Romeo and Juliet</play>
</plays>
```

Listing with the xmldb Extension Module

Another way to return a list is by using one of eXist's extension modules, xmldb (see "Controlling the Database from Code" on page 107). An extension module contains functions that perform actions that are difficult or impossible to do in standard XQuery. eXist has quite a lot of them, and Appendix A lists the most important ones. If you want to explore the wonderful features that extension modules have to offer, you can access their documentation through the dashboard's XQuery Function Documentation tile.

Instead of the code entered in the previous section, try the following:

```
xmldb:get-child-resources("/db/apps/exist101/data")
```

Your result window should now show a list of the play's resource names (but without their paths). So, where the collection function returned a document-node sequence, xmldb:get-child-resources returns a string sequence. To get from a string to inside the XML (to get the name of the play), we have to resolve a document-node from this string. For this we use the XPath doc function (see "The doc Function" on page 94).

So, without further ado, Example 3-3 shows a piece of code that returns exactly the same results as the code in Example 3-2, but by a different means.

Example 3-3. Getting the play information using an extension function

```
xquery version "3.0";
<plays>
{
    let $data-collection := "/db/apps/exist101/data"
    for $resource-name in xmldb:get-child-resources($data-collection)
    let $uri := concat($data-collection, '/', $resource-name)
    return
        <play uri="{$uri}"
          name="{util:unescape-uri($resource-name, "UTF-8")}">
        {
            doc($uri)/PLAY/TITLE/text()
        }
        </play>
}
</plays>
```

Listing the Plays (HTML)

As a next step, let's present this information to the user as a nicely formatted HTML page. For this we have to (you might have guessed) create the HTML ourselves, including pieces like headers. For now we won't bother to make it look fancy by using CSS and the like, but of course you can add that too if you want.

First, let's set up a basic HTML (or more precisely, XHTML) page without much content to see how this works. Enter the code in Example 3-4 and save it as *plays-home.xq*.

Example 3-4. Basic HTML page code

```
xquery version "3.0";
<html>
    <head>
        <meta HTTP-EQUIV="Content-Type" content="text/html; charset=UTF-8"/>
        <title>Our Shakespeare plays</title>
    </head>
    <body>
        <h1>Our Shakespeare plays</h1>
    </body>
</html>
```

Now run it. In the bottom part of the eXide screen, you'll see the same unexciting piece of XHTML as you entered. Running it from the browser (*http://localhost:8080/exist/rest/db/apps/exist101/plays-home.xq*) doesn't make it look any better. So what's missing?

What's missing here is that you have to tell eXist that this is XHTML and that it should *serialize* it as such. There is more than one way to do this (see "Serialization" on page 118), but the easiest is to add an `exist:serialize` option in the XQuery prolog to tell eXist to serialize the query result as XHTML and send it to the browser as an HTML page and not as a bare piece of XML:

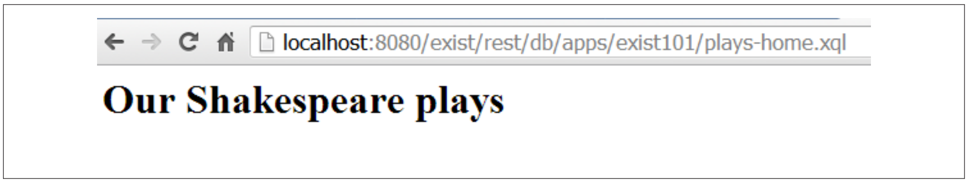
```
    declare option exist:serialize "method=xhtml media-type=text/html";
```

While we're changing things anyway, let's also get rid of the double entry for the page's title and put this in a global variable. Example 3-5 shows our improved basic HTML page.

Example 3-5. Improved version of the basic HTML page code

```
xquery version "3.0";

declare option exist:serialize "method=xhtml media-type=text/html";
declare variable $page-title := "Our Shakespeare plays";

<html>
    <head>
        <meta HTTP-EQUIV="Content-Type" content="text/html; charset=UTF-8"/>
        <title>{$page-title}</title>
    </head>
    <body>
        <h1>{$page-title}</h1>
    </body>
</html>
```

When you run this from within eXide nothing too exciting happens, but when you try it from the browser, you see the result shown in Figure 3-4.


localhost:8080/exist/rest/db/apps/exist101/plays-home.xql

Our Shakespeare plays

Figure 3-4. The exciting output of our first basic HTML page

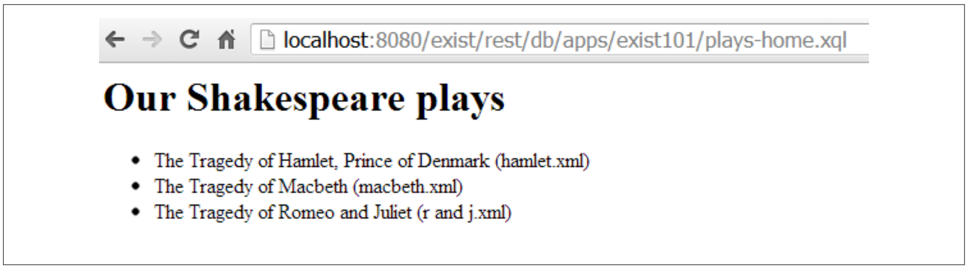
OK, now we're getting somewhere! Now let's use the output from "Listing the Plays (XML)" on page 45 to display a list of plays (we'll use the code with the `collection` function here, but the code that uses the `xmldb:get-child-resources` function is also fine). With a little copy and pasting, the code in Example 3-6 is constructed quickly.

Example 3-6. Code for the HTML page that returns a list of plays (plays-home.xq)

```
xquery version "3.0";

declare option exist:serialize "method=xhtml media-type=text/html";
declare variable $page-title := "Our Shakespeare plays";

let $play-info :=
    <plays>
    {
        for $resource in collection('/db/apps/exist101/data')
        return
            <play uri="{base-uri($resource)}"
                name="{util:unescape-uri(replace(base-uri($resource),
                ".+/(.+)$", "$1"), "UTF-8")}">
            {
                $resource/PLAY/TITLE/text()
            }
            </play>
    }
    </plays>
return

    <html>
        <head>
            <meta HTTP-EQUIV="Content-Type" content="text/html; charset=UTF-8"/>
            <title>{$page-title}</title>
        </head>
        <body>
```

```
            <h1>{$page-title}</h1>
            <ul>
            {
                for $play in $play-info/play
                return
                <li>{string($play)} ({string($play/@name)})</li>
            }
            </ul>
        </body>
    </html>
```

And in the browser, you should see the output in Figure 3-5.


localhost:8080/exist/rest/db/apps/exist101/plays-home.xql

Our Shakespeare plays

- The Tragedy of Hamlet, Prince of Denmark (hamlet.xml)
- The Tragedy of Macbeth (macbeth.xml)
- The Tragedy of Romeo and Juliet (r and j.xml)

Figure 3-5. Our HTML page with a list of plays

Those new to XQuery might wonder why on earth our HTML code (the part that starts with `<html>`) is now suddenly underneath a `return` statement. This is because we introduced a local variable (`let $play-info :=` …). As soon as you do this, your code becomes an XQuery FLWOR expression and needs a `return` statement for the part you want to return.

Analyzing the Plays

Now assume we're a famous play director and in need of some data about the play. Among the many, many questions directors struggle with are "Which character has the most lines?" and "How many actors do I need?" Let's find out the answers using XQuery.

Assume we have the `document-node` for the play's XML to analyze in a variable called `$play-document`. Subproblems to solve for our analysis are:

- We need a list of characters who speak. If you look inside the play's XML, you'll see that everything said is inside a `SPEECH` element with a `SPEAKER` subelement. So, to get a list of different speakers, all you have to do is get all the different values of `SPEECH/SPEAKER` and filter them for uniqueness:

  ```
  distinct-values($play-document//SPEECH/SPEAKER)
  ```

- To show totals and compute percentages for each speaker, we need the full set of spoken lines in the play:

  ```
  let $all-lines := $play-document//SPEECH/LINE
  ```

- We need the lines spoken by each speaker. Assuming the name of the speaker is in `$speaker`, we can get these with:

  ```
  $speaker-lines := $play-document//SPEECH[SPEAKER eq $speaker]/LINE
  ```

- Given a sequence of `LINE` elements in a variable `$elms` (the sequence of `LINE` elements we retrieved in one of the previous two bullets), how many words are spoken? A rough but usable approximation for this can be calculated by tokenizing everything said using whitespace boundaries and counigt/aggregating the results:

  ```
  sum($elms ! count(tokenize(., '\s+')))
  ```

  This might need a little further explanation:

  — The exclamation mark after the `$elms` expression is an XQuery 3.0 "bang" or "simple map" operator (see "The simple map operator" on page 114). It performs the operation on the right for all members of the sequence on the left.

  You could have done this in several other (and probably more customary) ways (e.g., using a FLWOR expression), but this seemed a useful way to introduce one of the new XQuery 3.0 capabilities.

  — The `tokenize` function tokenizes (breaks up) strings on boundaries given by a regular expression. The regular expression `'\s+'` signifies a sequence of whitespace characters, so that gives us the words (more or less, sometimes punctuation gets in the way, but let's forget about that for now).

  — We're not interested in the words themselves but only in how many there are. So we simply `count` them.

  — The outer `sum` function sums the numeric results of what's inside, returning the total of all words spoken in the elements in `$elms`.

As a last step, let's put this functionality into a local function (because we're going to use it twice in our code: once for the full play and once for every speaker):

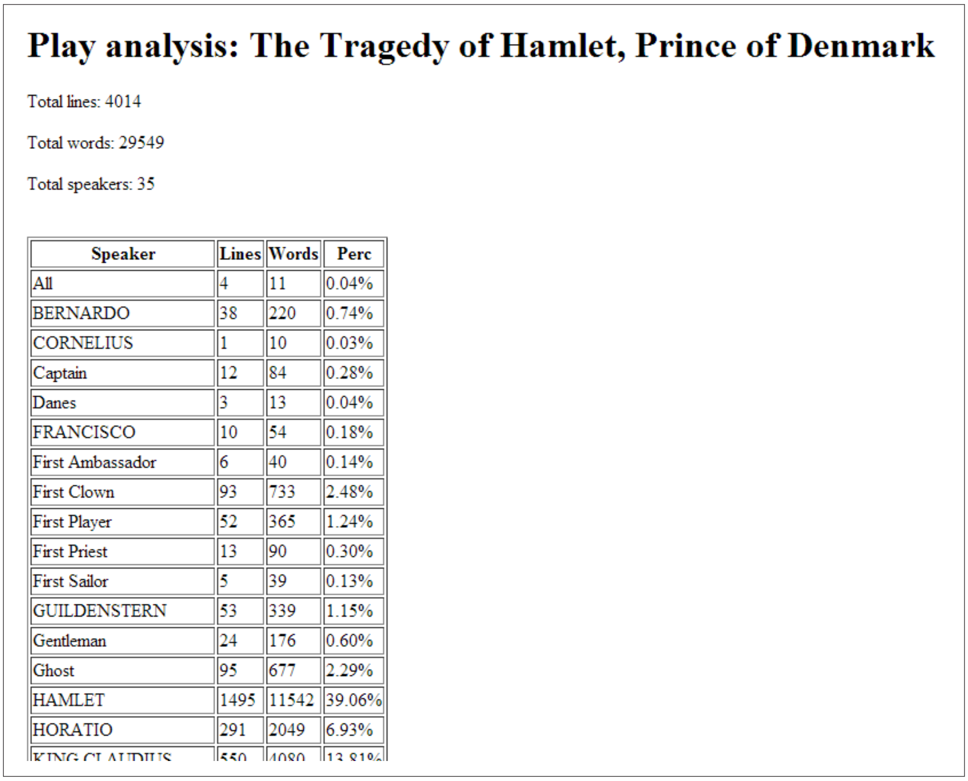
```
declare function local:word-count($elms as element()*) as xs:integer
{
  sum($elms ! count(tokenize(., '\s+')))
};
```

Now let's put this all together and create a page that shows us the results of our analysis (just for *Hamlet*) in a table. The code for this is shown in Example 3-7.

Example 3-7. Code to analyze Hamlet

```
xquery version "3.0";

declare option exist:serialize "method=xhtml media-type=text/html";
declare variable $page-title as xs:string := "Play analysis";
declare variable $play-uri as xs:string := "/db/apps/exist101/data/hamlet.xml";

declare function local:word-count($elms as element()*) as xs:integer
{
    sum($elms ! count(tokenize(., "\W+")))
};

let $play-document := doc($play-uri)
let $play-title := string($play-document/PLAY/TITLE)
let $speakers := distinct-values($play-document//SPEECH/SPEAKER)
let $all-lines := $play-document//SPEECH/LINE
let $all-word-count := local:word-count($all-lines)
return
    <html>
        <head>
            <meta HTTP-EQUIV="Content-Type" content="text/html; charset=UTF-8"/>
            <title>{$page-title}</title>
        </head>
        <body>
            <h1>{$page-title}: {$play-title}</h1>
            <p>Total lines: {count($all-lines)}</p>
            <p>Total words: {$all-word-count}</p>
            <p>Total speakers: {count($speakers)}</p>
            <br/>
            <table border="1">
                <tr>
                    <th>Speaker</th>
                    <th>Lines</th>
                    <th>Words</th>
                    <th>Perc</th>
                </tr>
                {
                    for $speaker in $speakers
                    let $speaker-lines :=
                      $play-document//SPEECH[SPEAKER eq $speaker]/LINE
                    let $speaker-word-count := local:word-count($speaker-lines)
                    let $speaker-word-perc :=
                      ($speaker-word-count div $all-word-count) * 100
                    order by $speaker
                    return
                        <tr>
                            <td>{$speaker}</td>
                            <td>{count($speaker-lines)}</td>
                            <td>{$speaker-word-count}</td>
                            <td>{format-number($speaker-word-perc, "0.00")}%</td>
                        </tr>
                }
            </table>
        </body>
    </html>
```

Save this code as *analyze-play.xq*. In the browser (*http://localhost:8080/exist/rest/db/apps/exist101/analyze-play.xq*), the result looks like Figure 3-6.



Play analysis: The Tragedy of Hamlet, Prince of Denmark

Total lines: 4014

Total words: 29549

Total speakers: 35

Speaker	Lines	Words	Perc
All	4	11	0.04%
BERNARDO	38	220	0.74%
CORNELIUS	1	10	0.03%
Captain	12	84	0.28%
Danes	3	13	0.04%
FRANCISCO	10	54	0.18%
First Ambassador	6	40	0.14%
First Clown	93	733	2.48%
First Player	52	365	1.24%
First Priest	13	90	0.30%
First Sailor	5	39	0.13%
GUILDENSTERN	53	339	1.15%
Gentleman	24	176	0.60%
Ghost	95	677	2.29%
HAMLET	1495	11542	39.06%
HORATIO	291	2049	6.93%
KING CLAUDIUS	550	4080	13.81%

Figure 3-6. Partial output of the analysis for Hamlet

Well, that wasn't too hard, was it? The analysis itself was almost laughably easy; most of the code is actually dedicated to creating a nicely formatted page.

As you might have noticed, we used a different approach here than in Example 3-6. There, we first created an XML fragment and later used it to create the HTML. In the analysis page, we put the FLWOR expressions directly inside the HTML. This was done intentionally, to show you two different approaches.

By the way: maybe you shouldn't go for the role of Hamlet—almost 1,500 lines with 11,500 words is a lot to learn by heart. And have mercy on the director: finding and directing 35 actors is not exactly a walk in the park!

Linking the Analysis to the Play Overview

At this moment our application still consists of two different pages: an overview and an analysis page for *Hamlet*. So how can we tie these together and use the analysis page for every play?

To do this, we first need to link from our overview to our analysis page and pass the URI of the play to analyze in the link. Go back to our basic home page (Example 3-6) and change the line starting with `<li>` as follows:

```
<li>{string($play)} ({string($play/@name)})
  <a href="analyze-play.xq?uri={encode-for-uri($play/@uri)}">analysis</a>
</li>
```

The `a` element (that produces an HTML link) links to the analysis page and passes the URI of the play in the `uri` parameter. The `encode-for-uri` function is necessary here because the URI passed contains characters that are misinterpreted if we pass them straight: `encode-for-uri` creates `%` encodings for them, e.g., `%2F` for a slash (`/`) character.

An interesting phenomenon occurs when you use this technique for *Romeo and Juliet*. Remember, the URI for this play already contains URI-escaped characters (the spaces in the resource name: *r%20and%20j.xml*). Passing this URI through the `encode-for-uri` function means the URI will become double URI encoded! The resource name is passed as *r%2520and%2520j.xml*.

The second thing we have to do is change our analysis page to retrieve the URI in the `uri` parameter and act on this. For this, change the declaration of the `$play-uri` variable to:

```
declare variable $play-uri as xs:string := request:get-parameter('uri', ());
```

`request:get-parameter` is an extension function from the request extension module (see "The request Extension Module" on page 209) that returns the value of a parameter passed in the URL (or of a control in an HTML form–driven request). The second parameter, which in this example is the empty sequence `()`, can be used to pass a default value.

After making these changes you can analyze not only *Hamlet*, but all three of the plays (and assuming you can provide the data, then many more). Neat, isn't it?

Searching the Plays

The next two enhancements add search functionality to our little play application. One uses straight XQuery (actually as an example of how *not* to do it), and the other uses eXist's full-text indexing capabilities.

Searching Using Straight XQuery

Let's assume we need to search the plays for certain words or phrases. A first naive approach could be to use just straight XQuery. The searching is not hard at all. For instance, try the following surprisingly short code snippet, which searches *all* the plays for the word *fantasy* (and adds a `play` attribute to tell us which play the results came from):

```
for $line in
  collection('/db/apps/exist101/data')//SPEECH/LINE[contains(., 'fantasy')]
return
  <LINE play="{base-uri($line)}">{string($line)}</LINE>
```

This shows that *Hamlet* is three times as fantastic as *Romeo and Juliet*, and there's no fantasy in *Macbeth* (for the Shakespearians: that was a joke):

```
<LINE play="/db/apps/exist101/data/hamlet.xml">
  Horatio says 'tis but our fantasy,</LINE>
<LINE play="/db/apps/exist101/data/hamlet.xml">
  Is not this something more than fantasy?</LINE>
<LINE play="/db/apps/exist101/data/hamlet.xml">
  That, for a fantasy and trick of fame,</LINE>
<LINE play="/db/apps/exist101/data/r%20and%20j.xml">
  Begot of nothing but vain fantasy,</LINE>
```

Although the searching itself is simple to program, we do need some code to tie it all together. First, add a search form to the main page. Reopen *plays-home.xq* (Example 3-6) and add the following lines between the `</ul>` and `</body>` closing elements:

```
<br/>
<h3>Search using XQuery:</h3>
<form method="POST" action="search-1.xq">
  <input type="text" name="searchphrase" size="40"/>
  <input type="submit" value="Search!"/>
</form>
```

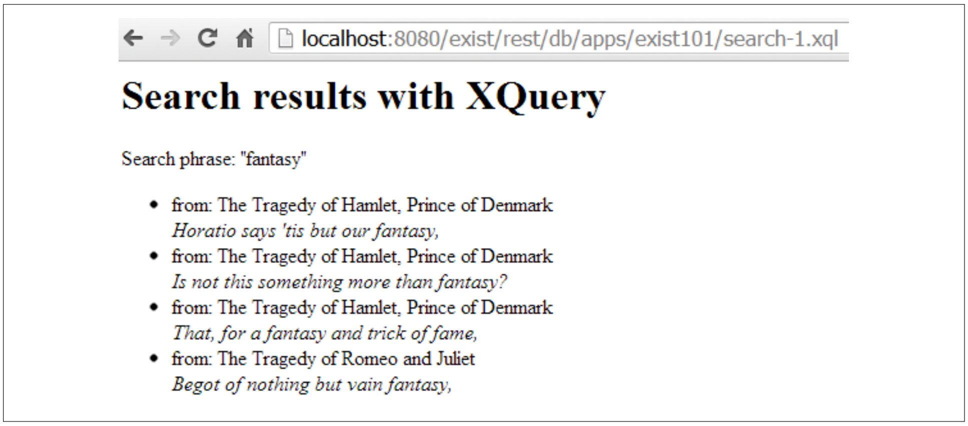
When you reopen the page in your browser (*http://localhost:8080/exist/rest/db/apps/exist101/plays-home.xq*) you'll see a nice little search form with a Search! button.

Now we have to create an XQuery script that does something with the search request and displays the results. Open a new XQuery file in eXide, save it as *search-1.xq*, and enter the code shown in Example 3-8.

Example 3-8. Search page that uses straight XQuery

```
xquery version "3.0";

declare option exist:serialize "method=xhtml media-type=text/html";
declare variable $page-title := "Search results with XQuery";
declare variable $searchphrase := request:get-parameter("searchphrase", ());

<html>
    <head>
        <meta HTTP-EQUIV="Content-Type" content="text/html; charset=UTF-8"/>
        <title>{$page-title}</title>
    </head>
    <body>
        <h1>{$page-title}</h1>
        <p>Search phrase: "{$searchphrase}"</p>
        <ul>
        {
            for $line in collection("/db/apps/exist101/data")//SPEECH/LINE
              [contains(., $searchphrase)]
            return
                <li>
                    from: {string(root($line)/PLAY/TITLE)}<br/>
                    <i>{string($line)}</i>
                </li>
        }
        </ul>
    </body>
</html>
```

When you now enter *fantasy* as a search phrase and press Search!, the result shown in Figure 3-7 should appear.


localhost:8080/exist/rest/db/apps/exist101/search-1.xql

Search results with XQuery

Search phrase: "fantasy"

- from: The Tragedy of Hamlet, Prince of Denmark
  Horatio says 'tis but our fantasy,
- from: The Tragedy of Hamlet, Prince of Denmark
  Is not this something more than fantasy?
- from: The Tragedy of Hamlet, Prince of Denmark
  That, for a fantasy and trick of fame,
- from: The Tragedy of Romeo and Juliet
  Begot of nothing but vain fantasy,

Figure 3-7. Results of searching the plays for "fantasy"

Well, that's not a bad result for a fairly minimal amount of code. But, we can do better:

- This solution doesn't scale well. Searching like this works by comparing the query string with each word in the text: all `LINE` elements are string-searched for the search phrase. When you get more and more data, the searches will become slower and slower.
- It searches for literal strings, which means it's case-sensitive and will also return results where the search phrase is part of a word (e.g., searching for *faun* will also return lines with *fauna*).
- It cannot handle search *expressions* (e.g., searching for lines with *fantasy* and *Macbeth*).

So, let's add a second search facility with some enhancements.

Searching Using an Index

An index in a database is comparable to the index in a book: it allows you to quickly find something based on an index *key*. Although useful and often even necessary, using indexes is not without disadvantages: it slows down creating and updating data and consumes (a little bit of) disk space. However, indexes make searching *much* faster, especially for large collections of documents.

eXist supports several types of indexes, which we will elaborate on in Chapters Chapter 11 and Chapter 12. In this example we're going to use a full-text index.

To illustrate things, let's first add all the code before creating the index. Again, we have to change *plays_home.xq* and add a new search form. Reopen *plays-home.xq* (Example 3-6) and add the following lines right before the closing `</body>` element:

```
<br/>
<h3>Search using index:</h3>
<form method="POST" action="search-2.xq">
  <input type="text" name="searchphrase" size="40"/>
  <input type="submit" value="Search!"/>
</form>
```

Now create *search-2.xq*, as shown in Example 3-9. It's almost the same as *search-1.xq* (Example 3-8), so copy, paste, and fiddle is probably a good option here.

Example 3-9. Search page that uses indexed search

```
xquery version "3.0";

declare option exist:serialize "method=xhtml media-type=text/html";
declare variable $page-title := "Search results with XQuery using full-text index";
declare variable $searchphrase := request:get-parameter("searchphrase", ());
```

```
<html>
    <head>
        <meta HTTP-EQUIV="Content-Type" content="text/html; charset=UTF-8"/>
        <title>{$page-title}</title>
    </head>
    <body>
        <h1>{$page-title}</h1>
        <p>Search phrase: "{$searchphrase}"</p>
        <ul>
        {
            for $line in collection("/db/apps/exist101/data")//SPEECH/LINE
              [ft:query(., $searchphrase)]
            return
                <li>
                    from: {string(root($line)/PLAY/TITLE)}<br/>
                    <i>{string($line)}</i>
                </li>
        }
        </ul>
    </body>
</html>
```

Its main difference from *search-1.xq* is the way the search expression is formulated:

```
    collection('/db/apps/exist101/data')//SPEECH/LINE[ft:query(., $searchphrase)]
```

The `ft:query` function looks for an index on the elements in its first argument. However, since this index is not yet there, the result set will currently still be empty (which we can easily prove by searching on *fantasy* using the indexed search). So, let's define this index.

Index definitions (and some other content) are kept in what is best described as a "shadow" database collection structure underneath */db/system/config*. If you go there (using, for instance, the collection browser of the dashboard), you'll see parts of the main database collection structure there too, starting with */db*.

To add our play index definition, create the collection */db/system/config/db/apps/exist101/data*. In this collection create an XML document called *collection.xconf* with the contents shown in Example 3-10.

Example 3-10. The index definition collection.xconf document

```
<collection xmlns="http://exist-db.org/collection-config/1.0">
  <index xmlns:tei="http://www.tei-c.org/ns/1.0">
    <lucene>
      <text qname="LINE"/>
    </lucene>
  </index>
</collection>
```

We're almost done. After defining a new collection, you'll have to reindex. Use the collection browser to go to */db/apps/exist101/data* and click "Reindex collection" (the second button from the left). This will take a few seconds. After this initial reindex you'll never have to do any manual maintenance again; eXist now knows the index and will keep it up to date for you when the dataset changes.

Our indexed search is ready to go. Try it by searching on *fantasy* again, and you should now see the same results as in our straight XQuery search (Figure 3-7).

So, have we gained anything? Yes, we have!

- Search on *FANTASY*: the search is now case-insensitive.
- Search on *fantasy horatio*. You'll get a long list with lines in which one or both of the two words are present.
- Search on *fantasy AND horatio*. Now there is only one line: the only one with both words.
- But perhaps most importantly: this search is faster (although you might not notice on modern hardware with this small dataset), and it scales! Add more plays, and it will stay fast.

This was the just tip of the iceberg in terms of what you can do with indexes. For example, in "Full-Text Index and KWIC Example" on page 285, you'll learn about sorting by relevance and how to use eXist's "keywords in context" feature to highlight the matching words in the search results. Adding such capabilities to our eXist 101 application is not difficult once you have progressed further through the book, and we'll leave this to you as an independent exercise.

Creating a Log

The last capability we're going to add to our application is having it maintain a logfile, showing you how you can update the database from XQuery. Since we'll probably want to use this feature from more than one XQuery script, we're going to put the code for it into an XQuery *module* and show you a little about modularization. While testing, we'll also run into some interesting security problems and show you how to solve them.

By way of preparation, create the collection */db/apps/exist101/log* to hold our log information.

We require our logging to write strings to a logfile and add a timestamp. If the logfile does not exist, it should be created. The logfile should be easy to use and callable from all our existing XQuery scripts without code duplication.

This functionality calls for a design that uses an XQuery module. The module should contain an XQuery function that performs the logging. To make it easy to use, this function must return zilch (a.k.a. the empty sequence in XPath lingo), so we can simply call it from anywhere without influencing our output. Example 3-11 is a module that does exactly this. Create this module and save it as */db/apps/exists101/log-module.xqm*.

Example 3-11. The logging module log-module.xqm

```
xquery version "3.0" encoding "UTF-8";

module namespace x101log = "http://www.exist-db.org/book/namespaces/exist101"; ❶

declare function x101log:add-log-message($message as xs:string)
  as empty-sequence() ❷
{
    ❸ let $logfile-collection := "/db/apps/exist101/log"
    let $logfile-name := "exist101-log.xml"
    let $logfile-full := concat($logfile-collection, '/', $logfile-name)
    ❹ let $logfile-created :=
        if(doc-available($logfile-full))then
            $logfile-full
        else
            xmldb:store($logfile-collection, $logfile-name, <eXist101-Log/>)
    return ❺
        update insert
          <LogEntry timestamp="{current-dateTime()}">{$message}</LogEntry>
            into doc($logfile-full)/*
};
```

❶ The `module namespace` definition at the top defines the code as an XQuery module. A module must have a namespace, and it's customary to use something that starts with a URL you own (to avoid name clashes). What comes after that is up to you. Don't let the starting `http://` fool you: it's just a string without any further special meaning.

❷ We declare a function that returns `empty-sequence()`?. In other words: nothing.

❸❹❺ The first three `let` statements in the function simply define the logfile's location, name, and full name.

❻ `let $logfile-created := ...` checks whether or not the logfile exists. If not, it uses the `xmldb:store` extension function to create a new logfile with an empty `<eXist101-Log/>` root element. If the logfile already exists, it simply emulates the return value of `xmldb:store` by returning the logfile's name.

Have a look at the rest of the code. We never actually *use* this `$logfile-created` variable, so if eXist did lazy evaluation (computing values only when needed), the logfile would never get created. Lucky for us, eXist always evaluates variable assignments from top to bottom, which we exploit here to allow side effects. Read more about this in "XQuery Execution" on page 118.

❼ The `return` part of the function's FLWOR expression contains an eXist `update insert` statement that inserts a `LogEntry` element, with a timestamp and the given text, as a child of the root element of our logfile. An `update` statement always returns an empty sequence, so that takes care of our required empty return value. More about eXist XQuery update functionality can be found in "Updating Documents" on page 101.

Let's call this logging function from our home page. To do this, first add the following `import module` statement at the top of *plays-home.xq* (see Example 3-6), directly after the `xquery version "3.0";` declaration:

```
import module namespace x101log=
          "http://www.exist-db.org/book/namespaces/exist101"
          at "log-module.xqm";
```

Now *plays-home.xq* "knows" about the logging module. To call the logging function, add the following line directly before the `let $play-info :=`... part (don't forget the finishing comma!):

```
x101log:add-log-message('Visited plays-home'),
```

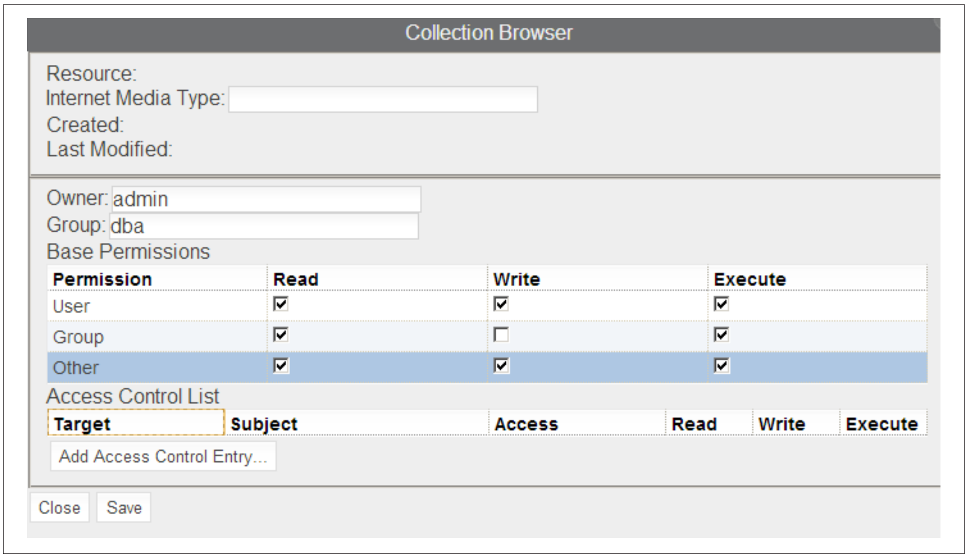
Let's test our changes. You might be logged in as `admin`, but to make a point we would like you to do this without being logged in. Please log out and close your browser, reopen it, and run *plays-home.xq* by visiting *http://localhost:8080/exist/rest/db/apps/exist101/plays-home.xq*. Oops—you get an error stating, "Write permission is not granted on the Collection." Why is that?

eXist has a strict security system. When you're logged in as `admin` you can do anything, but when you're not it's a different game. All access to collections and resources must abide by the strict eXist security rules, which look, not at all by accident, remarkably like those in a Unix environment.

To view the relevant security settings, open the collection browser in the dashboard and navigate to */db/apps/exist101*. The permissions on your *log* collection *(db/apps/exist101/log)* should read `crwxr-xr-x` (that is, if you haven't changed any defaults). You'll find a complete explanation of what this means in Chapter 8. For now, we'll focus on what's important for us.

When you visit eXist without being logged in, you're using a built-in account called `guest`. This is, as we can see, not the owner or the group the collection belongs to, so the relevant security settings are the last three characters of the permissions: `r-x`. This means anybody can list and open the collection, but not write to it. Let's change this so we can at least write to the collection and create our logfile.

Use the collection browser or the Java Admin Client to set the permissions to `crwxr-xrwx`—that is, set the Write permission for Other. Figure 3-8 shows how this looks in the collection browser.


Collection Browser

Resource:
Internet Media Type:
Created:
Last Modified:

Owner: admin
Group: dba
Base Permissions

Permission	Read	Write	Execute
User	✓	✓	✓
Group	✓	☐	✓
Other	✓	✓	✓

Access Control List

Target	Subject	Access	Read	Write	Execute

Figure 3-8. Changing the permissions of the log collection

After that, log out and close your browser (to stop being `admin`), open it again, and revisit the *plays-home.xq* page. This should now run smoothly. Look inside the *log* collection, and there it is: our *exist101-log.xml* document. It contains something like:

```
<eXist101-Log>
  <LogEntry timestamp="2013-04-09T20:51:27.205+02:00">Visited plays-home
    </LogEntry>
</eXist101-Log>
```

And yes, every time you revisit the *plays-home.xq* page, a new `LogEntry` element is added. I'll leave it up to you to add the logging code to the other pages of our small but beautiful application.

The `x` permission on a collection means something different than the `x` permission on a resource. For a resource it means *execute* rights, and this is important for XQuery scripts. Try this out by, for instance, revoking the `x` permissions on our *plays-home.xq* file. You should now get a security warning when you try to run it.

What's Next?

Although we could go on (and on and on), this is the end of our 101 course. We hope it gave you at least a taste of what's possible, and some pointers on how to start out with eXist. There is still much more to learn (that's why there are chapters after this one)! Here are some suggestions for further exploration:

- Explore eXist's extension modules' functionality using the information in Chapter 7 and through the dashboard's XQuery Function Documentation browser.
- Tighten security to a specific set of users with the information in Chapter 8.
- Change our little demo application into a real one using RESTXQ or URL rewriting. This will give you much more control over the URLs needed to visit the pages, security, error handling, and so on. More information can be found in Chapter 9.
- Explore other index types and settings, as described in Chapters Chapter 11 and Chapter 12.
- Use one of the other supported XML technologies from Chapter 10—for instance, creating a PDF version of our analysis report with XSL-FO.
- Integrate the application with the rest of the world using the technologies described in Chapter 13.

But most of all, remember to have fun!

CHAPTER 4

Architecture

In this chapter we look in detail at how eXist is constructed and how it processes your XML documents and executes your XQueries. This information should be considered advanced, so if you are a beginner you may want to skip to another chapter. However, for those wishing to master eXist, this information can be invaluable in helping you understand how to use it efficiently.

eXist is a large software project that has evolved over the last 13+ years, and is written predominantly in the Java programming language. Although extensions and add-ons to eXist are often written in pure XQuery and/or XSLT, the main body of eXist is written in Java.

Regardless of how you decide to deploy and use eXist, its architecture (see Figure 4-1) remains largely the same, with various optional components depending on your use.

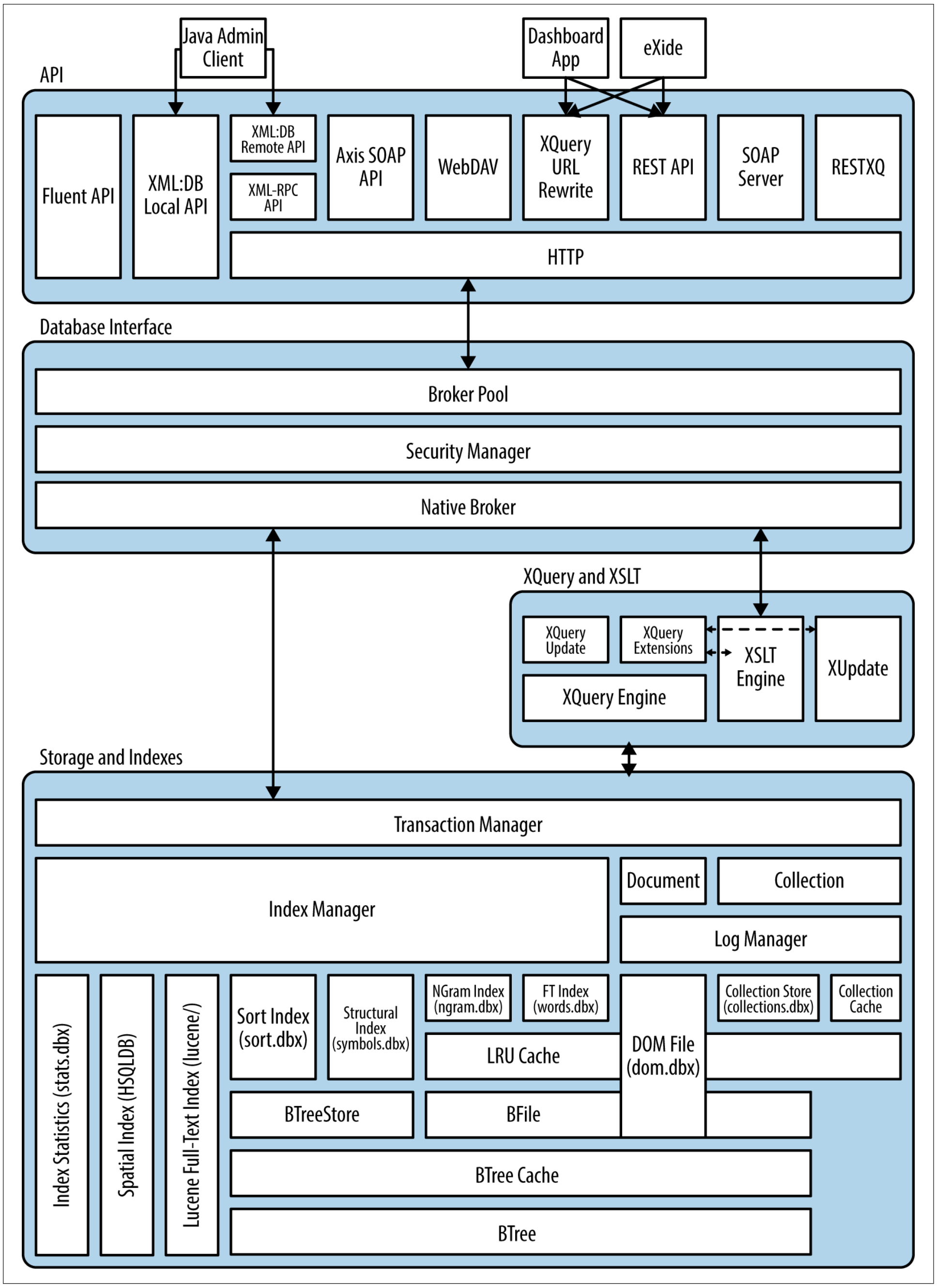


Figure 4-1. Complete high-level diagram of eXist's architecture

Each connection from an API to eXist is a single thread that interacts with the *broker pool*, which is configured with a number of *broker*s (20 by default). Each broker is a thread that interacts with the database, and represents a database request; this might be a database update operation (add/delete/store/update) or a query operation (XPath, XQuery, or XSLT). If you connect to eXist and all the brokers are busy, your request will pause until a broker becomes available to service your request. Figure 4-2 shows this broker architecture.

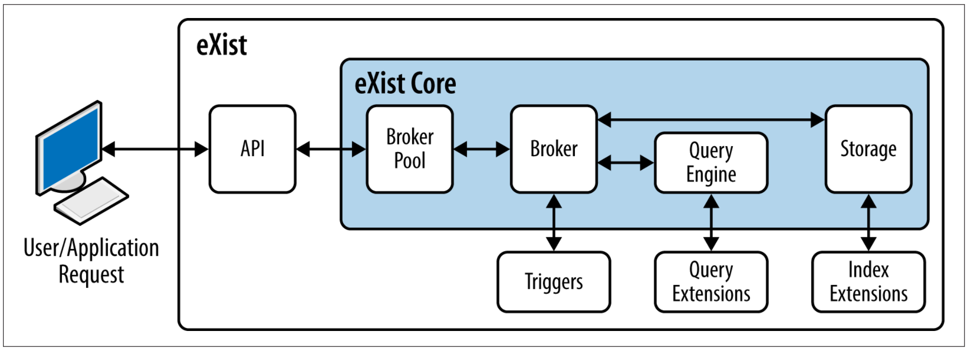


Figure 4-2. eXist's broker architecture

If you are using eXist directly embedded within your own application, then you are responsible for deciding the threading model for interacting with the broker pool.

The number of brokers available in the broker pool is configurable in *$EXIST_HOME/conf.xml*, in the attribute located by the XPath */exist/db-connection/pool/@max*. You should configure the number of brokers to be slightly higher than the expected number of *concurrent* connections to eXist from your users.

Deployment Architectures

eXist offers many options for deployment. These options have informed the architecture of eXist, and your deployment scenario will determine how much or how little of eXist you use and how the components of eXist are interconnected. eXist has three main faces—the first you may see only if your interaction with eXist is as a software developer, while the second two you may also see if you are an eXist user:

An embedded native XML database library
eXist can be compiled as a set of libraries that can be directly embedded into your own application running on a JVM (Java Virtual Machine). In this instance you talk directly to eXist via Java function calls. See "XML:DB Local API" on page 366 and "Fluent API" on page 369.

A native XML database server
: eXist can be deployed as a native XML database server, allowing you to store and manage both XML and binary documents and perform queries upon these documents across a network. This approach is similar to a classic client/server networked database architecture.

A web application platform
: Building upon the preceding item, eXist offers so many web-oriented features and capabilities that they deserve mentioning in their own right. You can build entire web applications by writing very simple and powerful high-level web application code in XQuery, XForms, and XSLT, and eXist will provide the HTTP glue to serve these up as complete web applications.

Embedded Architecture

If you choose to embed eXist directly in your own application (see "Local APIs" on page 364), eXist operates as any other third-party library would; that is, you make function calls in your application upon the classes that make up eXist in order to perform various operations. When eXist is embedded in your application, you have two local APIs to choose from—either the XML:DB Local API (see "XML:DB Local API" on page 366) or the Fluent API (see "Fluent API" on page 369).

It is also possible to directly call eXist's internal API functions; however, this is strongly discouraged, as you are then responsible for the correct locking and transaction management of resources, which is nontrivial to achieve correctly. In addition, these API functions are subject to change at any time.

Embedding eXist within your application (as shown in Figure 4-3) can be a suitable choice if you wish to release a convenient standalone application to your users that has advanced XML processing and query facilities.

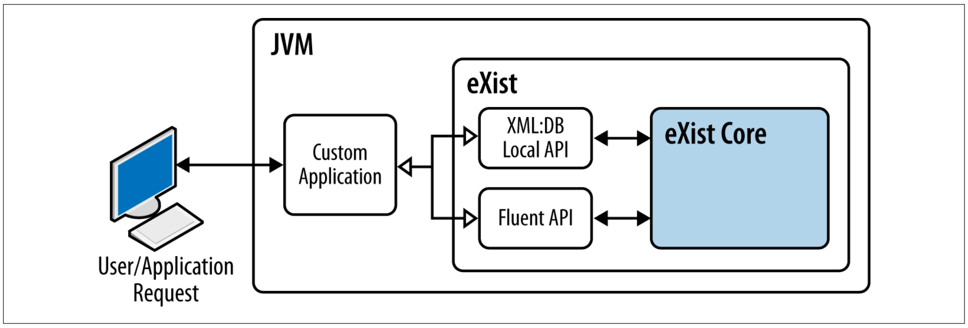


Figure 4-3. JVM-embedded eXist architecture

The downside to embedding eXist within your own application rather than connecting to it as a server is a lack of isolated concerns. When eXist is embedded in a developer application, it runs within the same JVM instance, and thus the memory and resources available to the JVM are shared between both eXist and the developer's application. Should a fault or design error exist within the developer's application, this could potentially easily exhaust the resources or memory available to the JVM instance. This results in a lockup of the application and unpredictable behavior that is beyond the control of eXist. Such unpredictable behavior is not desirable when you're operating a database system, as it can lead to stability issues and possibly data corruptions.

Client/Server Database Architecture

There is no fixed pattern for running eXist as a database server, as eXist offers many different options; however, for the purposes of illustration we will assume that the database server architecture is akin to a classic client/server networked database. In the client/server networked database, users typically connect to a database server with a fat client application and/or communicate with the database server from custom applications by opening connections to the database across the network. In this architecture we will exclude the more web-oriented aspects of eXist, as they are more applicable to the next section, "Web Application Platform Architecture" on page 70.

In the client/server database architecture, eXist is deployed to a central server and users are given access to the database by both the WebDAV and XML:DB Remote APIs (Figure 4-4). The WebDAV API allows users to directly manipulate documents in the database from their operating system's file explorer, just as if the files were in local folders on their computers (see "WebDAV" on page 305). The XML:DB Remote API, on the other hand, allows users to connect to eXist from the Java Admin Client GUI (see "XML:DB Remote API" on page 349), from which they can manage the database and perform queries upon it. If you enable the XML:DB Remote API, XML-RPC (see "XML-RPC API" on page 342) also has to be enabled as a prerequisite, so application developers are free to develop applications that can talk to the database programmatically using one or a combination of the WebDAV, XML:DB, or XML-RPC protocols.

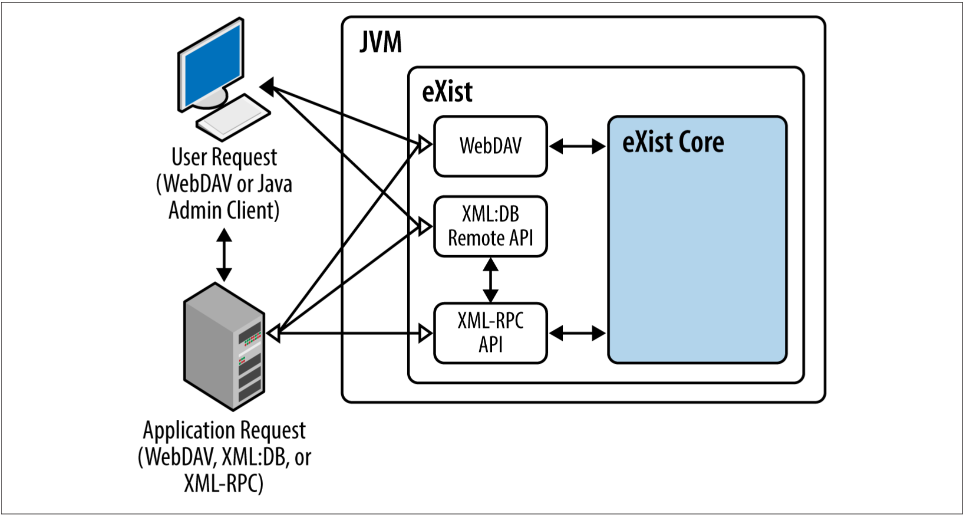

JVM
eXist
WebDAV
eXist Core
XML:DB Remote API
XML-RPC API
User Request (WebDAV or Java Admin Client)
Application Request (WebDAV, XML:DB, or XML-RPC)

Figure 4-4. eXist client/server database architecture

Web Application Platform Architecture

When developing a web application, you typically have two domains of code: *server-side* and *client-side*. Server-side code is executed on the web application server itself (in this case, eXist), while client-side code is executed by the client (typically a web browser).

eXist can serve as a complete web application platform whereby your applications are developed in one or more of the following server-side XML languages: XQuery, XSLT, XProc, and XForms.

Upon receiving a web request from a client, eXist processes your server-side code and generates a response containing the results of the processing (which may even include further client-side code) for the client. The response sent to the client could be anything, but typically will be in one or more of these formats: HTML5, XHTML, XML, JSON, XForms, CSS, or a variety of binary formats (e.g., images for display in a page).

To understand how to develop web applications with eXist, see Chapter 9. When eXist is deployed as a web application platform there are many components that you may or may not wish to use, depending on your web application; however, it is recommended (as described in "Reducing the Attack Surface" on page 177) that you disable those that you are not using. Regardless of your application, if you wish to use eXist as a web application platform, it will require the use of a *Java web application server*, and here you have two main choices for deployment:

Jetty™ (default)

eXist ships with the Jetty Java web application server. This can be used with very little effort and is the recommended approach, as it is well understood and supported by the eXist community and developers. See "Starting and Stopping eXist with a GUI" on page 23.

WAR file and third-party Java web application server

eXist can be built and deployed as a WAR file to your choice of Java web application server (e.g., Apache Tomcat, GlassFish, JBoss Application Server, etc.). This can work well if your organization is already heavily invested in a particular technology. However, it is recommended that you run eXist solely in its own web application server; otherwise, it will potentially be competing for resources with other applications in the same server. See "Building eXist from Source" on page 485.

Either way—whether you use the built-in Jetty or a third-party Java web application server—the web application platform architecture of eXist remains much the same (see Figure 4-5).

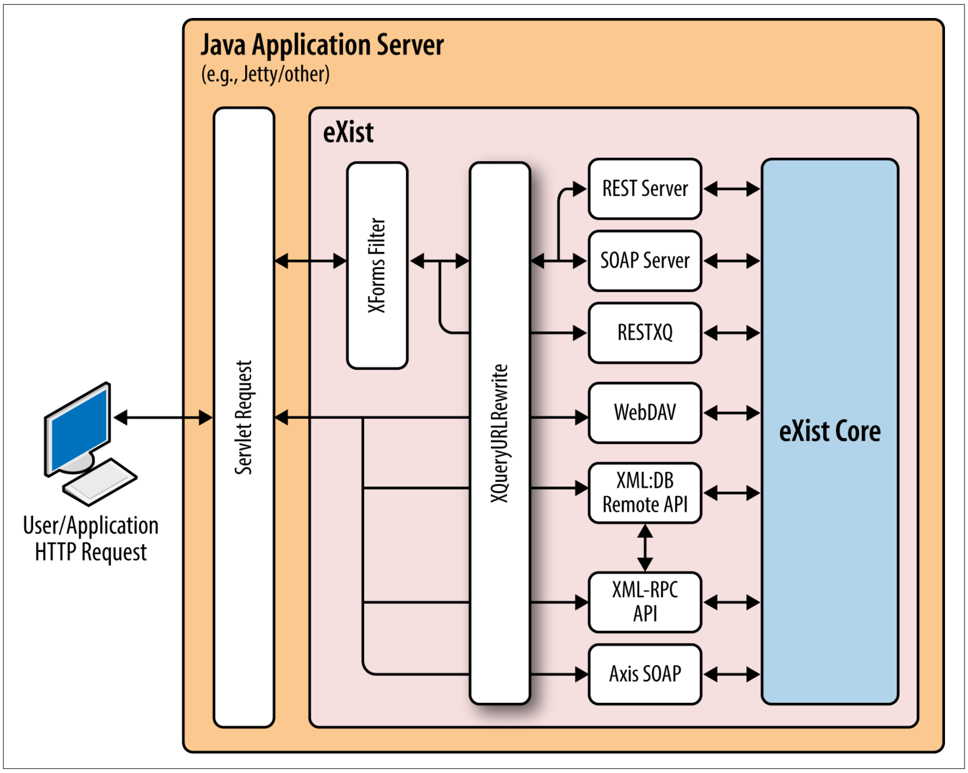


Figure 4-5. eXist web application platform architecture

Figure 4-5 should be interpreted as depicting that only the REST Server and SOAP Server services of eXist are directly manipulatable by the XQuery URL Rewrite facility (see "URL Mapping Using URL Rewriting" on page 194). However, it also shows that RESTXQ can be coupled with the XForms Filter; while this is indeed the case and may be desirable, it is not enabled by default in eXist (but may be configured easily).

Storage Architecture

So far we have mostly been looking at the high-level architecture of eXist depending on the type of application you wish to build with it. You should at least have a cursory understanding of how eXist structures its resources into collections from reading Chapter 3. We will now look much more closely at the storage architecture of eXist, as you may be wondering, "What happens when I actually store a document into eXist?"

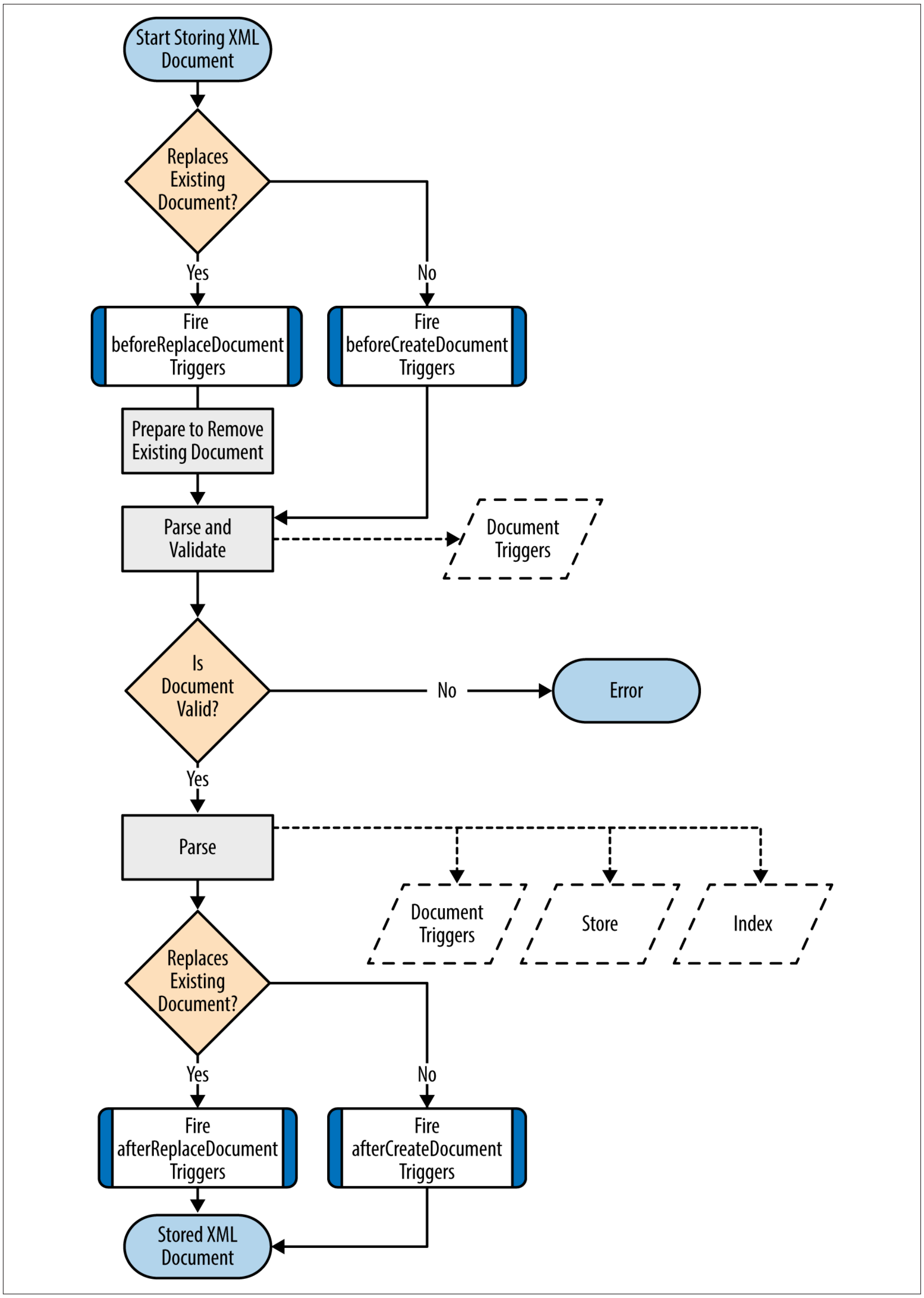
XML Document Storage and Indexing

When given an XML document, eXist first takes the document and *parses* it (while *validating* it if requested), and then extracts all of the information from the document and *stores* it while also *indexing* the document's key features (see Figure 4-6).

eXist does *not* store your XML documents as a series of XML files on disk, as this is not an efficient storage format for database operations.

eXist stores an XML document by taking the information making up the document and separating it into distinct parts that are then stored in a series of optimized binary files on disk. This process is transparent to users; they can always ask eXist for the XML document they originally stored, and the complete document will be reconstituted and presented to them as it was.

The indexes that eXist builds from your XML documents during storage enable you to later perform fast and efficient queries against the documents in the database. The structural index employed by eXist is fixed, but additional indexes are configurable by the users depending on their query requirements; see "Configuring Indexes" on page 275 for index configuration details.



Start Storing XML Document
Replaces Existing Document?
Yes
No
Fire beforeReplaceDocument Triggers
Fire beforeCreateDocument Triggers
Prepare to Remove Existing Document
Parse and Validate
Document Triggers
Is Document Valid?
No
Error
Yes
Parse
Document Triggers
Store
Index
Replaces Existing Document?
Yes
No
Fire afterReplaceDocument Triggers
Fire afterCreateDocument Triggers
Stored XML Document

Figure 4-6. XML storage high-level process

The following steps occur when eXist stores an XML document into a collection in the database (see Figure 4-7).

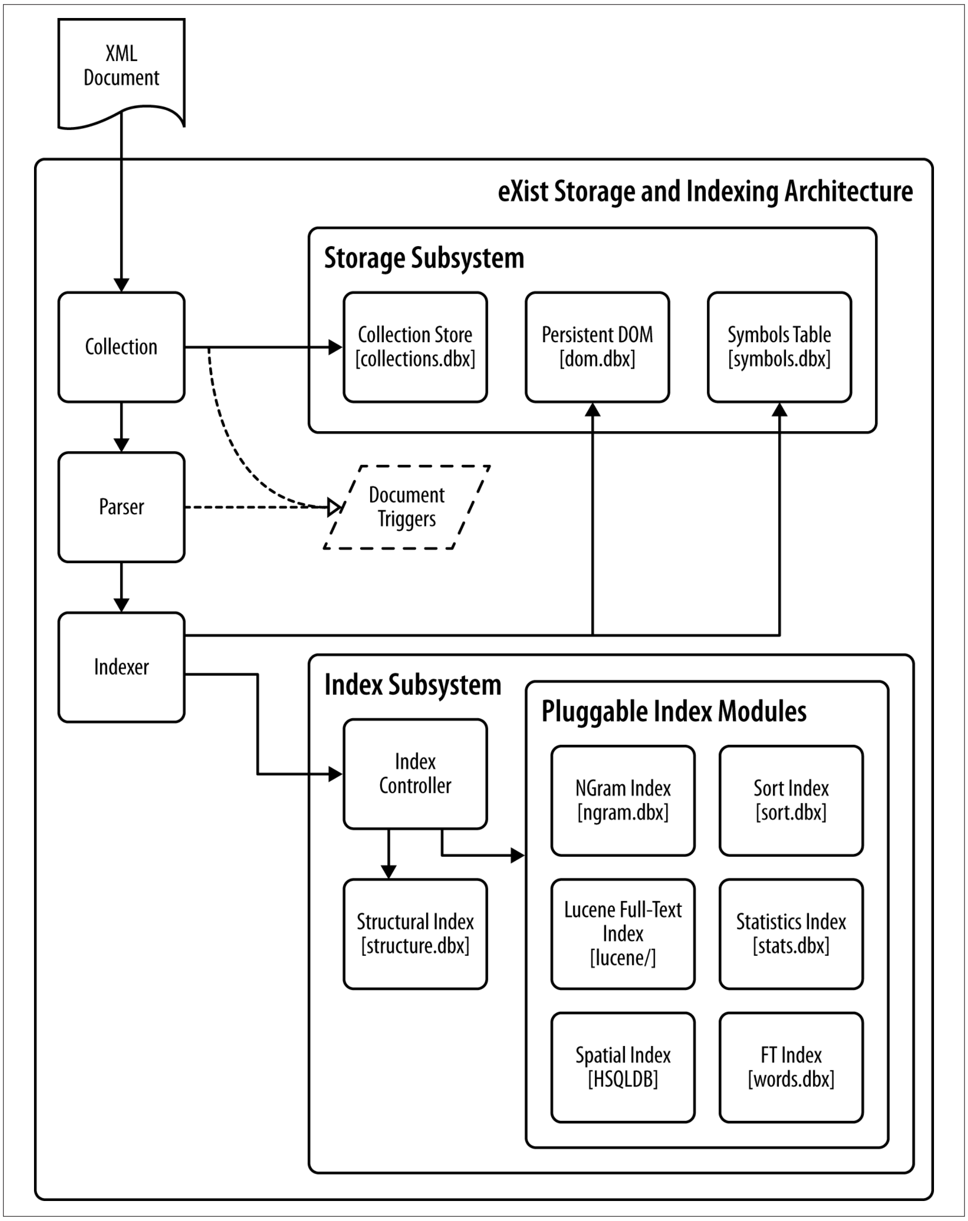


Figure 4-7. eXist XML storage and indexing architecture

1. Either a `beforeCreateDocument` or a `beforeUpdateDocument` event is raised with each *document trigger* configured on the target collection.
2. The validation phase begins. The parser begins parsing the XML document with validation (if requested) and dispatches events as it moves through the document.
3. If any document triggers are configured on the target collection, each of these will receive the events from the parser in turn. This effectively allows the document triggers to interrupt the parse and hence fail the validation phase, which will abort storing the document.
4. If the parse succeeded, the validation phase is complete. If the parse failed, the process of storing the document is aborted and an error is reported.
5. The parser begins parsing the XML document and dispatches events as it moves through the document.
6. If any document triggers are configured on the target collection, each of these will receive the events from the parser in turn before they are then passed on to the *indexer*. This effectively allows the document triggers to dynamically modify the document should they wish.
7. The indexer receives the events from the parser (or document triggers, if configured) and performs a number of steps:
    a. It extracts *symbols* from the event (e.g., the *qualified name* of an element) and places them in the *symbols table* if they are not already present. The symbols table file is *$EXIST_HOME/webapp/WEB-INF/data/symbols.dbx*.
    b. For each node from the event, it substitutes names for symbols, and then stores the node details into the *persistent DOM*. Elements and attributes are placed into a *B+-tree* index, while other nodes are stored into *pages* in the persistent DOM. The B+-tree index maps nodes to their values located in the pages. The persistent DOM file is *$EXIST_HOME/webapp/WEB-INF/data/dom.dbx*.
    c. It forwards each event to the *index controller*.
8. The *index controller* passes each event through an *indexing pipeline*. The indexing pipeline always starts with the *structural index* (which is mandatory in eXist), then follows any *pluggable index modules* that a user has configured on the target collection (see "Configuring Indexes" on page 275 and "Configuring Full-Text Indexes" on page 286). The structural index file is *$EXIST_HOME/webapp/WEB-INF/data/structure.dbx*.
9. The target collection adds an entry for the newly stored XML document to itself in the *collection store*. The collection store file is *$EXIST_HOME/webapp/WEB-INF/data/collections.dbx*.

10. Either an `afterCreateDocument` or an `afterUpdateDocument` event is raised with each document trigger configured on the target collection.

Binary Document Storage

When given a binary document (i.e., any document that is not XML), eXist stores a copy of it in a *shadow* of its collection hierarchy on disk under the folder *$EXIST_HOME/webapp/WEB-INF/data/fs*; it also stores a copy of the document metadata (e.g., database permissions, owner, group, and created date) into the collection store for the target collection. See Figure 4-8.

Whilst eXist does indeed store the contents of your binary documents as a series of files on disk, you should not manually change the files in the *$EXIST_HOME/webapp/WEB-INF/data/fs* folder, as eXist will be unaware of the changes that you have made. Any changes to this folder could cause the database to lose integrity for binary resources and could lead to stability issues.

While the storage of binary documents into eXist is relatively simple (i.e., the content is maintained in files on disk and additional metadata is maintained in eXist's collection store), you can go further by making use of eXist's binary content extraction facilities (see `contentextraction`) to enable indexing and search of binary documents.

Efficient XML Processing Architecture

There are several crosscutting concerns of eXist's architecture that focus on efficient XML processing and storage; we will take a look at a couple of the more significant ones next. These topics are certainly advanced and may be skipped if you are not interested in eXist's internals.

There are three levels of granularity that eXist is concerned with:

Collection
  An arbitrary grouping of XML and binary documents

Document
  Either an XML document or a binary document

Node
  The nodes within XML documents (i.e., elements, attributes, text, etc.)

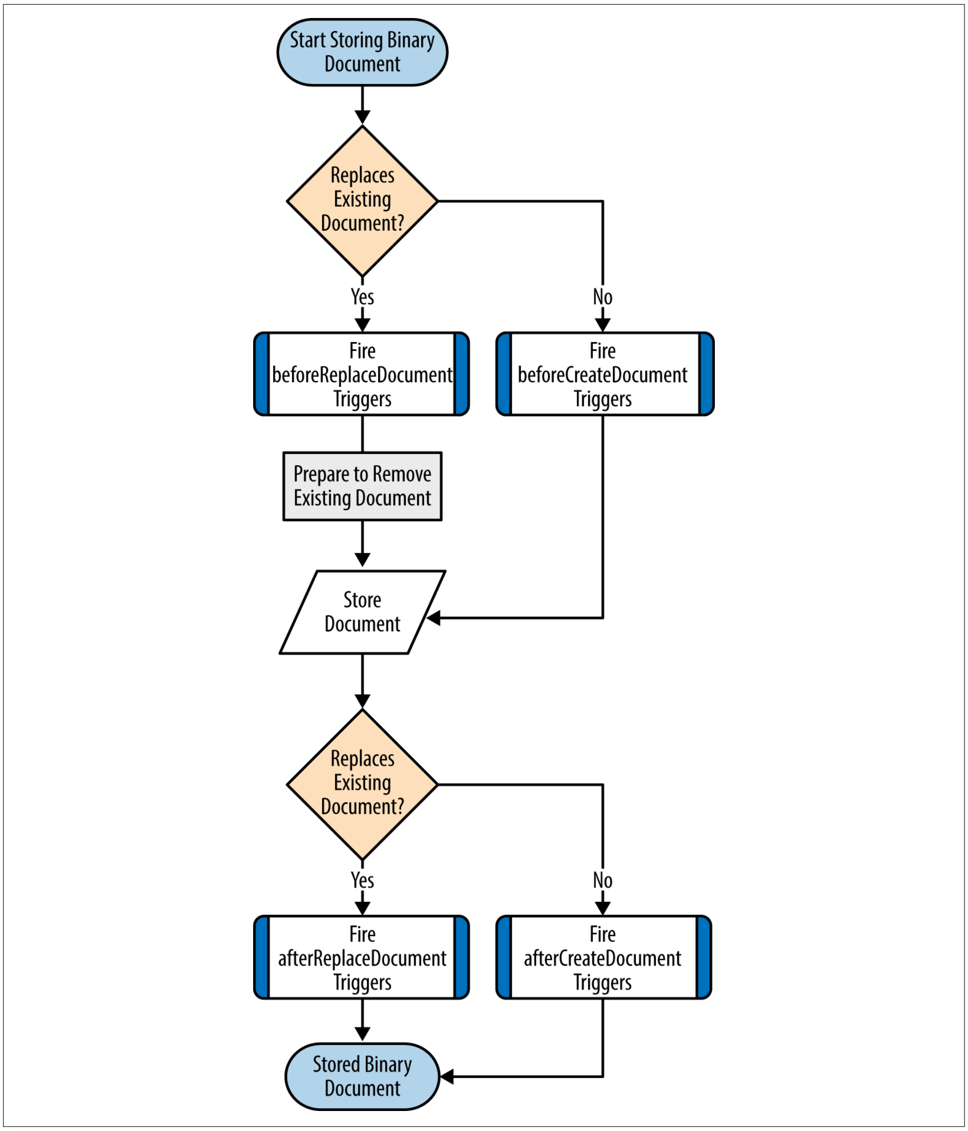

Start Storing Binary Document
Replaces Existing Document?
Yes
No
Fire beforeReplaceDocument Triggers
Fire beforeCreateDocument Triggers
Prepare to Remove Existing Document
Store Document
Replaces Existing Document?
Yes
No
Fire afterReplaceDocument Triggers
Fire afterCreateDocument Triggers
Stored Binary Document

Figure 4-8. Binary storage high-level process

Collections

A collection in eXist is very similar to a folder in a filesystem. It groups a number of related XML and/or binary documents together. Each collection in eXist has a *name* and is identified by a *URI*. A collection stores its own *metadata*, and also all of the metadata of all of the documents present in that collection, as you can see in Table 4-1.

Table 4-1. Collection metadata properties

Metadata property		Description
URI		The URI of the collection in the database
Created Date		The date and time the collection was created
Permissions		
	Owner	The owner user of the collection
	Group	The owner group of the collection
	Mode	The security mode of the collection (e.g., `rwxr-xr-x`)
	Access Control List	The access control list for the collection

In eXist collections are identified by a *hierarchical URI scheme*, and are in fact *hierarchical containers* (Figure 4-9). That is to say, a collection *inherits* all documents from any subcollections, and the configuration of a collection is applied to all subcollections, unless it is explicitly overridden by a subcollection. If you request a collection in eXist, you receive all of the documents in that collection, but if that collection has subcollections, you also receive all of the documents from all of the subcollections of that collection, and so on.

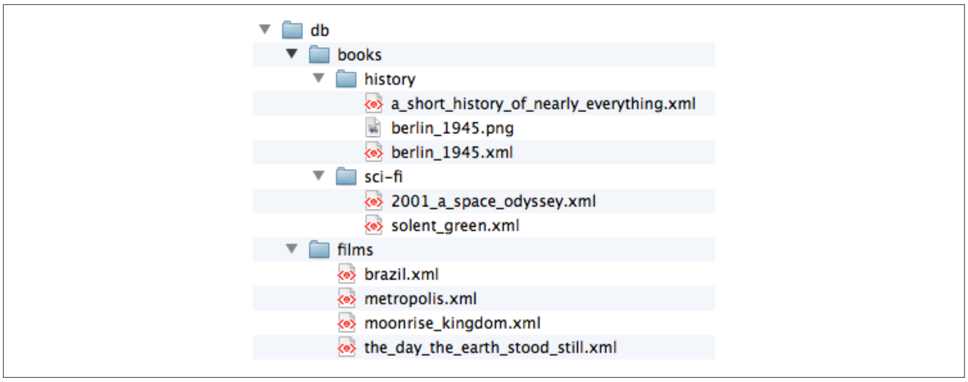


Figure 4-9. Example collection hieracrhy

Collections in eXist always start from the root collection, named *db*, with the corresponding URI */db*. In the example hierarchy shown in Figure 4-9 we can see the following collections:

Name	URI
db	*/db*
books	*/db/books*
history	*/db/books/history*
sci-fi	*/db/books/sci-fi*
films	*/db/films*

Careful organization of your documents into collections in eXist can vastly improve query performance. It is sensible to organize your documents into collections based on the queries that you will want to make of them; typically this is also the way that your organization logically thinks about documents and their meaning.

Organizing documents into collections and querying only those collections that are relevant reduces the amount of documents that eXist must interrogate during a query, and hence this approach lends itself to more efficient processing. In Figure 4-9, for example, you might want to answer queries about all history books; by querying just the */db/books/history* collection, you ensure that you are not querying any documents that you already know are irrelevant. You can also answer queries about all books by querying the */db/books* collection, as this will query each of its subcollections recursively.

Documents

Documents are the basic unit that you work with in eXist, and they come in two forms: XML documents and binary documents. eXist is not particularly fussy about how you organize your information into documents; whether you choose to use several large documents or many smaller documents, eXist will happily store your data. However, your data architecture is an important concern for your business and applications, so here are some concerns that you should consider when deciding on document granularity:

Locking

When a document is being read, it is *read locked*, which means that any incoming updates to the document will be blocked. If you have a few large documents and your system is not mostly concerned with reads, this can introduce contention for the document. If you instead have several smaller documents, they can be read and updated independently. Likewise, when you are updating a document, it is *write locked*. If you have several smaller documents rather than a few large documents, several updates can take place independently.

In addition, your own application may require you to explicitly lock documents—for example, to stop two users in an XForms application from simultaneously editing the same document.

Retrieval

Users retrieving documents directly by URI may not wish to retrieve an entire large document if they are concerned only with a small portion of its information. To a certain extent, you can mitigate this by placing a virtual URI space over the database by using XQuery with URL rewriting or RESTXQ, which then just returns a specific portion of the document. However, using URL rewriting or RESTXQ adds complexity, so if your data architecture doesn't require this, things will be much simpler for you without them.

All that being said, we advise you to make decisions about document granularity based on your domain model. Most importantly, ask yourself, "What do we think of as a document?" Typically the technology concerns can be solved later.

Each document in eXist has a filename and a URI, the latter of which is made up of the collection URI and the filename of the document. Documents, like collections, also maintain a small set of metadata, as Table 4-2 shows.

Table 4-2. Document metadata properties

Metadata property		Description
Name		The filename of the document in the database
Created Date		The date and time the document was created
Last Modified Date		The date and time the document was last modified
Internet Media Type		The Internet media type of the document
Permissions		
	Owner	The owner user of the document
	Group	The owner group of the document
	Mode	The security mode of the document (e.g., `rwxr-xr-x`)
	Access Control List	The access control list for the document

Dynamic Level Numbering of Nodes

You have already seen in "XML Document Storage and Indexing" on page 72 that while, as a user of eXist, you are working with XML documents (and the nodes from those documents), eXist itself does not work with the textual XML document you provided it, but rather with computationally-optimized model of that document.

XML documents themselves are constructed from nodes, and at the very core of everything that eXist does with nodes is DLN (dynamic level numbering). DLN enables eXist to efficiently identify both a node and its structural relationship to other nodes.

Each node from an XML document is given a *DLN identifier*. DLN identifiers are hierarchical in nature and borrow concepts from the Dewey Decimal Classification system. Each DLN identifier assigned to a node starts with the identifier of the parent node and concludes with a number for the node under consideration, which indicates its position among its siblings.

DLN identifiers are unique within a document, which allows for easy identification of a node. They also have some additional structural properties that are very useful:

- They include the ID of the parent node.
- By virtue of the IDs being hierarchical, you also have the ID of every ancestor node.
- Given two DLN IDs, you can determine the structural relationship between two nodes.

Here are some examples of how the DLN identifiers on the nodes shown in Figure 4-10 can easily be used to perform structural joins in eXist and answer queries without your needing to examine the nodes and traverse node relationships themselves:

Root
By looking at the first level of any node identifier, we find its root node. For example, if you take the node identifier for the text node "Apple," which is `1.1.1.1`, and look at the first number before the first period, you can see that the root identifier is `1`, and the node with the identifier `1` is the element "Shopping."

Ancestors
By looking at all parts of a node identifier from left to right, excluding the last part, we can find all ancestors of that node. For example, if you take the node identifier for the text node "Apple," which is `1.1.1.1`, and work backward from right to left looking at the parent, then the parent of the parent, and so on until you reach the root, you will find the ancestor nodes "Name" (`1.1.1`), "Fruit" (`1.1`), and "Shopping" (`1`).

Parent
By looking at the parent identifier levels of any node identifier, we can find its parent node. For example, if you take the node identifier for the element "Quantity," which is `1.2.2`, and look at the identifier all the way up to the last period, you will see that the parent identifier is `1.2`, and the node with the identifier `1.2` is the element "Vegetable."

Child

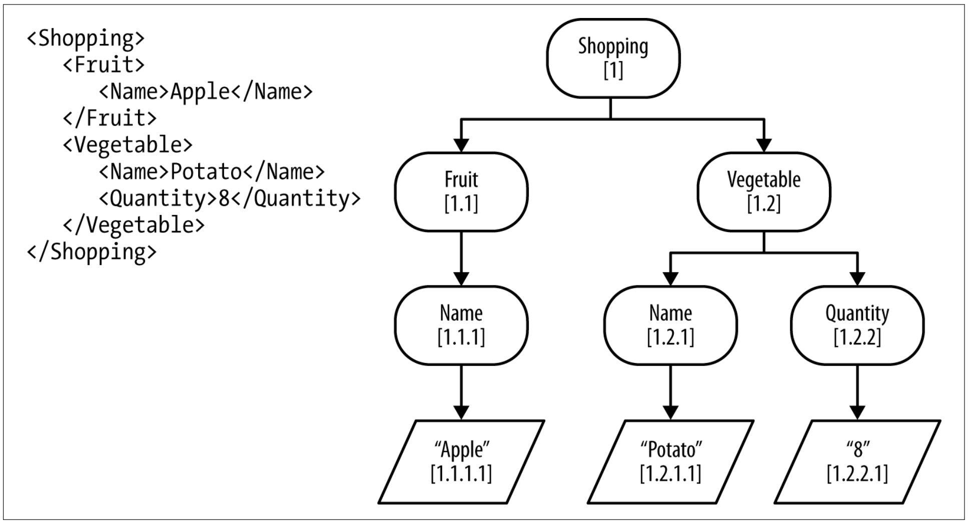
By looking at the identifiers of any two nodes, we can determine if one is the child of the other. Given the identifiers for the element "Name," which is `1.2.1`, and for the text node "Potato," which is `1.2.1.1`, because the identifier for "Potato" has as a prefix the identifier for "Name" and has one extra level in its identifier, we can tell that it is a child of "Name." We can also tell that it is the first child of "Name" from the last level of its identifier.

Siblings

By looking at the identifiers of any two nodes, we can determine if they are siblings. Given the identifier for the element "Name," which is `1.2.1`, and for the element "Quantity," which is `1.2.2`, because their identifiers have the same number of levels and have the same prefix we can tell that the node with identifier `1.2` ("Vegetable") is their parent. We can also tell that "Name" is the preceding sibling of "Quantity" because its last level has the lower identifier number of `1`; "Quantity" is numbered `2`, and therefore must be the following sibling of "Name."

Common ancestor

By looking at the common prefix of any two node identifiers, we can find their common ancestor node. For example, if you take the node identifier for the text node "Potato," which is `1.2.1.1`, and the node identifier for the text node "8," which is `1.2.2.1`, and examine them from left to right, you will find that the common prefix is `1.2`, and the node with the identifier `1.2` is the element "Vegetable."



```
<Shopping>
   <Fruit>
      <Name>Apple</Name>
   </Fruit>
   <Vegetable>
      <Name>Potato</Name>
      <Quantity>8</Quantity>
   </Vegetable>
</Shopping>
```

Figure 4-10. A simple XML document and the corresponding DLN model

eXist provides some extensions that allow you to view the DLN identifiers assigned to nodes, and also to retrieve nodes by their DLN identifiers. See the `util:node` and `util:node-by-id` functions in `util`. eXist also provides the `add-exist-id` serialization option, which inserts the DLN identifiers into attributes on each element of a document or node when you retrieve it from eXist; see "Serialization" on page 118.

DLN helps eXist perform efficient queries of XML documents, as often eXist can calculate the sets of nodes involved in an XPath expression by performing computations using just their DLN identifiers.

Should you choose to retrieve nodes from eXist using DLN identifiers, you should be aware that DLN identifiers are not guaranteed to be stable. After you update a document by replacing it or using the XQuery update extension or XUpdate, it is likely that some of the nodes in the document will have different identifiers.

Dynamic Level Numbering and Updates

So far we have looked at DLN within a fixed document, but it can also be useful to understand how the DLN identifiers on nodes in a document may change if that document is updated. There are three ways in which an XML document in eXist may be updated:

Replacement
: In this instance the node identifiers are not updated; rather, the original document is deleted and a new document is stored, and they just happen to have the same URI within the database.

XUpdate
: An XUpdate document may be processed against one or more documents; this could lead to the insertion, deletion, or modification of nodes in the documents. These modifications will lead to identifier changes.

The XQuery update extension
: Similarly to XUpdate documents, XQuery update statements may be processed against one or more documents, which could lead to the insertion, deletion, or modification of nodes in the documents. These modifications will lead to identifier changes. The difference between this mechanism and XUpdate is that all operations from a single XUpdate document will occur within the same database transaction, while each XQuery update statement will be executed in its own transaction.

When a document is modified, we want to preserve the integrity of the conceptual model of hierarchical identifiers that have previously been assigned to nodes in the document. However, we need only concern ourselves with insertions between existing nodes, because:

- Insertions as the *last following sibling* or insertions as a new *only child* can be assigned new DLN identifiers without our needing to change any existing identifiers to maintain integrity.
- Modifications to existing nodes (e.g., renaming) do not change their identifiers.
- Deletions do not break the integrity of our model. It does not matter if sibling `1.2` is followed by sibling `1.4` because `1.3` was deleted, as long as higher numbers indicate the order.

If we could not maintain our existing identifiers on insertion, we would have to renumber all of the nodes in the document, which would be a very computationally expensive exercise. In fact, this is exactly what eXist used to do before it implemented DLN. Luckily, DLN provides a mechanism whereby we don't need to renumber existing nodes but can still perform insertions: *subvalue identifiers*.

If we consider the case where we have two elements called "Name" and "Quantity" with the identifiers `1.2.1` and `1.2.2`, respectively, we could insert a new sibling element named "Cost" between them by using subvalues. As shown in Figure 4-11, the new element "Cost" would have the identifier `1.2.1/1`. The `/1` indicates the subvalue; that is, "Cost" follows "Name" (`1.2.1`) but precedes "Quantity" (`1.2.2`).

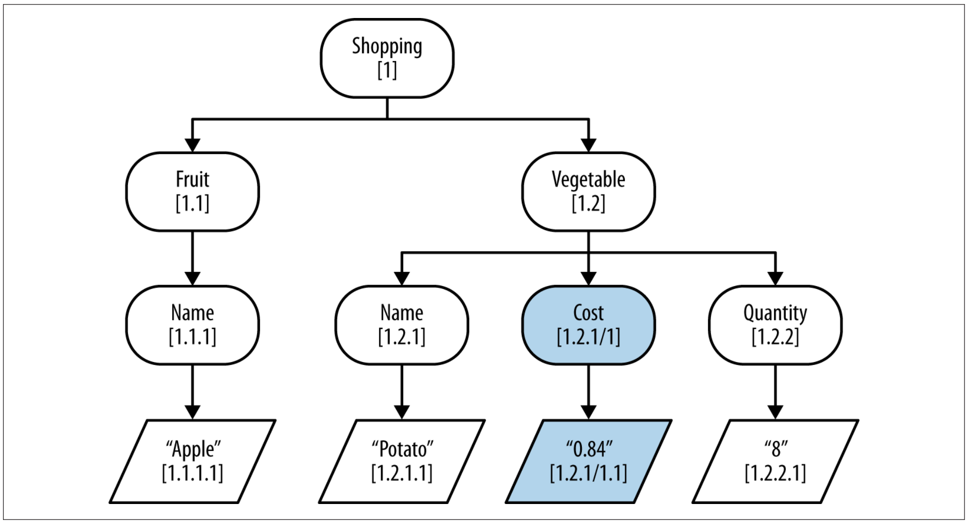


Figure 4-11. A simple XML document and the corresponding DLN model after update

While subvalues give us the ability to perform fast updates and still query documents, comparing subvalue DLN identifiers against non-subvalue DLN identifiers is more costly. For this reason, eXist will occasionally trigger the background *defragmentation* of a document that has had significant updates made to it. The defragmentation effectively renumbers all of the nodes in the document, removing the subvalues.

The fragmentation of documents occurs when XQuery update or XUpdate expressions are executed against XML documents. After the evaluation of an XQuery running XQuery update expressions or an XUpdate document, the updated documents will be checked for fragmentation. If they exceed the allowed fragmentation limit set in *$EXIST_HOME/conf.xml*, then they will be queued for defragmentation. Defragmentation happens in the background in the database, but during defragmentation a document is locked against further writes!

Paging and Caching

Several of the core database files (*dom.dbx*, *structure.dbx*, and *collections.dbx*) that are kept on disk are organized into *pages* of 4 KB. A page is simply a contiguous region that is read or written in an atomic operation. Rather than randomly seeking and reading individual bytes as required, mechanical rotational storage systems (such as hard disks) are much faster at larger linear reads and writes. However, pages themselves are not necessarily always in the order that you need to answer a query; as such, good random-access speed is still a requirement of the filesystem and underlying storage system.

The size of a page in eXist is configurable in *$EXIST_HOME/conf.xml* via the attribute indicated by the XPath */exist/db-connection/@pageSize*. This should be aligned with the block size of your filesystem; today 4 KB is typically correct. You can also experiment with setting this to a multiple of the block size when testing for the optimal performance of eXist with your data. However, be aware that you can only change the page size before creating a database (i.e., with no *.dbx* files in *$EXIST_HOME/webapp/WEB-INF/data*).

The persistent on-disk DOM (*dom.dbx*) and collections (*collections.dbx*) files are split into two parts, a data section and an index section (which is a B+ tree). The data section contains the node or document and collection metadata, while the index section ensures the quick lookup of collections, documents, and nodes. The structural index (*structure.dbx*) file is literally just an index and has no data section to its file.

In any database system, aside from network I/O, disk I/O is always the slowest component. For example, at the time of writing, suppose you want to read 1 MB of data from disk. It typically takes *32 times as long* just to locate the data on the disk as it does to read it from memory. To then actually read the data from disk takes *80 times as long* as it does to read it from memory. Therefore, it is highly desirable to optimize any database system to avoid disk I/O as much as possible, and eXist is no exception.

Access to all database files in eXist that make use of paging is performed through specific page caches. The job of the caches is to keep frequently accessed data in memory, to avoid having to retrieve it from disk. eXist has separate caches for the data and index sections of its files. The caches in eXist are LRU (Least Recently Used) caches, so pages are evicted from the caches (under various circumstances) based on the time they were last accessed. In particular, the caches for the index sections (B+ tree) distinguish between pages that contain internal (nonleaf) nodes and leaf nodes. Since internal nodes are likely to be accessed more frequently, they are assigned a higher priority and are not evicted from the cache before related leaf nodes.

Caches grow (up to a maximum limit) and shrink automatically as required. If a cache is *thrashing* (i.e., pages are frequently replaced), it will request more memory from the cache manager. Providing the cache manager has memory available, extra memory will always be granted to a requesting cache. Exactly how and when a cache grows is specific to each cache implementation. For example, the data section cache of the persistent on-disk DOM will never grow, because it is very unlikely that the same page will be frequently accessed over a short, finite duration; therefore, the cache size is fixed to 256 pages (typically 1 MB when you're using 4 KB pages). Conversely, the collection cache attempts to calculate the memory required by the metadata of each collection, and tries to fill itself with the metadata of the hottest collections. During periods of low database activity, the cache manager may decide to reclaim space from the caches so that it can quickly be reallocated in the future.

Caches in eXist are configured to use an explicit maximum amount of memory, which is subtracted from the maximum memory made available to eXist (the JVM's `-Xmx` maximum memory setting).

All caches (except the collection cache) share a single caching space in memory. The size of this caching space is configurable in *$EXIST_HOME/conf.xml* via the attribute indicated by the XPath */exist/db-connection/@cacheSize*. The collection cache (for *collections.dbx*) has its own caching space in memory and may likewise be configured in the same file at the XPath */exist/db-connection/@collectionCache*.

For information on how to tune eXist's memory and cache settings for your database, see "Cache Tuning" on page 394.

CHAPTER 5

Working with the Database

At its core, eXist is an XML database. It stores XML efficiently and makes fast querying possible. Besides XML, it is also capable of storing other file types. Although in its default configuration it doesn't do much with them besides storing and retrieving, this capability is useful when you are building applications with eXist.

This chapter is about eXist's database: what's in it, how it's structured, how you access and update its content, and (of course) how you query it.

The Database's Content

In this section we will dive into the contents of the database.

Help: Where Is My XML?

Superficially, when you're accessing the database using a WebDAV client, eXist's database looks like a filesystem: you'll see directories and files, and you can work with them directly as you would in any other filesystem.

Of course, under the surface things look very different. XML files are "ripped apart" (or "shredded"), indexed, and stored in a way that makes searching, indexing, and retrieval efficient. You can see this in action when you store a file in the database and reopen it: the indentation after reopening will probably look different. This is because the document was not stored as is, but rather as a tree structure according to the XML data model. The document was likely recreated using different indentation rules than those with which the original was created.

Related to this is one of the most frequently asked questions from eXist newcomers: "Where is my XML?" People look in the database storage directory *$EXIST_HOME/webapp/WEB-INF/data/* (if the installation defaults were used) and see a bunch of

**.dbx* files. The stored XML is nowhere in sight. What makes things even more confusing is that stored non-XML files (binaries and queries) can be found in the *fs* subdirectory.

So where is the XML? Don't worry. Although you can't find it as a file, the XML document is stored inside the **.dbx* files. eXist ensures that you can access the XML as though it is still a file *and* access and query it in an efficient way.

If you're interested in how eXist performs this trick, please refer to Chapter 4.

Terminology

Let's define some important terminology:

Collections

As noted previously, what is called a *directory* in a filesystem is called a *collection* inside eXist's database. When you use, for instance, the WebDAV interface to look inside the database, you'll see no difference.

The reason it is called a collection is linked to the definition of the XPath `collection` function. This function retrieves documents from something it calls a collection, but the XPath specification does not say exactly what this collection concept actually is. eXist uses an internal directory-like structure as the basis for its collections. Read more about this in "Collections" on page 77 and "The collection Function" on page 93.

Resources

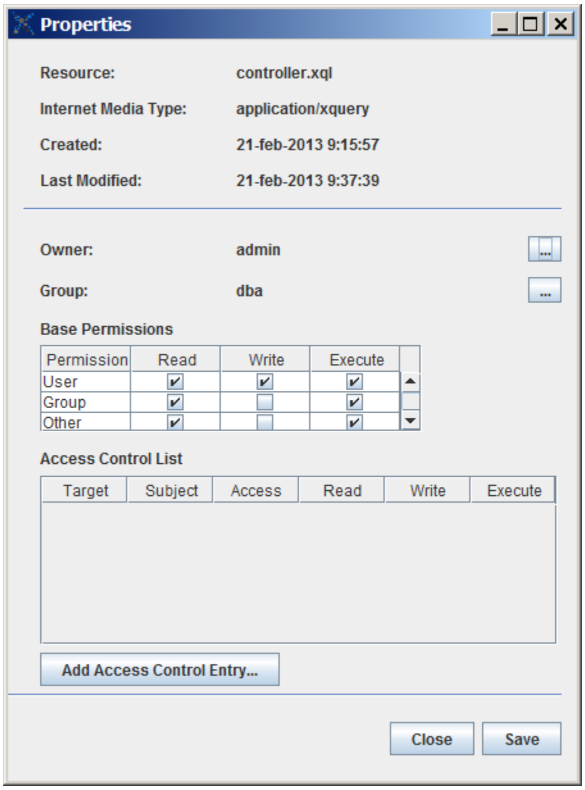
What is called a *file* in a filesystem is called a *resource* inside eXist's database. Resources can be anything you usually store in a file: images, CSS files, XQuery scripts, and, of course, XML documents.

Documents

A resource containing a (well-formed) XML document is called a *document*.

Properties of Collections and Resources

Anything stored in the database has several properties attached. You can view and change these properties using, for instance, Java Admin Client tool (see Figure 5-1).



Resource: controller.xql
Internet Media Type: application/xquery
Created: 21-feb-2013 9:15:57
Last Modified: 21-feb-2013 9:37:39

Owner: admin
Group: dba

Base Permissions

Permission	Read	Write	Execute
User	✓	✓	✓
Group	✓		✓
Other	✓		✓

Access Control List

Target	Subject	Access	Read	Write	Execute

Figure 5-1. Viewing and modifying properties using Java Admin Client tool

Alternatively, you can work with the collection and resource properties through the dashboard's collection browser or from within eXide. You can also view or change these properties programmatically from XQuery code with functions from eXist's `xmldb` extension module. See more about this in "Controlling the Database from Code" on page 107.

On the Client tool's Properties screen, you'll see all of the following for the selected resource:

Internet Media Type
: The Internet media type of a resource is first set by eXist when the resource is created, based on its content and/or file extension. However, if you create resources with the `xmldb` extension module (see "Controlling the Database from Code" on page 107) you have the option to control this yourself.

After creation, you can change the Internet media type of a resource programmatically. However, you can't turn a non-XML resource into an XML one, or vice versa.

Collections do not have an Internet media type.

> The Internet media type of a resource (when not set explicitly) is determined from the configuration file *$EXIST_HOME/mime-types.xml*. This file maps file extensions to Internet media types (or MIME types, as they used to be known) and tells eXist when to treat a file as XML instead of binary data.

Created and Last Modified
: Resource have both created and last modification date and time, however collections only have a created date and time.

Owner, Group, Base Permissions, and Access Control List
: The security settings for this resource. Find more information about this in Chapter 8.

System Collections

The database's collection */db/system* contains eXist system-specific information. Most of the information underneath this collection is maintained by eXist itself and there is usually no need for a "normal" user to access it, because there are extension functions for this.

For instance, in */db/system/security/exist/accounts* you'll find user information, and by querying the resources found there you could create a list of registered users and the user groups they participate in. However, we definitely advise against doing this! eXist does not guarantee that these files nor the format of their content will remain stable in the future. Rather, this is internal configuration information and as such is potentially subject to change without notice whenever a new version of eXist is released. The preferred alternative is to instead use the appropriate functions from the Security Manager and/or the `xmldb` extension modules, which should remain stable.

There is one important exception to this rule: the */db/system/config* collection *must* be used to configure important properties for collections such as: indexes, triggers, and validation. Underneath */db/system/config* you'll find a partial copy of the database's collection structure with *collection.xconf* resources in some of them. We'll give you more information about this in the chapters to come.

Addressing Collections, Resources, and Files

To work with documents and collections stored in the database, you need to be able to address them, point to them, and know their names. There are several ways to do this. Extra care should be taken when your resource and/or collection names contain spaces or special (e.g., accented) characters.

Use URIs

There is one extremely important thing you should be aware of when addressing resources and collections inside eXist's database: *eXist uses URL-encoded URIs for naming*. This means that all reserved characters, according to the URL encoding rules (for more information, see *http://tools.ietf.org/html/rfc3986#page-12*), *must* be percent-encoded!

For instance:

- A document that shows up as *an example.xml* in the WebDAV browser *must* be addressed programmatically as *an%20example.xml*.
- A document in the collection */db/my app (test)/test.xml* must be addressed with */db/my%20app%20%28test%29/test.xml*.
- When you request a list of documents inside a collection (with the function `xmldb:get-child-resources`), the names returned are URIs, and as such they are URL-encoded.

> If you are not extremely careful with this, your application will quickly become a mess. Always distinguish internally between *names* (useful for displaying document/collection names to the user) and *URIs* (useful for addressing the documents/collections).

Conveniently, eXist contains standard functions to transform a name into a URI, and vice versa:

`xmldb:decode-uri`
: Decodes a URI into a name; that is, it changes all percent-encoded characters into their UTF-8 equivalent characters.

`xmldb:encode-uri`
: Encodes a name into a URI; that is, it checks for reserved characters and changes them into the equivalent URL percent-encoding.

Unfortunately, `xmldb:encode-uri` does not check for the optional *xmldb:exist://* database prefix (see "XMLDB URIs" on page 92) and erroneously encodes *xmldb:exist:///a/b/c* into *xmldb%3Aexist%3A/a/b/c*, which is probably not what you want.

Relative versus absolute paths

Using relative paths in eXist can be confusing. To help clarify when to use relative versus absolute paths, we have to distinguish between two situations:

Paths in a static context
: These are paths resolved at compile time—for instance, an XQuery `import module at` clause, or an `xsl:include` or `xsl:import` in an XSLT stylesheet. These paths are resolved, as expected, against the location of the code that does the import or include. We strongly advise using relative paths here because it makes moving your code around much easier.

Paths in a dynamic context
: These are paths resolved at run time—for instance, in code like `doc("/db/myapp/data/data.xml")`. These paths are resolved using what is called the *base collection* of a query. How this base collection is determined unfortunately depends on the way the query came to life. The rules are difficult to remember, confusing, and subject to change as eXist evolves over time.

: Because the invocation of a query matters, a relative path is not guaranteed to always work the same way. So, our advice is to use absolute paths for addressing collections and resources in code whenever possible.

XMLDB URIs

Instead of writing a direct path to a database resource, such as */db/mycollection/test.xml*, you can use a so-called *XMLDB URI*. For this, add the prefix *xmldb:exist://* in front of the resource, as in *xmldb:exist:///db/mycollection/test.xml*. (Note that there are now three slashes after *xmldb:exist:*.) Used like this, both notations are equivalent and point to the same document. It's a matter of preference which one to use: the XMLDB URI is somewhat longer but more specific, and therefore you might consider it more self-documenting. It is worth noting that when using the Java Admin Client to execute queries in embedded mode, you have to use the XMLDB URI. Likewise, when writing XQuery using oXygen with eXist, depending on the version, you may also need to use the XMLDB URI for eXist to recognize module import sources.

Accessing files

If you want to access a file on the filesystem (not in the database), use the *file://* prefix, as in:

```
doc("file:///home/erik/test.xml")
```

To manipulate the filesystem, eXist has a `file` extension module. In contrast to what you might expect, this module does *not* use the *file://* prefix syntax.

The XPath Collection and Doc Functions in eXist

The XPath `collection` function is defined as returning a sequence of (usually document) nodes. The `doc` function is defined as returning `document-node`, or the empty sequence if it cannot find the document indicated by the passed URI. Both use a URI as a parameter. These are important functions for XML databases because they allow you to easily address subsets of your database content for further inspection or processing. The XPath standard defines their behavior as implementation-dependent, so we need to know how eXist handles them.

The collection Function

The `collection` function in eXist returns the set of *resources* residing in the collection identified by its URI parameter, including those in its subcollections, recursively. In other words, it will return a sequence containing all resources underneath a certain collection path in the database. If you want only the resources in the collection itself (without those in subcollections) you can use the extension function `xmldb:xcollection` instead.

Now, maybe to your surprise, `collection` returns not only the XML documents found, but *all* resources. To illustrate this, assume we have a collection called */db/test* in which there are two files: *test.xml* (an XML file) and *test.pdf* (a PDF file). Now run the following query:

```
for $doc in collection("/db/test")
return
  base-uri($doc)
```

The result will be `/db/test/test.xml` *and* `/db/test/test.pdf`.

Of course, besides getting their URIs with the `base-uri` function, you won't be able to do much with the non-XML nodes returned by `collection`. However, getting a list of all resources can be quite useful in some cases—for instance, for showing the user a list of available content.

You can also use functions from the `xmldb` extension module for iterating over the contents of collections. Read more about this in "Controlling the Database from Code" on page 107.

You can easily check whether a node points to an XML document by calling `exists($doc/*)`, where `$doc` is a member of a sequence returned by `collection` as in the preceding example. This will only return `true` for XML documents.

Remember, the behavior of the collection function is implementation-defined. This means that your XQuery's calls to `fn:collection` may not behave the same on different platforms, which can introduce issues for code portability.

The doc Function

If you know (or have computed) the URI to an XML document, the easiest and most straightforward way to address its content is using the XPath `doc` function. For example:

```
let $documenturi := "/db/myapp/contents.xml"
for $item in doc($documenturi)//Item
return ...
```

An interesting (and sometimes useful) behavior of the `doc` function in eXist is that when it gets passed the URI of a nonexistent or non-XML document, it silently returns the empty sequence (without throwing an error).

Querying the Database Using REST

The simplest and most often used way to access the database's content is using eXist's REST (a.k.a. REST-style or RESTful) interface. It allows you to query the database by firing HTTP requests at it. You can do this programmatically from another application or, for `GET` requests, by hand using a web browser. This section will examine eXist's REST interface at a fairly basic level; a more thorough explanation (from a system integration point of view) can be found in "REST Server API" on page 319.

By default, to access the REST interface for a standard eXist setup, start the URLs with:

```
http://localhost:8080/exist/rest/
```

You might be rightfully worried now about how your URLs are going to look when you're building an application in eXist. The */exist/rest* looks awful! Don't worry: you can tune the URLs to your liking; Chapter 9 explains how to do this.

The database operations are, like in a true REST interface, mapped to the HTTP request methods GET, PUT, DELETE, and POST. Most often used is GET.

Security

Of course, everything you do through the REST interface is subject to the strict security rules of eXist. For instance, if you're not allowed to see a file (that is, you have no read permission), it will not show up when you request the contents of the containing collection in a GET request.

The default identity when you're firing a request through this interface is the limited guest account. HTTP authentication is supported to change identity (see, for instance, *http://www.httpwatch.com/httpgallery/authentication/*).

Even with eXist's strict security in place, allowing REST access on a production server can be scary: it is a powerful tool with lots of opportunities for misuse or inadvertent damage. You can turn it off completely, which is probably something you'd want to do on production servers (unless you really need direct REST access). See "Disabling direct access to the REST Server" on page 180 for more information.

GET Requests

HTTP GET requests are the workhorses for accessing data and for XQuery scripts. Using GET requests is probably the most convenient way to execute queries stored in the database and/or retrieve XML and other contents from it: you can simply do it from your web browser.

For an HTTP GET request, eXist examines the remainder of the URL (by default the part after */exist/rest*) and reacts in one of the following ways:

- If a _query URL parameter is present (see the following), it will use this as the XQuery to execute and will return its result.
- If the remainder of the URL points to a collection, it will return an XML fragment describing its content. For instance:

```
<exist:result xmlns:exist="http://exist.sourceforge.net/NS/exist">
  <exist:collection name="/db/test"
     created="2012-09-13T08:18:02.35+02:00"
     owner="guest"
     group="guest" permissions="rwxr-xr-x">
    <exist:resource name="test.xml"
```

```
       created="2012-09-13T08:18:25.088+02:00"
       last-modified="2012-09-13T08:18:25.088+02:00" owner="guest"
       group="guest" permissions="rw-rw-rw-"/>
    <exist:resource name="test.pdf"
       created="2012-09-13T08:18:29.9+02:00"
       last-modified="2012-09-13T08:18:29.9+02:00" owner="guest"
       group="guest" permissions="rw-rw-rw-"/>
  </exist:collection>
</exist:result>
```

- If the remainder of the URL points to a file (XML or otherwise), the contents of the file are returned using the Internet media type stored in the database.
- If the remainder of the XML points to an XQuery script (Internet media type `application/xquery`) with execute permission, it will be executed and the results returned.

A `GET` request accepts the following parameters:

`_xsl=xsl-stylesheet-reference | no`
: Applies an XSLT stylesheet to the result of getting an XML resource or executing an XQuery script. The path is considered an internal database path, unless it contains an external URI (e.g., starts with *http://*). Setting this parameter to `no` disables all stylesheet processing.

> Applying an XSLT stylesheet in this manner always changes the response's Internet media type to `text/html`.

`_indent=yes | no`
: Specfies whether to indent (pretty-print) the returned XML. The default is `yes`.

`_encoding=character-encoding`
: Indicates the character encoding to use. The default is `UTF-8`.

`_query=XQuery-expression`
: Executes the given XQuery expression on the result.

`_howmany=number-of-items`
: When you pass a query by the `_query` parameter and the result is a sequence, specifies how many items to return from the sequence. The default is `10`.

`_start=starting-position`
: When you pass a query by the `_query` parameter and the result is a sequence, specifies at which position to start returning results from the sequence. The default is `1`.

`_wrap=yes | no`
: Indicates whether returned query results (and collection contents) should be wrapped in an `exist:result` root element (the namespace prefix `exist` is bound to the namespace `http://exist.sourceforge.net/NS/exist`). The default is `yes` for collection contents and queries passed in the `_query` parameter, and `no` otherwise.

`_source=yes | no`
: Indicates whether the query should display its source code. You must explicitly allow this by adding the name of the query file to *$EXIST_HOME/descriptor.xml* in the `allow-source` section (and then restart eXist for the changes to take effect).

PUT Requests

HTTP `PUT` requests can be used to store or update documents in the database. The remainder of the URI (the part after */exist/rest*) is used as the target location of the document.

As an example of how to do this, we will use eXist's own `httpclient` extension module to store an XML document into the database:

```
let $URI :=
  'http://localhost:8080/exist/rest/db/apps/exist-book/data/put-example.xml'
return
  httpclient:put(xs:anyURI($URI), <new-file-by-rest-put/>, false(), ())
```

Be aware that this is an example to show you how to use HTTP `PUT`. If you want to store a document into the database from your own XQuery program, it is more efficient to avoid the HTTP overhead and use `xmldb:store` instead (see "Creating Resources and Collections" on page 109).

DELETE Requests

An HTTP `DELETE` request does exactly what its name implies: it deletes the collection or resource pointed to by the remainder of the URL (the part after */exist/rest*) from the database. The returned HTTP status code will indicate whether the deletion was successful.

POST Requests

HTTP POST requests can be used for three distinct purposes:

- If the remainder of the URI (the part after */exist/rest*) references an XQuery program stored in the database, it will be executed.
- If the body of the POST request is a valid XUpdate document, the XUpdate processor will be invoked to update the database. An explanation of how this works and the XUpdate XML format is described in "XUpdate" on page 105.
- If the body of the POST request is XML and uses the http://exist.sourceforge.net/NS/exist namespace, it is interpreted as a so-called *extended query request*. These can be used to post complex XQuery scripts that are too large or too unwieldy to pass in a _query parameter of a GET request. The result will be wrapped in an exist:result element. The XML format for extended query requests is described in "Extended query request XML format" on page 99.

For example, let's fire an HTTP POST request containing an extended query request at our own database using the eXist httpclient extension module:

```
let $URI := 'http://localhost:8080/exist/rest/doesnotmatter'
let $query := 'for $i in 1 to 10 return <Result index="{$i}"/>'
let $request :=
  <query xmlns="http://exist.sourceforge.net/NS/exist" start="3" max="3">
    <text>{$query}</text>
  </query>
return
  httpclient:post(xs:anyURI($URI), $request, false(), ())
```

The result will be something like:

```
<httpclient:response xmlns:httpclient="http://exist-db.org/xquery/httpclient"
  statusCode="200">
  <httpclient:headers>
    <httpclient:header name="Date" value="Mon, 17 Sep 2012 12:45:02 GMT"/>
    <httpclient:header name="Set-Cookie"
       value="JSESSIONID=4mpvajj2ez99sa2ik31kzjvq;Path=/exist"/>
    <httpclient:header name="Expires" value="Thu, 01 Jan 1970 00:00:00 GMT"/>
    <httpclient:header name="Content-Type" value="application/xml;
      charset=UTF-8"/>
    <httpclient:header name="Transfer-Encoding" value="chunked"/>
    <httpclient:header name="Server" value="Jetty(7.5.4.v20111024)"/>
  </httpclient:headers>
  <httpclient:body mimetype="application/xml; charset=UTF-8" type="xml">
    <exist:result xmlns:exist="http://exist.sourceforge.net/NS/exist"
    exist:hits="10"
      exist:start="3" exist:count="3">
      <Result index="3"/>
      <Result index="4"/>
      <Result index="5"/>
    </exist:result>
  </httpclient:body>
</httpclient:response>
```

The major part of the output is generated by the `httpclient:post` function. For the result of our extended query `POST` request, look at the contents of the `httpclient:body` element. Notice that since the request returns a sequence and we specified a start position and length of `3`, we get the third through fifth elements of the sequence only.

Extended query request XML format

An extended query request has the following format:

```
<query xmlns = "http://exist.sourceforge.net/NS/exist"
       start? = integer
       max? = integer
       cache? = "yes" | "no"
       session-id? = string >
    text
    properties?
</query>
```

- `start` contains the index (counting from 1) of the first item to be returned.
- `max` is the maximum number of items to return. Together with `start`, it allows you to control which part of the results you'll see.
- Setting `cache` to `yes` will have the query start a session. The session ID will be returned in the result (in the `exist:session` attribute on the `exist:result` element) and must be passed in the `session-id` attribute on subsequent requests.
- `session-id` allows you to pass a previously created session identifier.

The `text` element is used to pass the query. You will most likely want to enclose its contents in a `CDATA` section (`<text><<![CDATA[ ... ]]></text>`) to avoid XML parsing errors.

The `properties` element can be used to set serialization properties like indenting and character encoding:

```
<properties>
  <property name = string
            value = string >*
</properties>
```

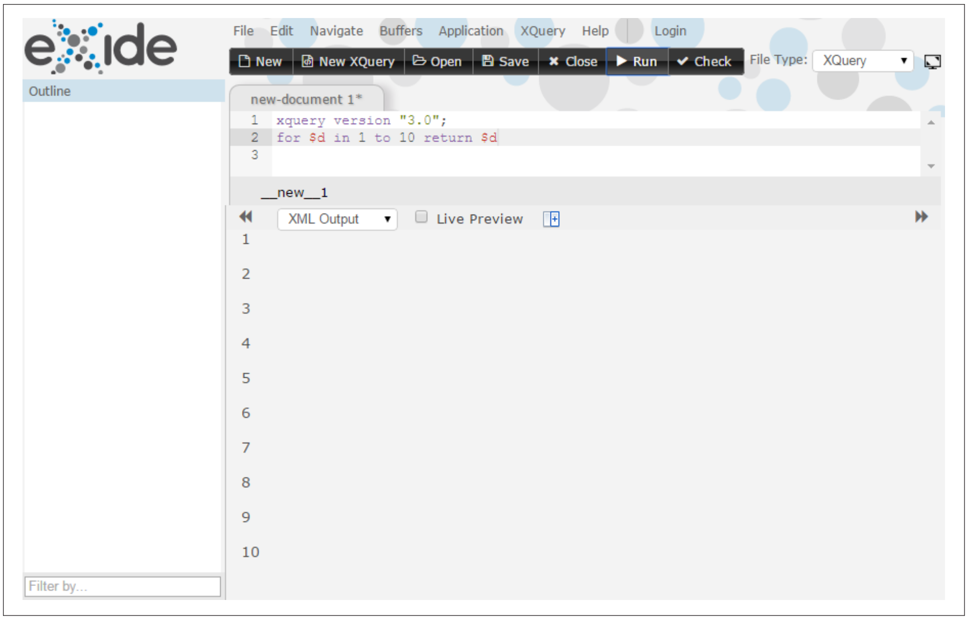
A list of serialization properties can be found in "Serialization Options" on page 119.

Ad Hoc Querying

There are several ways to query the database on a more "ad hoc" basis; that is, not really to create applications with, but rather to explore and experiment.

Querying using eXide

To query using eXide, simply fire up eXide from the dashboard, type your query in an empty document, and click Run (or press Ctrl-Return), as shown in Figure 5-2.



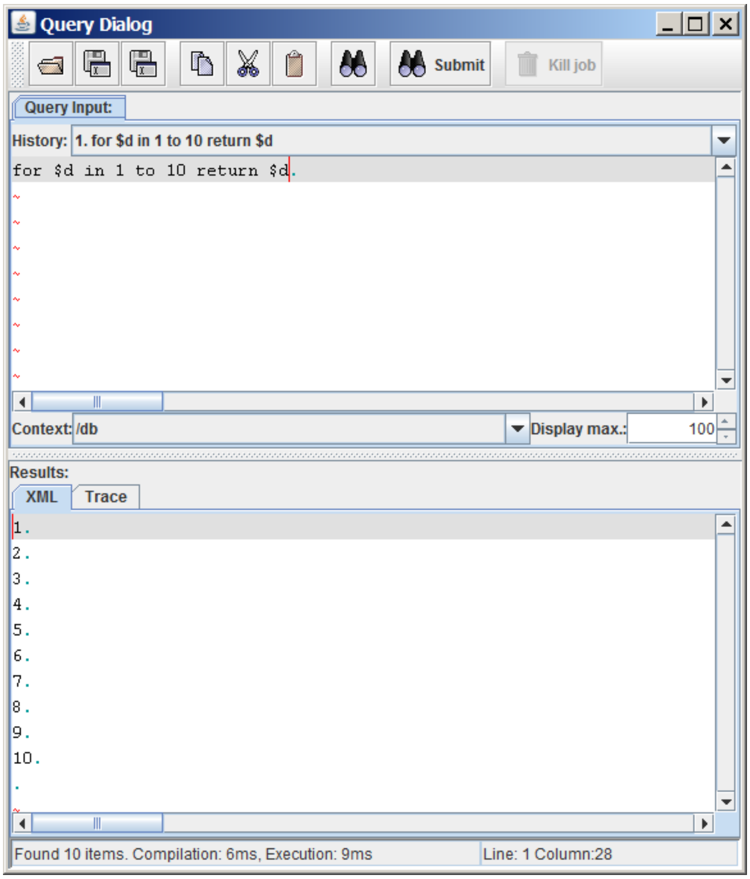
```
xquery version "3.0";
for $d in 1 to 10 return $d
```

```
1
2
3
4
5
6
7
8
9
10
```

Figure 5-2. Ad hoc querying in eXide

Querying using the eXist Client tool

eXist's Java Admin Client tool can be used to query the database too. Click on the binoculars icon and the XQuery dialog will open, as Figure 5-3 shows.



```
for $d in 1 to 10 return $d.
```

```
1.
2.
3.
4.
5.
6.
7.
8.
9.
10.
```

Found 10 items. Compilation: 6ms, Execution: 9ms   Line: 1 Column:28

Figure 5-3. The XQuery dialog in eXist's Java Admin Client tool

You can type queries (or open them from files), view the results, and even get a trace of their execution.

Updating Documents

A database wouldn't be of much use if you couldn't update its contents. Of course, you can always replace complete documents with new, updated ones, but for larger documents that's not very efficient. Therefore, XML databases—and eXist is no exception—provide mechanisms to update specific nodes within XML documents directly.

A document update in eXist has a relatively large overhead. It creates a transaction, updates the XML, and updates the relevant indexes. This makes updating expensive. For this reason, try to do as much as possible in a single update. For instance, creating a node and then adding its attributes all in separate updates is not a good idea. So, try to avoid this (we'll get to the exact syntax later):

```
let $elm as element() := doc('/db/Path/To/Some/Document.xml')/*
return (
  update insert <NEW/> into $elm,
  update insert attribute x { 'y' } into $elm/*[last()],
  update insert attribute a { 'b' } into $elm/*[last()]
)
```

Instead, create the node with its attributes as an XML fragment first and update your document in one go:

```
let $elm as element() := doc('/db/Path/To/Some/Document.xml')/*
return
  update insert <NEW x="y" a="b"/> into $elm
```

Or:

```
let $elm as element() := doc('/db/Path/To/Some/Document.xml')/*
let $new-element as element() := <NEW x="y" a="b"/>
return
  update insert $new-element into $elm
```

Also be aware that updating documents is not done in isolation. Updating something that another running XQuery script is using at the same time may lead to unanticipated results.

eXist has two mechanisms to directly update XML documents:

The eXist XQuery update extension
: This is an eXist-specific extension to XQuery (based on an early draft of the XQuery update specification) that allows you to write XQuery statements that alter the database. This is the preferred mechanism for performing in-place updates.

XUpdate
: You specify your desired alterations to the database in an XML document using the XUpdate syntax, and then pass it to the XUpdate processor to have them applied. The XUpdate specification is no longer maintained and so using this mechanism is discouraged, but it's still present mainly for backward compatibility reasons.

eXist's XQuery Update Extension

eXist defines its own XQuery language constructs to update documents in the database. These language constructs follow a proposal from Patrick Lehti.

To get a feeling for it, here is an example of an XQuery update adding a log message to an XML logfile:

```
update insert <LogEntry>Something happened...</LogEntry> into
   doc('/db/logs/mainlog.xml')/*
```

The following (somewhat more complicated) example makes sure that the number of messages does not exceed 10, and that the most recent message is on top:

```
let $document := doc('/db/logs/mainlog.xml')
let $newentry := <LogEntry>Something happened...</LogEntry>
return
  update delete $document/*/LogEntry[position() ge 10],
  if (exists($document/*/LogEntry[1]))
    then update insert $newentry preceding $document/*/LogEntry[1]
    else update insert $newentry into $document/*
```

All eXist XQuery update statements start with the keyword `update`, followed by an update action. Available actions are `delete`, `insert`, `rename`, `replace`, and `value`.

The return type of an update statement is always the empty sequence `()`.

An eXist update statement can only be used to update persistent XML documents stored in the database. It cannot be used to update temporary documents or document fragments stored in memory. For example, the following code fragment is illegal and will result in an error:

```
(: This is invalid: :)
let $document as document-node() := <Root><a/></Root>
return
  update insert <b/> into $document/*
```

You can use eXist's update statements anywhere in your XQuery main code or function bodies. However, take care when using them inside a FLWOR expression return statement. Update statements take effect immediately, and changing the structure your query is looping over may lead to some unexpected results!

update delete

`update delete` simply deletes nodes. Its syntax is:

```
update delete expr1
```

where `expr1` is an XQuery expression resolving to any kind of node. All nodes in `expr1` will be deleted by this statement.

Note that you cannot delete document root elements with `update delete`.

update insert

`update insert` inserts content into an element node. Its syntax is:

```
update insert expr1 [ into | following | preceding ] expr2
```

where `expr1` is an XQuery expression resolving to the content sequence to insert, and `expr2` is an XQuery expression resolving to the content sequence to insert *into*. It must resolve to one or more element nodes. If it contains more than one element node, the insertion takes place for all of them.

Where to insert is determined by the keywords `into`, `following`, and `preceding`:

`into`
: Appends the content in `expr1` after the last child element node of `expr2`

`following`
: Inserts the content in `expr1` immediately after the element node `expr2`

`preceding`
: Inserts the content in `expr1` immediately before the element node `expr2`

update rename

`update rename` renames nodes. Its syntax is:

```
update rename expr1 as expr2
```

Here, `expr1` is an XQuery expression resolving to element or attribute nodes. These are the nodes to rename. `expr2` is an XQuery expression. From the result of this expression, the string value of the first item is used as the new name.

Note that you cannot rename document root elements using `update rename`. Only nodes with a parent element node can be renamed.

update replace

`update replace` replaces element, attribute, or text nodes. Its syntax is:

```
update replace expr1 with expr2
```

where `expr1` is an XQuery expression resolving to a single element, attribute, or text node, and `expr2` is an XQuery expression. Rules and treatment depend on the type of `expr1`:

- When `expr1` is an element node, `expr2` must be an element node too.
- When `expr1` is an attribute or text node, the *value* of `expr1` is replaced by the concatenated string value of `expr2`.

As an example of the second case, the following update statement replaces the value of the `name` attribute on the root element of the given document with `aaabbb`:

```
update replace doc('/db/test/test.xml')/*/@name with
  <a>aaa<b>bbb</b></a>
```

Note that you cannot replace document root nodes using `update replace`. Only nodes with a parent element node can be replaced.

update value

`update value` updates the content of nodes. Its syntax is:

```
update value expr1 with expr2
```

Its functionality is equivalent to `update replace`, but it updates only values, not nodes.

XUpdate

XUpdate is an older mechanism to change the contents of the database in an indirect way. You first specify the required changes in an XML document using the XUpdate syntax. You then pass this document to an XUpdate processor that applies them. The full XUpdate specification can be found at *http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html*. Using the XUpdate mechanism is mostly discouraged, but it's still there for backward compatibility.

In Example 5-1, the XUpdate document inserts a log message in a logfile, before any existing ones.

Example 5-1. XUpdate document

```
let $xupdate-specification :=
  <xupdate:modifications version="1.0" xmlns:xupdate="http://www.xmldb.org/xupdate">
   <xupdate:insert-before select="/Log/LogEntry[1]" >
     <LogEntry>Something happened...</LogEntry>
   </xupdate:insert-before>
 </xupdate:modifications>
let $update-count := xmldb:update('/db/logs', $xupdate-specification)
return
  <p>The number of log entries added was {$update-count}</p>
```

XUpdate XML format

An XUpdate document is an XML document that contains one or more XUpdate XML fragments. An XUpdate XML fragment always uses the namespace `http://www.xmldb.org/`:

```
<modifications xmlns = "http://www.xmldb.org/xupdate"
               version = "1.0">
  ( insert-before | insert-after | append | update | remove )*
</modifications>
```

- The `version` attribute always has the fixed value `"1.0"`.

Inside a `modifications` element, you can specify the updates using certain XML elements (all of these elements have a `select` attribute that must contain an XPath expression evaluating to a node set):

`insert-before`, `insert-after`
: Inserts content before or after the node(s) specified in the `select` attribute

`append`
: Appends content as a child of the node(s) specified in the `select` attribute

`update`
: Updates the contents of the node(s) specified in the `select` attribute

`remove`
: Removes the node(s) specified in the `select` attribute

To specify new content, you can use a direct XML fragment as shown in Example 5-1 or use one of the following constructions (all elements in the `http://www.xmldb.org/xupdate` namespace):

`element name="..."`
: Creates an element node.

`attribute name="..."`
: Creates an attribute node. The contents of the element will become the attribute's value.

`text`
: Creates a text node. The contents of the element will become the node's value (a.k.a. the text).

The original XUpdate specification also mentions `processing-instruction`, `comment`, and using variables with `variable` and `value-of`. This is not supported in eXist.

Executing XUpdate

To execute an XUpdate document, eXist contains the following XQuery extension function:

```
xmldb:update($collection-uri as xs:string, $modifications as node())
as xs:integer
```

Here, `$collection-uri` is the collection the XUpdate is applied to. Notice that this is a *collection*, so the XUpdate is performed against *all* documents in the collection (and any subcollections, recursively). Make sure your XUpdate specification targets the right XML document(s) and/or design your collection structure carefully to constrain this.

`$modifications` contains your XUpdate document.

The function returns the number of updates applied.

Controlling the Database from Code

Learning how to control the database from XQuery code is important because when writing an application, sooner or later you'll want to create or delete documents and collections, change their properties, interrogate them, and so on. It's all part of the game.

Most functions for controlling the database are in eXist's `xmldb` extension module. To find the exact details of these functions, refer to the online eXist module documentation in the XQuery Function Documentation app in the dashboard. In this section, we will provide you with an overview.

This subject is somewhat dependent on that of security. For instance, if you try to create a document but the parent collection doesn't allow the current user to do so, you have a problem. Also, after you've created something you'll probably want to set its security properties (owner, group, permissions), right? Security is a big subject and is handled in depth in Chapter 8. This section provides you only with the basic information of how to work with the various security-related settings, not their meaning. In general, eXist's security system closely mimics that of UNIX systems.

Specifying Collections and Resources for the xmldb Extension Module

The `xmldb` extension module is somewhat fickle in how it handles addressing collections and resources:

- A collection must always be passed as a URL-escaped URI (of type `xs:string`), as specified in "Use URIs" on page 91. For instance:

  ```
  xmldb:get-child-resources("/db/new%20collection")
  ```

- A resource name can be passed as a URL-escaped URI, but surprisingly also as a normal, nonescaped string. Therefore, the following function calls are equivalent:

  ```
  xmldb:size("/db/new%20collection", "new document.xml")

  xmldb:size("/db/new%20collection", "new%20document.xml")
  ```

Accessing external databases using extended XMLDB URIs

The XMLDB extension function collection parameters all accept an extended syntax for the XMLDB URIs, as described in "XMLDB URIs" on page 92:

```
xmldb:exist://username:password@server:port/exist/xmlrpc/db/...
```

You can use this to access remote servers and update their databases directly.

Getting Information

There are many functions in eXist's `xmldb` extension module that can provide you with information about the database's content. Here is an overview of the most important ones:

`xmldb:last-modified`, `xmldb:size`, `xmldb:get-mime-type`
: Retrieve the basic properties of a resource.

> Most `xmldb` functions take two parameters—a collection and a document URI—but `xmldb:get-mime-type` is an exception (for unknown reasons). It takes the full URI to the document as its input.
>
> Additionally, you should note that `xmldb:size` will not give you the exact size of an XML resource, but an estimate based on the number of database pages the document occupies.

`xmldb:get-owner`, `xmldb:get-group`, `xmldb:get-permissions`
: Retrieve the security settings for a collection or resource.

  Permissions are returned as integer values; the function `xmldb:permissions-to-string` turns these into something more readable. It is strongly recommended instead to use the newer `sm:get-permissions` function from the Security Module instead.

`xmldb:get-child-collections`, `xmldb:get-child-resources`
: Provide you with sequences of the child collections or resources of a given parent collection. You can use these functions to traverse and inspect the database's collection/resource structure. Example 5-2 is a little XQuery program that displays the database's content. It uses a recursive function to traverse the collection tree, which is a pattern you'll see quite often in XQuery code.

Example 5-2. Traversing and displaying the database structure

```
xquery version "1.0" encoding "UTF-8";

declare option exist:serialize "method=html media-type=text/html indent=no";
declare function local:traverse-collection($collection as xs:anyURI,
    $indent as xs:integer) as element(p)*
{
    for $sub-collection in xmldb:get-child-collections($collection)
    return
    (
        <p style="margin-left: {$indent}pt"><b>{$sub-collection}</b></p>,
        local:traverse-collection(xs:anyURI(concat($collection, '/',
          $sub-collection)), $indent + 10),
        for $document in xmldb:get-child-resources($collection)
        return
            <p style="margin-left: {$indent + 5}pt">{$document}</p>
    )
};

<body>
{
    local:traverse-collection(xs:anyURI('/db/apps/exist-book'), 0)
}
</body>
```

Creating Resources and Collections

For creating resources and collections, the following functions are available:

`xmldb:create-collection`
: As its title implies, it creates a new collection in the database. It will return the path to the new collection when successful, or the empty sequence otherwise.

`xmldb:store`
: Creates a new resource, storing some data passed as a parameter. It will return the path to the new resource when successful, or throw an error when unsuccessful.

  Specifying what to store is quite flexible: you can pass data directly (for instance, as an XML fragment or a string), and it will be stored as the new resource. However, when the data is passed as type `xs:anyURI`, this is taken as the URI *to* the data and eXist will try to read it from there.

  There are two variants for this function: one where eXist will try to guess the Internet media type of the data to store, and one where you can explicitly specify this.

`xmldb:store-files-from-pattern`
: Bulk-loads files from the filesystem. There are several variants of this function that allow you more or less control over what is stored.

Setting Permissions

Every collection or resource that you create will be assigned the current user, group, and default security permissions. That might not be what you want, so it's quite common to change these after creating a collection or resource. Of course, there might also be other situations where you have to change some security setting. You can use the following functions to do this:

`xmldb:set-collection-permissions`, `xmldb:set-resource-permissions`
: Change the user, group, and/or permissions for a collection or resource.

  Notice that permissions must be passed in as integer values. The function `xmldb:permissions-to-string` might be of help here.

While the `xmldb:set-collection-permissions` and `xmldb:set-resource-permissions` functions are still available, they are in fact *deprecated* by the newer `sm:chmod` and `sm:chown` functions in the Security Manager module (see "Executing XQuery functions" on page 153).

Moving, Removing, and Renaming

The following functions can be used to move, rename, and remove collections and resources:

`xmldb:move`
: Moves collections and resources from one location (parent collection) to another

`xmldb:rename`
: Renames collections or resources

`xmldb:remove`
: Removes (deletes) collections or resources

Be warned, removing collections or resources is permanent: there is no such thing as a *trash can* collection in eXist!

CHAPTER 6

XQuery for eXist

Most of your work using eXist will be done in the XQuery programming language. This chapter covers what is and is not supported. It will also describe some eXist XQuery specifics, like controlling serialization and available pragmas.

eXist's XQuery Implementation

Currently, 2.0+ versions of eXist support almost the full XQuery 1.0 specification (as eXist has done for years) and quite a lot of XQuery 3.0. This section will provide you with the details.

XQuery 1.0 Support

eXist implements almost all of the full XQuery 1.0 specification, with the following exceptions:

- eXist's XQuery processor does not support the schema import and schema validation features. This is perfectly reasonable as they are defined as optional in the XQuery specification (`validate` and `import schema`). The database does not store type information along with the (values of) nodes; consequently it cannot know the typed value of a node and has to assume `xs:untypedAtomic`. This is compliant with the behavior defined by the XQuery specification.
- You cannot specify a data type in an element or attribute test. eXist suports the node test `element(test-node)`, but the test `element(test-node, xs:integer)` results in a syntax error.

The absence of the features does not mean that eXist is not type-safe; it is, very much so. It only means that type checking based on schema imports is not implemented.

eXist tested its implementation against the official XQuery Test Suite (XQTS version 1.0.2). Of the more than 14,000 tests, it passed over 99%.

eXist does not yet type-check the *name* of an element or attribute. So, strangely enough, you can write `let $elm as element(a) := <b/>` and eXist will find it absolutely OK, although this is a relaxation from the XQuery specification. The advice is not to use name tests in element or attribute data type specifications, though. So, use `element()` or `attribute()` instead of `element(a)` or `attribute(b)`, since specifying a name implies type checking that alas never occurs.

XQuery 3.0 Support

New since version 2.0 is eXist's support for XQuery 3.0. As of writing, this specification had reached Proposed Recommendation status and several partial implementations were available.

To enable the XQuery 3.0 support, start your XQuery program with:

```
xquery version "3.0";
```

XQuery 3.0 is a relatively new and probably not yet very well known standard. Therefore, the support eXist offers is handled in somewhat more detail next. For the exact details, please refer to the standard.

XPath 3.0 functions

Many of the extra functions defined in XPath and XQuery 3.0 are implemented. Among them are some very useful ones, like `format-dateTime`.

An exact list of what is available and what isn't can be found with the XQuery Function Documentation browser in the dashboard. Browse the `http://www.w3.org/2005/xpath-functions` module.

try/catch

The XQuery 3.0 `try/catch` mechanism allows you to catch errors raised during execution. These can be errors raised by the XQuery engine (meaning your code did something wrong), or errors you've explicitly raised with the `error` function. The following example shows a `try/catch` usage example that sets a variable to `-1` if a division-by-zero error occurs:

```
let $result as xs:decimal :=
  try
  {
    $something div $something-else
  }
  catch err:FOAR0001 { -1 }
```

This example tests for a very specific error, which is good, because we would like a warning when something unexpectedly goes wrong. If you want to test for *all* errors that can occur, change the `err:FOAR0001` into an `*`.

If you want to test for a specific error condition but don't know its code, probably the easiest way to find it is to force the error and copy the error code reported back into the XQuery.

Inside the `catch` operand you have access to information about the error through a number of implicitly declared variables—`$err:code`, `$err:line-number`, `$err:column-number`, `$err:description`, `$err:value`, `$err:module`, and `$err:additional`. Please refer to the XQuery 3.0 specification for full details.

switch expression

The XQuery 3.0 `switch` expression implements that which in other languages is often called a `case` expression or, in XSLT, an `xsl:choose`. This example was copied from the XQuery 3.0 specification:

```
switch ($animal)
   case "Cow" return "Moo"
   case "Cat" return "Meow"
   case "Duck" return "Quack"
   default return "What's that odd noise?"
```

Higher-order functions

A higher-order function is a function that takes another function as a parameter or returns a function. The normal use case for this is mapping or filter functions. Here is an example:

```
declare function local:map-example($func, $list) { ❶
    for $item in $list
    return
      $func($item)
};
```

```
let $f := upper-case#1 ❷
return
  local:map-example($f, ("Hello", "world!")) ❸
```

❶ We first define a function, `local:map`, that runs the function passed in its first parameter, `$func`, over all members of its second operand, `$list`.

❷ We then assign the `upper-case` function to `$f`. The `#1` after the function name means that we want the `upper-case` function with only one parameter (in case there are more).

❸ Finally, we call `local:map` with our function and some input strings. It returns the expected `HELLO WORLD!`.

Higher-order functions is a serious subject in its own right and includes topics such as: inline and partial functions, closures, currying, and more. For further information, you can refer to the XQuery 3.0 specification and to this excellent eXist wiki article.

The simple map operator

The XQuery 3.0 bang operator `!` (or simple map operator, as it is officially called) can be seen as a shorthand for simple FLWOR expressions. It applies the right-hand expression to each item in the sequence gained from evaluating the left-hand expression. For instance:

```
(1 to 10) ! . + 1
```

is the same as:

```
for $i in (1 to 10) return $i + 1
```

The string concatenation operator

The string concatenation operator `||` is a shorthand replacement for the `concat` function: it concatenates strings. For example, the following expression will be true:

```
'Hello '|| 'world' eq concat('Hello ', 'world')
```

Annotations

XQuery 3.0 allows annotations for functions and variables. This is used, for instance, to make them private (visible only in the enclosing module) or public:

```
declare %private variable $myns:only-i-can-see-this := 'secret';

declare
  %public
function myns:do-something-public() {
    (: some function body here:)

}
```

Within eXist, annotations are also used for RESTXQ (see "Building Applications with RESTXQ" on page 215).

Serialization

eXist now supports the new XQuery 3.0 manner of controlling serialization. For instance, this:

```
declare namespace output = "http://www.w3.org/2010/xslt-xquery-serialization";
declare option output:method "xml";
declare option output:media-type "text/xml";
```

is exactly the same as (eXist's incumbent nonstandard mechanism):

```
declare option exist:serialize "method=xml media-type=text/xml";
```

The options supported are the same also. More about serialization and the full list of options supported can be found in "Controlling Serialization" on page 119.

The group by clause

eXist has had an `order by` clause for its FLWOR expressions since 2006. Unfortunately, this was not compatible with the XQuery 3.0 `group by` clause, and so it was replaced in the 2.0 release with the official version. Here is an example:

```
let $data as element()* := (
  <item>Apples</item>,
  <item>Bananas</item>,
  <item>Apricots</item>,
  <item>Pears</item>,
  <item>Brambles</item>
)

return
  <GroupedItems>
  {
    for $item in $data
     group by $key := upper-case(substring($item, 1, 1))
     order by $key
     return
          <Group key="{$key}">
            {$item}
          </Group>
  }
  </GroupedItems>
```

The fruits are grouped and sorted based upon the uppercased first characters of their names. This returns:

```
<GroupedItems>
  <Group key="A">
    <item>Apples</item>
    <item>Apricots</item>
  </Group>
  <Group key="B">
    <item>Bananas</item>
    <item>Brambles</item>
  </Group>
  <Group key="P">
    <item>Pears</item>
  </Group>
</GroupedItems>
```

Other XQuery Extras

Beside eXist's support for XQuery 1.0 and the majority of XQuery 3.0, it also has a few interesting features which are currently specific to its XQuery implementation.

The map data type proposed for XQuery 3.1

The `map` data type in eXist is essentially a key/value lookup table. Keys must be atomic values (e.g., `xs:string`, `xs:integer`, `xs:date`, etc.). Values can be anything from a simple numbers to complete XML documents. Here is a basic example of creating and using a map:

```
let $map1 := map {
  "a" := 1,
  "b" := <XML>this is <i>cool</i></XML>
} return
  ($map1("a"), $map1("b"))
```

This will return:

```
1
<XML>this is <i>cool</i></XML>
```

Working programmatically with maps is possible through the `map` extension module. This module allows you to do everything from checking for the existence of keys up to constructing maps on the fly. Please refer to the online function documentation for more information.

Maps are immutable, like any other XQuery variables. So, changing a map using the functions from the `map` extension module (e.g., calling `map:remove`) will create a *new* map.

There is also an article about maps on the eXist wiki.

Java binding

eXist allows you to make arbitrary calls to Java libraries using the so-called *Java binding*. For example:

```
declare namespace javasystem="java:java.lang.System";
declare namespace javamath="java:java.lang.Math";

javasystem:getenv('JAVA_HOME'),
javamath:sqrt(2)
```

For security reasons, the Java binding is disabled by default. If you want to use it, edit *$EXIST_HOME/conf.xml*, search for the `enable-java-binding` attribute, set its value to `"yes"`, and restart eXist for the change to take effect.

There are some specifics you need to know about when using the Java binding:

- If the function name in XQuery contains a hyphen, the hyphen is removed and the character following it is converted to uppercase. So, a call in XQuery to `to-string` will call the Java method `toString`.
- Java constructors can be called using the `new` function.
- eXist adds a generic type, `object`, to its data-model, which is used for all Java objects.
- Instance methods of a class (methods that work on a specific object, like most of the Java methods) must get the object reference as their first parameter.
- When a method returns an array, it is converted to a sequence and you can iterate over it using a FLWOR expression.

Here is an example that will return a list containing the names of all files and subdirectories in the *$EXIST_HOME* directory:

```
declare namespace javafile="java:java.io.File";

let $fobject as object := javafile:new(system:get-exist-home())
return
  for $file in javafile:list($fobject)
  return
    $file
```

If you only want to get a list of files and directories, it is probably easier to use the `file` extension module instead of the Java binding.

XQuery Execution

There are some details you should be aware of regarding XQuery execution in eXist. These include:

Transaction boundaries

eXist is transactional only during updates to the database; that is, a single update either succeeds or fails atomically, not something in between, even if a crash occurs in the middle of the operation.

eXist is *not* transactional during the execution of a full XQuery script (like some other XQuery engines are). An XQuery script does not run in isolation, and updates made by it or by concurrently running neighbor scripts will immediately be visible. However, you can group updates into a single transaction; see the `exist:batch-transaction` pragma in "eXist XQuery Pragmas" on page 123.

Evaluation of expressions

eXist does *not* lazily evaluate expressions. For instance, a series of `let` expressions will all be evaluated, from top to bottom, even if some of the variables are never used again.

The reasoning behind this has to do with side effects, which XQuery officially doesn't have, but which (as we all know) in a real-world program are a necessity. For instance, when you have a function that adds to a logfile, you want it executed even if you don't do anything with its return value.

As a consequence, be careful computing expensive values that might never be used. It's better to either defer this until you really need them (e.g., nesting them inside an `if-then-else` structure) or do something along the lines of:

```
let $expensive-value :=
  if (...decide-whether-value-is-really-needed...)
    then compute-value...
    else ()
```

Serialization

Although it may seem as though eXist works directly with XML, as in "text with a lot of angle brackets," it does not. Internally, XML is represented as an efficient tree-structured data type. Only on the way out, in the final step, are the angle brackets added and the XML displayed once again as we know it. This process of changing the internal representation into something suitable for the outside world is known as *serialization*.

Controlling serialization is important: sometimes you may want XML, while at other times you want HTML and/or JSON. You may perhaps also want to set the Internet media type explicitly, or control indentation.

For the XSLT programmers among us who think this sounds familiar: you're right. In XSLT, serialization is controlled likewise through the `xsl:output` element.

Controlling Serialization

There are a number of ways you can control serialization from within your XQuery scripts:

`option exist:serialize`
: You can control serialization by adding a `declare option exist:serialize` statement to the XQuery prolog. For instance:

```
declare option exist:serialize
  "method=html media-type=text/html indent=no";
```

The contents of the `exist:serialize` option are a whitespace-separated list of name/value pairs, as described inthe following section. You do not have to define the `exist` namespace prefix. eXist automatically binds this to the appropriate `http://exist.sourceforge.net/NS/exist` namespace.

`util:get-option`, `util:declare-option`
: These extension functions allow you to inspect and set the value of an XQuery script option programmatically. For instance, setting the serialization options can be done with:

```
util:declare-option("exist:serialize",
  "method=html media-type=text/html indent=no")
```

XQuery 3.0 serialization settings
: eXist now also supports the standard XQuery 3.0 way of controlling serialization. This is described in "Serialization" on page 115.

Serialization Options

This section will list all the serialization options that eXist supports.

General serialization options

The more general serialization options closely mimic the options of the same name available on the XSLT `xsl:output` command:

`method=xml|microxml|xhtml|html5|text|json`
: Sets the principal serialization method.

The `microxml` method produces MicroXML as opposed to full XML. You can find out more about MicroXML from the W3C MicroXML Community Group.

The `xhtml` method makes sure that only the short form is used for elements that are declared empty in the XHTML specification. For instance, a `br` element is always returned as `<br/>`. In addition, if you omit the XHTML namespace from your XML, you can have the XHTML serializer inject it for you by setting the serialization option `enforce-xhtml=yes`.

If you specify the `text` method, only the atomized content of elements is returned: for example, `<foo>this is content</foo>` will return `this is content`. Namespaces, attributes, processing instructions, and comments are ignored.

For JSON and JSONP serialization options, see "JSON serialization" on page 121.

`media-type=...`
: Indicates the Internet media type of the output. This is used to set the HTTP `Content-Type` header if the query is running in an HTTP context.

`encoding=...`
: Specifies the character encoding used for serialization. The default is the encoding set in the XQuery declaration at the top of the program. If that is not set, the default is `UTF-8`.

`indent=yes|no`
: Indicates whether the output should be indented.

`omit-xml-declaration=yes|no`
: Specifies whether the XML declaration (`<?xml version="1.0"?>`) at the top of the output should be omitted.

`doctype-public=... doctype-system=...`
: When at least one of these is present, a `doctype` declaration is included at the top of the output.

`enforce-xhtml=yes|no`
: Forces *all* output to be in the XHTML (`http://www.w3.org/1999/xhtml`) namespace.

Post-processing serialization options

eXist can do post-processing of the XQuery result by processing `xi:include` elements and `<?xml-stylesheet?>` processing instructions referencing XSLT stylesheets. You can control this with the following options:

`expand-xincludes=yes|no`
: Indicates whether the serializer should process any `xi:include` elements (see "XInclude" on page 243). The default is `yes`.

`process-xsl-pi=yes|no`
: Indicates whether the serializer should process any `<?xml-stylesheet type="text/xsl" href="..."?>` processing instructions (see "Invoking XSLT by Processing Instruction" on page 242). The default is `yes`.

eXist-specific serialization options

eXist-specific options include the following:

`add-exist-id=element|all`
: If you output elements that come from the database, eXist will add an attribute `exist:id` to them, showing the internal node identifier of each element. Setting this option to `element` will show only the node identifier of the top-level element; setting it to `all` will show all node identifiers.

  There are functions in the `util` extension module to work with these identifiers.

`highlight-matches=both|elements|attributes|none`
: When querying text with the full-text or NGram extensions, the query engine tracks the exact position of all matches inside text content. The serializer can later use this information to mark those matches by wrapping them into an `exist:match` element. Find more information about this in "Locating Matches" on page 296.

JSON serialization

JSON (JavaScript Object Notation) is a lightweight data-interchange format. eXist has a JSON serializer built in that you can enable by setting the serialization method to `json` (see "General serialization options" on page 119). There is one related serialization option:

`jsonp=...`
: Produces JSONP (JSON with padding) output by wrapping the JSON output in the named function. For example, specifying `jsonp=abc` causes the output to be wrapped in the JavaScript function `abc` like so: `abc({"hello": "world"})`. This can be useful when you're working around *same-origin policies* in some web browsers.

  It is also possible to set the JSONP function dynamically by calling the function `util:declare-option` and passing in the function name; for example, `util:declare-option("exist:serialize", "method=json jsonp=myFunctionName")`.

Here is a summary of how eXist performs the JSON serialization (see also the wiki entry on this subject):

- The root element is absorbed: `<root>A</root>` becomes `"A"`.
- Attributes are serialized as properties, with the attribute name and its value.
- An element with a single text child becomes a property whose value is the text child: `<e>text</e>` becomes `{"e": "text"}`.
- Sibling elements with the same name within a parent element are added to an array: `<A><b>1</b><b>2</b></A>` becomes `{ "b" : ["1", "2"] }`.
- In mixed-content nodes, text nodes are dropped.
- If an element has attribute and text content, the text content becomes a property: `<A a="b">1</A>` becomes `{ "A" : { "a" : "b", "#text" : "1" } }`.
- An empty element becomes `null`: `<e/>` becomes `{"e": null}`.
- An element with name `<json:value>` is serialized as a simple value, not an object: `<json:value>my-value</json:value>` becomes `"my-value"`.

Sometimes it is necessary to ensure that a certain property is serialized as an array, even if there's only one corresponding element in the XML input, you can use the attribute `json:array="true|false"` for this.

By default, all values are strings. If you want to output a literal value—for example, to serialize a number—use the attribute `json:literal="true"`.

The JSON prefix `json` should be bound to the namespace `http://www.json.org`. As an example, here is some XML:

```
<Root xmlns:json="http://www.json.org">
  <Items>
    <Item id="1">Bananas</Item>
    <Item>CPU motherboards</Item>
  </Items>
  <Items >
    <Item json:array="yes">Bricks</Item>
  </Items>
  <Mixed>This is <i>mixed</i> content</Mixed>
  <Empty/>
  <Literal json:literal="yes">1</Literal>
</Root>
```

And here is its JSON serialization:

```
{ "Items" : [{ "Item" : [{ "id" : "1", "#text" : "Bananas" },
"CPU motherboards"] }, {"Item" : ["Bricks"] }],
"Mixed" : { "i" : "mixed" }, "Empty" : null, "Literal" : 1 }
```

In addition to the JSON serializer in eXist, which attempts to convert XML into JSON with as little effort from the developer as possible, there are three other XQuery modules that enable you to work with JSON. The first two modules—JSON XQuery (see json) and JSONP XQuery (see jsonp)—work in much the same way as the JSON serializer. The third module, XQJSON (see xqjson), which was written by John Snelson and adapted by Joe Wictenowski, is the newest JSON addition to eXist; it allows you to serialize XML to JSON as well as parse JSON back into XML so that you can round-trip your data.

Controlling XQuery Execution

There are several mechanisms which give you control over the execution of your XQuery scripts.

eXist XQuery Pragmas

With XQuery pragmas, you can set implementation-specific options for parts of your code. The general syntax is:

```
(# pragmaname #) {
  (: Your XQuery code block :)
}
```

eXist has the following pragmas:

`exist:batch-transaction`
: Provides a way to combine multiple updates on the database into a single transaction. Only works for updates done through eXist's XQuery update extension. For example:

```
(# exist:batch-transaction #) {
  update delete $document/*/LogEntry[position() ge 10],
  update insert $new-entry preceding $document/*/LogEntry[1]
}
```

`exist:force-index-use`
: Useful for debugging index usage (see Chapters Chapter 11 and Chapter 12). Will raise an error if there is *no* index available for the given XQuery expression. This can help you to check whether indexes are correctly defined.

`exist:no-index`
: Prevents the use of indexes on the given XQuery expression. Useful for debugging or for curiosity purposes ("How long does my query take without indexes?"). Also, sometimes it is more efficient to run without indexes than with —for instance, when a search isn't very selective.

`exist:optimize`
: Enables optimization for the given XQuery expression. If you've turned optimization off (with `declare option exist:optimize "enable=no";`, as discussed

in "Serialization Options" on page 119), you can turn it on again for specific expressions with this pragma.

`exist:timer`
: Measures the time it takes to execute the XQuery expressions within the pragma—for instance, `(# exist:timer #) { count(//TEST) }`.

  To see the timer, you need to enable tracing in the *$EXIST_HOME/log4j.xml* configuration file (set `<priority value="trace"/>` for the root logger). You'll see an entry like this in the *$EXIST_HOME\webapp\WEB-INF\logs\exist.log* file:

```
2012-09-12 15:01:29,846 [eXistThread-31] TRACE (TimerPragma.java [after]:63)
  - Elapsed: 171ms. for expression: count([root-node]/descendant::{}TEST)
```

Limiting Execution Time and Output Size

You can control execution time and query output size by adding the correct `declare option exist:...` statement to the XQuery prolog:

`declare option exist:timeout "time-in-msecs"`
: Indicates the maximum amount of time (specified in milliseconds) that a query is allowed to execute for. If this is exceeded, an error will be raised.

`declare option exist:output-size-limit "size-hint";`
: Defines a limit on the maximum size of created document fragments. This limit is an estimation, specified in terms of the accumulated number of nodes contained in all generated fragments. If this is exceeded, an error will be raised.

Other Options

Here are some miscellaneous options you can set by adding a `declare option exist:...` statement to the XQuery prolog:

`declare option exist:implicit-timezone "duration";`
: Specifies the implicit time zone for the XQuery context as defined in the XQuery standard. More information is available at *http://www.w3.org/TR/xquery/#dt-timezone*.

`declare option exist:current-dateTime "dateTime";`
: If for some reason you don't want to use your operating system's date/time, you can specify your own using this option (it is merely there to enable some of the XQuery test suite cases to run).

`declare option exist:optimize "enable=yes|no";`
: Use this to disable the query optimizer in eXist (the default is `yes`, of course). This is linked to the `exist:optimize` pragma; see "eXist XQuery Pragmas" on page 123.

XQuery Documentation with xqDoc

xqDoc is an effort to standardize XQuery documentation in a similar vein to how JavaDoc has for Java. xqDoc works by reading specialized comments you insert into your XQuery code. A parser can then use these to extract additional information about your module, its (global) variables, and its functions. This information could then, for example, be used to display details about a module to the user. The eXist function browser is a good example of an implementation which uses xqDoc to achieve exactly that.

Here is an example of a little module containing xqDoc information:

```
xquery version "1.0" encoding "UTF-8";
(:~
 :  Example module with xqDoc information
 :
 :  @version 1.0
 :  @author Erik Siegel
 :)
module namespace xquerydoc="http://www.exist-db.org/book/XQueryDoc";

(:~
 :  Example dummy function
 :
 :  @param $in The input to the function
 :)
 declare function xquerydoc:test($in as xs:string+) as xs:string
{
  'Dummy'
};
```

All comments starting with `(:~` are parsed by the xqDoc parser. Keywords in these comments start with an `@` character. The exact syntax can be found on the xqDoc website.

eXist has an `inspect` extension module to work with xqDoc. The functions in this module return an XML representation of the module's content, including possible annotations by the xqDoc comments. For instance, running `inspect:inspect` on the preceding example module returns:

```
<module uri="http://www.exist-db.org/book/XQueryDoc" prefix="xquerydoc">
  <description> Example module with xqDoc information </description>
  <author> Erik Siegel </author>
  <version> 1.0 </version>
  <variable name="xquerydoc:global" type="xs:string" cardinality="exactly one"/>
  <function name="xquerydoc:test"
    module="http://www.exist-db.org/book/XQueryDoc">
    <argument type="xs:string" cardinality="one or more" var="in">
      The input to the function</argument>
    <returns type="xs:string" cardinality="exactly one"/>
    <description> Example dummy function </description>
  </function>
</module>
```

From this XML you could easily create any required HTML or PDF documentation.

eXist does not support the full xqDoc specification. If you need some specific xqDoc feature, please run some tests to see if it is present.

CHAPTER 7

Extension Modules

eXist has a large number of extension modules built in that allow you to do lots of wonderful things that you can't do with straight XQuery, such as manipulating the database's content, inspecting an HTTP request, encrypting/decrypting data, and performing XSLT transformations. This chapter will provide you with an overview of the module mechanism in general. An overview of the available modules can be found in Appendix A.

Types of Extension Modules

Extension modules come in two flavors, which you need to be aware of because they behave differently in eXist. The module descriptions in Extension Module Descriptions will tell you the module type. We will also explain how to find out the module type yourself in "Enabling Extension Modules" on page 128.

Extension Modules Written in Java

Extension modules written in Java are fully integrated into eXist. You don't even have to put an `import module` statement in your XQuery code to be able to use them. Most of the eXist core functionality you use regularly is Java-backed (e.g., `xmldb`, `request`, `transform`, `response`, and `util`).

For instance, when you want to find out the eXist home directory by calling `system:get-exist-home`, you can simply do this in your code without an `import module` statement for the `system` module. However, it is generally considered good practice to have this explicitly set out, so you may choose to add, for example:

```
import module namespace system="http://exist-db.org/xquery/system";
```

Not all of the available Java modules are enabled by default. Even worse, some of them are not even built in (compiled and linked) to the default eXist configuration! How to find out which are built in and/or enabled, and what to do about it is covered in "Enabling Java Extension Modules" on page 129. Writing your own extension modules in Java is covered in "Internal XQuery Library Modules" on page 467.

If you use eXide, oXygen, or another eXist-aware IDE, there is a very easy way to find out whether a Java-based extension module is enabled. While editing an XQuery script, type the prefix of the module and press the shortcut keys that pop up autocompletion suggestions (usually Ctrl-space bar). If a list of functions appears, the module is enabled.

Extension Modules Written in XQuery

Some extension modules are written in XQuery. To use them, you need to explicitly import the module by placing an `import module` statement in your XQuery code. You need not specify the `at` clause because eXist already knows where to find them.

For instance, to use the KWIC (see "Using Keywords in Context" on page 297) extension module, you would add the following to your XQuery prolog:

```
import module namespace kwic="http://exist-db.org/xquery/kwic";
```

XQuery-based modules can be disabled too. However, in contrast to Java-based modules, this only means eXist doesn't know their location and you have to add an `at` clause to your `import module` statement if you still want to use them. Read more about this in "Enabling XQuery Extension Modules" on page 130.

Enabling Extension Modules

This section will tell you how to enable an extension module. There are two files/locations that are important here:

$EXIST_HOME/conf.xml

Search this file for the `builtin-modules` element. Within you'll find `module` elements, some of which are commented out. As you might have guessed, enabling a module here means removing the comment from the appropriate `module` element (and restarting eXist).

The `module` elements also show you the type of the module (Java or XQuery). Java modules have a `class` attribute. For instance:

```
<module uri="http://exist-db.org/xquery/xmldb"
        class="org.exist.xquery.functions.xmldb.XMLDBModule" />
```

But XQuery modules have a `src` attribute:

```
<module uri="http://exist-db.org/xquery/kwic"
        src="resource:org/exist/xquery/lib/kwic.xql" />
```

If there is a `resource:` prefix it indicates that the module is located within a package inside one of eXist's JAR files.

$EXIST_HOME/extensions/build.properties
For some Java modules (and some additional features), you have a choice of whether to include them in the eXist build. The default settings for this are in *$EXIST_HOME/extensions/build.properties*. More information can be found in the next section.

Enabling Java Extension Modules

To use a Java extension module, two conditions must be met:

- It must be part of the eXist build. You can determine this as follows:
  - Open *$EXIST_HOME/extensions/build.properties* (this is a text file, not an XML document), and see if the module or feature you want to use is mentioned there. For instance, the `memcached` extension module has this line:

    ```
    include.module.memcached = false
    ```

  - If your module is not mentioned in *build.properties*, you don't have to worry about it being part of the build; that will always be the case.
  - If your module is mentioned in *build.properties* and its `include.module` entry is set to `true`, it will be part of the build.
  - If your module is mentioned in *build.properties* and its `include.module` is set to `false`, it will not be part of the build. To enable it, you will have to change this and rebuild eXist. The following subsection explains how to do this.
  - If you (or somebody else working on the same eXist installation) has already changed some of the settings in *build.properties*, there will be a *local.build.properties* file also. Settings in this file override the settings in *build.properties*, so you have to check this too.
- It must be enabled in the built-in modules list in *$EXIST_HOME/conf.xml*. To determine this, do the following:
  - Open *$EXIST_HOME/conf.xml* and search for the modules list in the `builtin-modules` element.
  - If the child `module` element for the module you're looking for is commented out, the module is disabled. Remove the comment and restart eXist to enable it.

Rebuilding eXist

If you've found out that the module you needed was not part of the eXist build because its `include.module` entry was set to `false` in *$EXIST_HOME/extensions/build.properties*, here is how to change it:

1. Install (if it's not on your system already) the Java SE JDK—that is, the development kit, not the runtime environment. For instructions and downloads, refer to *http://www.oracle.com/technetwork/java/javase/overview/index.html*.
2. Make sure you have an environment variable called `JAVA_HOME` pointing to the root directory of your Java installation. If it does not exist, create it.
3. If you haven't done so before, copy *$EXIST_HOME/extensions/build.properties* to *local.build.properties* (in the same directory).
4. Edit *local.build.properties* and set the `include.module` entry for the module you want to enable to `true` (for instance, `include.module.xslfo = true`).
5. Stop eXist.
6. Open a command window, navigate to *$EXIST_HOME*, and issue the appropriate build command:

    - On Unix-based systems: `./build.sh extension-modules`
    - On Windows-based systems: `build.bat extension-modules`

    The build will run. If you examine the output you should see some messages scroll by regarding your module.
7. Since eXist is still stopped at this point, enable the corresponding `module` element in the built-in modules list in *$EXIST_HOME/conf.xml* now.
8. Restart eXist.

Now the module is enabled. This not only means that you can use its functions from within your XQuery code, but also that it should show up in the eXist function documentation browser (after a regeneration).

Enabling XQuery Extension Modules

An XQuery extension module is not so different from an XQuery module that you could write yourself. The real difference is that the extension modules are a part of the main eXist distribution and are included in one of eXist's JAR files that is present on the classpath. Paths to a resource on the classpath in *$EXIST_HOME/conf.xml* are prefixed with `resource:`, like so:

```
<module uri="http://exist-db.org/xquery/kwic"
        src="resource:org/exist/xquery/lib/kwic.xql" />
```

The manifest difference between an XQuery extension module provided on the class-path and your own extension modules is that eXist already knows where to find them, so you don't have to add an `at` clause to the `import module` statement when using a provided module. So, to use the `kwic` module, you can write:

```
import module namespace kwic="http://exist-db.org/xquery/kwic";
```

However, if you want to, you can also specify it in full, in which case the built-in modules list in *$EXIST_HOME/conf.xml* is not even used:

```
import module namespace kwic="http://exist-db.org/xquery/kwic"
  at "resource:org/exist/xquery/lib/kwic.xql";
```

Specifying an XQuery-based module in the built-in modules list only tells eXist where to find it, nothing more. It actually has nothing to do with enabling or disabling a module, since you can always reference it using an explicit `at` clause.

By the way, this also means that you can add your own modules to the built-in modules list by specifying their full paths; for example:

```
<module uri="http://www.mycompany.org/modules/helper"
        src="xmldb:exist:///db/myapp/lib/helper.xq" />
```

However, the advice is not to do this. Keep the built-in modules list for, well, the built-in modules, and don't litter it with your own stuff, as it will make upgrading in the future more tedious. In any case, changes to the built-in modules list only take effect after a restart.

CHAPTER 8
Security

eXist integrates a comprehensive and flexible security subsystem within the core of the database that cascades up through each API and web server. It is impossible to access any resource or collection within eXist without authorization or access rights being granted to the resource.

eXist at its simplest uses classic username and password credentials for authentication. The essence of its security model was very much inspired by the Unix permissions model. Permissions are applied at a resource level, and each resource and collection in the database must have Unix-style permissions assigned to it; these are validated when the resource or collection is accessed.

The Unix-style security model in eXist is adequate for many applications, but it does not scale well when you have hundreds of users with different roles. While you can solve this by creating many groups containing many permutations of user accounts, this quickly becomes unmanageable, and if you cannot understand your own security model you have little chance of asserting its integrity. For larger uses, eXist supports ACLs (access control lists), which allow you to place many modes for different users and groups onto the same document or collection (see "Access Control Lists" on page 156). eXist does not yet natively implement RBAC (role-based access control), but it's not too hard to add this at your application layer as an organization of ACLs.

eXist's Security Manager also permits pluggable modules that provide an authentication realm, and through this mechanism eXist can authenticate against external providers to better integrate with your existing infrastructure. eXist provides a default internal realm in which user and group credentials are stored in a set of XML documents within a special collection in the database, */db/system/security/exist*.

Security Basics

The basic authentication model in eXist follows the Unix model of having users and groups of users. eXist does not support groups of groups. Each resource and collection in the database is assigned an owner user, group, and mode. The mode describes the access permissions that the owner, user group, and other users have to that resource or collection.

User and group names in eXist are case-sensitive, so, for example, the username `James` is not the same as `james`.

Out of the box, eXist's internal authentication realm provides you with some default users and groups to get you started.

Users

Table 8-1 outlines the default users provided with eXist out of the box.

Table 8-1. Default users

Username	Description
`guest`	The `guest` user represents unauthenticated users. Until a user authenticates with eXist, she is a guest. It is possible to allow users access to some resources as `guest` without authentication; this is particularly useful for serving content to web users without them having to log in to your website. The `guest` user has a default auto-set password of `guest`, although you should never need it.
`admin`	The `admin` user is the default `dba` (database administrator) user for eXist, and will be the first user that you log in as after installing eXist. By default, the `admin` user's password is empty.
SYSTEM	The `SYSTEM` account is used internally by eXist processes to modify resources in the database and manage the database. Even eXist has to authenticate itself! You cannot authenticate as the `SYSTEM` user, and eXist cannot function without that account.

You really should consider setting a strong password for the `admin` user to secure the system, either during or immediately after installing eXist. What constitutes a strong password? Well, that's hard to explain simply, and advice tends to change over time, but this website can help you generate strong passwords: *http://strongpasswordgenerator.com*. If you're more security conscious, check out *https://www.grc.com/passwords.htm*.

Each user in eXist must belong to at least one group, and may belong to many groups. If a user is a member of many groups, then the default group for ownership of resources or collections created by a user is that user's *primary group*. The primary group, by default, is the first group that a user is added to, but this can be reconfigured later as the user is added to further groups.

For those wishing to operate a secure environment, it is recommended that you create your own `admin` user with a different username and place it in the `dba` group. You can then log out and log in as your new `admin` user and disable the default `admin` user account through the Java Admin Client's User Manager.

Groups

Table 8-2 outlines the default groups that eXist provides out of the box.

Table 8-2. Default groups

Group name	Description
`guest`	The `guest` group can be thought of as a group representing unauthenticated users. It is really present to support the `guest` user.
`dba`	The database administrator (`dba`) group is all-powerful: if you are in this group, then there is nothing that you cannot do with eXist. This is akin to the `root`/`toor`/`wheel` group in a lot of Unix systems, or the `Administrators` group in Windows systems.

You should give great consideration to adding a user to the `dba` group. Often, you need only a single user in the `dba` group. It is better to create your own `admin` groups that have more limited permissions on resources in the database.

In addition, you should never attempt to delete the `guest` or `dba` groups from eXist, as they are required for the proper functioning of the system. If you wish to prevent anonymous access to eXist, you can disable the `guest` account, using the same method described in the hint for the `admin` user in the previous section.

Permissions

As well as an owner user and group, each resource and collection in eXist has a permissions *mode*, which is expressed in the same way in eXist as in Unix systems. The mode is made up of three user classes:

- Owner user
- Owner group

- Other users

Each class consists of three mode bits (or flags, if you like) that describe whether read (`r`), write (`w`), and/or execute (`x`) access is permitted for that class.

In Figure 8-1, we can see that the owner user class of the */db* collection has read, write, and execute access; the owner group class has just read and execute access; and the other users class also has just read and execute access.

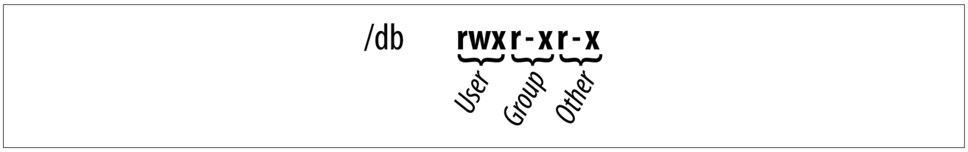


Figure 8-1. Example permission classes on the /db collection

So what do these read, write, and execute bits mean? Well, they are interpreted differently for resources and collections, as outlined in Table 8-3.

Table 8-3. Mode bits in eXist

Bit	Meaning
Resources	
Read	The user class has *read access* to the content and metadata of the resource.
Write	The user class has *write access* to the content and metadata of the resource.
Execute	If the resource is an XQuery module, then it can be *executed* by the user class.[a] If it is not an XQuery module, then this bit is ignored.
Collections	
Read	The user class may *list* the contents of the collection.
Write	The user class may *write* to the collection; this includes adding and deleting resources or subcollections to and from this collection.
Execute	The user class may *open* this collection.

[a]Currently in eXist 2.0 and 2.1, both execute and read access are required (not just execute access) in order for a user to execute a stored XQuery module. This limitation persists because the XQuery interpreter in eXist operates entirely within the permissions of the user invoking the XQuery; in future versions of eXist it is likely that this limitation will be lifted.

If a user class does not have execute permission on a collection, then the collection is closed to it, and the user(s) in question may neither list the collection's contents, nor read or write documents in that collection.

If you unset the read bit on a collection and set the execute bit, this allows a user or group to read resources in the collection, but not to list the contents of the collection. In effect, users can only access a resource in that collection if they know the resource's name in advance and have appropriate permissions on it. If the collection is accessed from the REST Server, it effectively disables collection listings.

Internally in eXist, the mode is held as a series of binary bits, but they are typically represented in one of two ways: either as an octal number or as a mode string (as you have just seen). eXist supports both approaches and provides assistance for converting from one to the other (see Table 8-4).

Table 8-4. Mode bits

Character representation	Octal number representation	Binary representation
Read (`r`)	`04`	`100`
Write (`w`)	`02`	`010`
Execute (`x`)	`01`	`001`

When octal numbers are used, the sum of each user class is placed side by side. For example, `rwxrwxrwx` would equate to:

04 + 02 + 01 = 07

for each user class (i.e., `0777`). As another example, `0744` would equate to `rwxr--r--`.

eXist provides XQuery functions for converting between integer and octal values in the Util module (see `util`), while several of the security functions in eXist can use either octal or character representations as arguments.

If you are browsing the database contents through either the Java Admin Client or the dashboard Collections app, then you may notice that some permissions strings are prefixed with either a `c` or `-` character. The `c` prefix stands for collection and the `-` prefix stands for resource (i.e., not a collection). This is similar to performing an `ls -la` command on a Unix-like system, except that whereas there `d` denotes a directory, eXist has collections instead of directories and therefore uses `c`, not `d`. Likewise, if you see a `+` character on the end of a permission string when browsing the database, this implies that the permission incorporates an access control list (see "Access Control Lists" on page 156).

Default Permissions

eXist will apply a default set of permissions to the database when it is first created, and then also to new resources and collections as they are created by users in the database (see Table 8-5).

Table 8-5. Default permissions

Thing	Owner user	Owner group	Mode
/db	`SYSTEM`	`dba`	`rwxr-xr-x` (0755)
/db/system	`SYSTEM`	`dba`	`rwxr-xr-x` (0755)
/db/system/config	`SYSTEM`	`dba`	`rwxr-xr-x` (0755)
/db/system/plugins	`SYSTEM`	`dba`	`rwxrwx---` (0770)
/db/system/security	`SYSTEM`	`dba`	`rwxrwx---` (0770)
New resource	Logged-in user	Logged-in user's primary group	`0666 - umask`[a]
New collection	Logged-in user	Logged-in user's primary group	`0777 - umask`
User mask	-	-	`022`

[a]The concept of the *umask*, or *user mask*, will be explained in the following section.

You might be asking yourself, if the */db* collection is only writable by the `SYSTEM` user by default, how is it that the `admin` user, which eXist creates by default, can write to the database?

The answer is that the `admin` user is a member of the `dba` group, which is *all-powerful*. The permissions mode is not checked for `dba` users.

You may be wondering why some of eXist's collections have a default permissions mode of `0775` and others have a permissions mode of `0770`. Put simply, the collections with mode `0770` have higher security concerns—for example, the configuration of security realms, user accounts, and groups is kept under the */db/system/security* collection, and it is undesirable to allow non-`dba` users access to this collection.

User masks

In eXist-db, user accounts can be assigned a mask, just like in Unix; this is known as a *umask*. The umask adjusts the permissions applied to new resources and collections created by that user. The effective permissions applied at creation time are calculated by taking the default permissions and subtracting the umask of the user creating the resource or collection.

The default umask in eXist is `022`, but this is configurable for each user account. Table 8-6 shows some examples of how the effective permissions are calculated when a new resource or collection is created.

Table 8-6. Effective default permissions

For	Permissions	umask	Applied permissions
New resource	`0666` (default)	`022` (default)	`0666` - `022` = `0644` (`rw-r--r--`)
New collection	`0777` (default)	`022` (default)	`0777` - `022` = `0755` (`rwxr-xr-x`)

So what does this all mean? Quite simply, that by default:

- For new resources, the owner can only read and write (this prevents accidentally granting execution permission on XQuery resources to unintended users). The group and other users can only read the resource.
- For new collections, the owner can open the collection, list the contents, and add or remove resources and subcollections. The group can open the collection and list the contents, but cannot add or delete. Other users, similar to group users, can open the collection and list the contents, and also cannot add to or delete from it.

As the default resource permissions are `0666`, and with the default umask applied they are `0644`, XQueries that you store into the database are not executable by default. This is an intentional decision by the eXist developers, made for two reasons: 1) it follows the Unix security model; and 2) database administrators *should* be aware of which XQueries are executable and by whom, thus forcing them to enable execution of an XQuery encourages such a mindset.

Many users ask, "How can I set all XQueries to be executable?" This is achieved relatively easily by creating an XQuery (which you can run once) that uses the functions in the `xmldb` XQuery module to enumerate the XQueries stored in the database, and the Security Manager XQuery module to set the permissions of those XQueries (see `xmldb` and `sm` in Appendix A).

Managing Users and Groups

In eXist the creation of users and groups is also restricted by permissions, and there are differences between the required permissions for each. Table 8-7 shows who is able to make modifications to users and groups.

Table 8-7. Permissions to modify principals

Action	User modification requirement	Group modification requirement
Creation	• Member of the `dba` group	• Member of the `dba` group
Modification	Either: • Target user • Member of the `dba` group	Either: • Group manager • Member of the `dba` group
Deletion	Either: • Target user • Member of the `dba` group	Either: • Group manager • Member of the `dba` group

Group Managers

When a group is created in eXist it has a single member, which is the user who created it. A group is typically used to model collaboration between users on resources in the database. As you build up more and more users and groups, it becomes necessary to share the administration of these groups of users.

eXist extends the Unix model for user groups, and introduces the concept of *group managers*. Members of a group in eXist may be promoted to managers of that group; in fact, the first member of any group (i.e., the creating user) is automatically set up as the first group manager.

Group managers share control of the group, and may perform several actions upon the group:

- Adding or removing members to or from the group, including other group managers
- Promoting or demoting members of the group to or from group managers
- Modifying the metadata of the group
- Deleting the group, and therefore removing all members from the group

As group managers maintain complete control of the group and its membership, it is important that they all share the same trust relationship. If you were a group manager, it would be inadvisable to promote someone else to a group manager if you did not trust her, as she could effectively remove you and take control of the group.

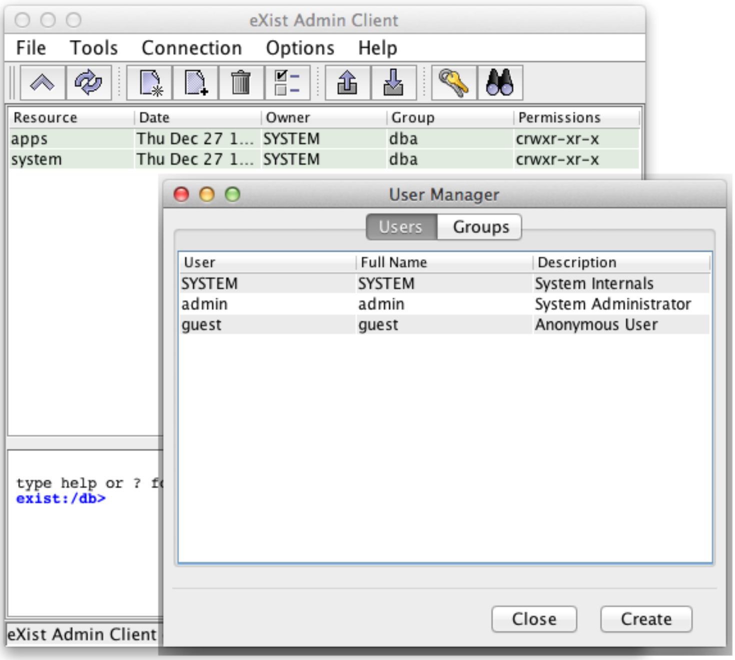
Tools for User and Group Management

So now that you have a good grounding in the basic security concepts involved in eXist, how do you actually apply these by creating and managing users and groups?

In eXist there are often several ways to achieve the same goal, depending on how you want to approach the problem. Security configuration is no exception: there are at least five possible ways to manage users and groups in eXist.

Using the Java Admin Client

eXist ships with an admin client application written in Java (see "The Java Admin Client" on page 29) that provides an excellent user management facility to make working with users and groups simple. This is the tool we will focus on in "User and Group Management with the Java Admin Client" on page 145. The User Manager is available via the Tools menu in the Java Admin Client, as you can see in Figure 8-2.



Resource	Date	Owner	Group	Permissions
apps	Thu Dec 27 1...	SYSTEM	dba	crwxr-xr-x
system	Thu Dec 27 1...	SYSTEM	dba	crwxr-xr-x

```
type help or ? fc
exist:/db>
```

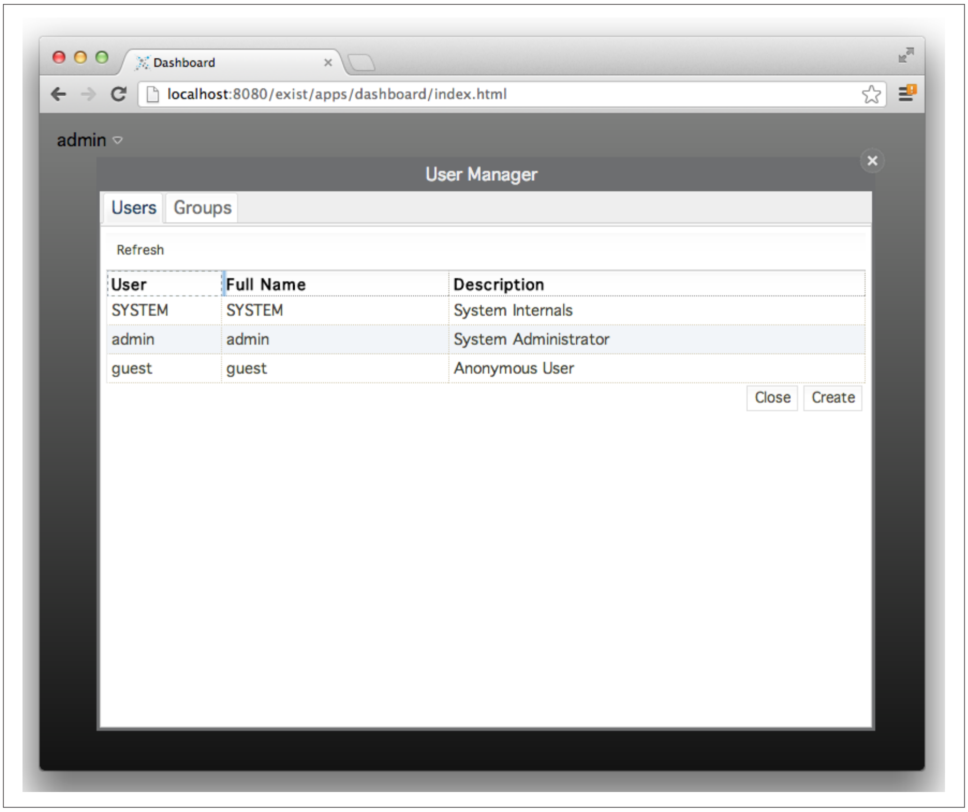
User Manager

User	Full Name	Description
SYSTEM	SYSTEM	System Internals
admin	admin	System Administrator
guest	guest	Anonymous User

Figure 8-2. The Java Admin Client User Manager

Using the User Manager web app

eXist provides a User Manager app out of the box as part of its dashboard (see Figure 8-3). This app provides identical functionality to the User Manager in the Java Admin Client and looks almost identical. We have chosen not to focus on this, as having such web apps installed may be undesirable in a production server environment (see "Removing preinstalled EXPath packages" on page 184).



User	Full Name	Description
SYSTEM	SYSTEM	System Internals
admin	admin	System Administrator
guest	guest	Anonymous User

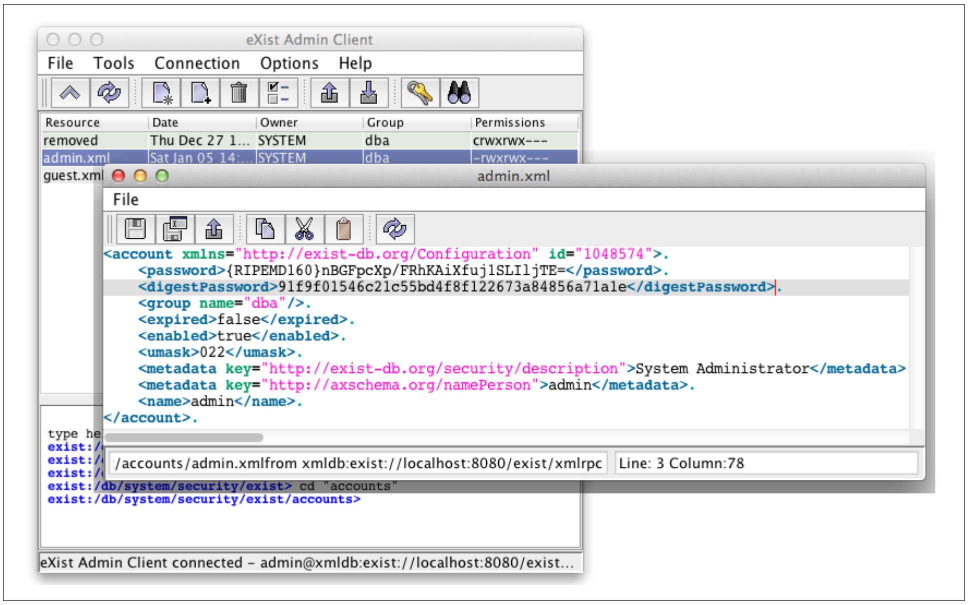
Figure 8-3. The dashboard User Manager app

Executing XQuery functions

eXist provides an XQuery module called the *Security Manager*. This module has many functions that can be called from your own XQuery modules for programmatically managing security within the database; see `sm`.

Modifying the security collection

All user and group information in eXist is kept in a series of XML documents within the database (see Figure 8-4). The system collection */db/system/security* contains all user and group configuration for all authentication realms known to eXist. The eXist internal authentication realm stores its data in */db/system/security/exist*, which has two subcollections: *accounts* and *groups*. Unsurprisingly, each XML document in these collections represents a single user or group within the system.



Resource	Date	Owner	Group	Permissions
removed	Thu Dec 27 1...	SYSTEM	dba	crwxrwx---
admin.xml	Sat Jan 05 14:...	SYSTEM	dba	-rwxrwx---
guest.xml				

```
<account xmlns="http://exist-db.org/Configuration" id="1048574">.
    <password>{RIPEMD160}nBGFpcXp/FRhKAiXfuj1SLIljTE=</password>.
    <digestPassword>91f9f01546c21c55bd4f8f122673a84856a71a1e</digestPassword>.
    <group name="dba"/>.
    <expired>false</expired>.
    <enabled>true</enabled>.
    <umask>022</umask>.
    <metadata key="http://exist-db.org/security/description">System Administrator</metadata>
    <metadata key="http://axschema.org/namePerson">admin</metadata>.
    <name>admin</name>.
</account>.
```

/accounts/admin.xmlfrom xmldb:exist://localhost:8080/exist/xmlrpc   Line: 3 Column:78

```
type he
exist:/
exist:/
exist:/
exist:/db/system/security/exist> cd "accounts"
exist:/db/system/security/exist/accounts>
```

eXist Admin Client connected – admin@xmldb:exist://localhost:8080/exist...

Figure 8-4. Example of the XML document for the eXist admin user

eXist has a dynamic configuration system, which means that when you modify user or group information, the XML documents in the relevant collection are updated, and vice versa; that is, if you were to modify these XML documents, the system would load the changes to the users and groups immediately. Modifying these documents directly is generally discouraged, as their syntax is subject to change at any time, and any mistakes made in the syntax could upset the stability of the system or your ability to access it. However, while this approach is not generally recommended, when used carefully, it does provide a convenient mechanism for those accessing eXist via WebDAV (see "WebDAV" on page 305) to create, modify, and delete users and groups simply by adding, updating, or removing XML documents in these collections. Should you opt for this approach, you must ensure that you do not change the `id` attribute of an account or group, as these are internal pointers within eXist. In addition, if you create a new account or group, you must be certain that you assign it the next free account or group ID.

Using the APIs: XML-RPC, XML:DB, and SOAP

eXist has several APIs for programmatically connecting to the database remotely (or even within an embedded JVM, if you're using XML:DB) and managing users and groups. The use of these APIs is discussed more thoroughly in Chapter 13.

User and Group Management with the Java Admin Client

In this section we provide a practical walkthrough and explanation of creating a group and subsequently a user. We also deal with group membership and group managers.

Scenario

In our fictitious organization, Ficto Ltd., a new team has been formed to manage content about animal welfare, which Ficto wishes to store, transform, and publish as part of its veterinary journal. The team is expected to grow in the near future. We need to add new accounts for the four team members, James Smith, Joe Brown, Helen Finkle, and Laura Laurence. Laura and James are in charge of the new team.

We will create a security group for the team, create accounts for each member of the team, and set up James and Laura as group managers, so that they can add any future team members to the group.

Creating a Group

First, we need to create our group for the Animal Welfare Content Team.

Launch the Java Admin Client, log in to the database, open the Tools menu, and select Edit Users (Figure 8-5).

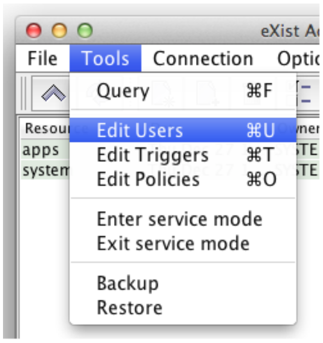
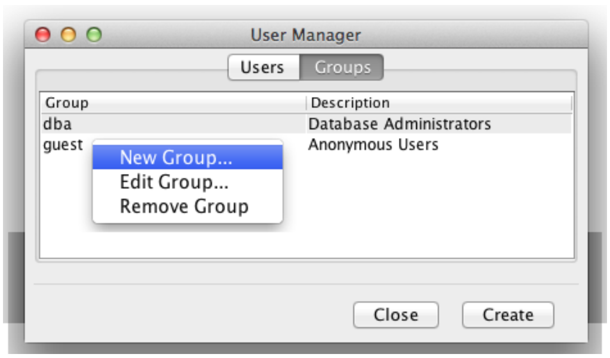


Figure 8-5. Launching the User Manager

Once in the User Manager, click the Groups tab. Then you can either right-click on an existing group in the list to get a pop-up menu and click the New Group menu item, or simply click the Create button at the bottom of the User Manager (Figure 8-6).

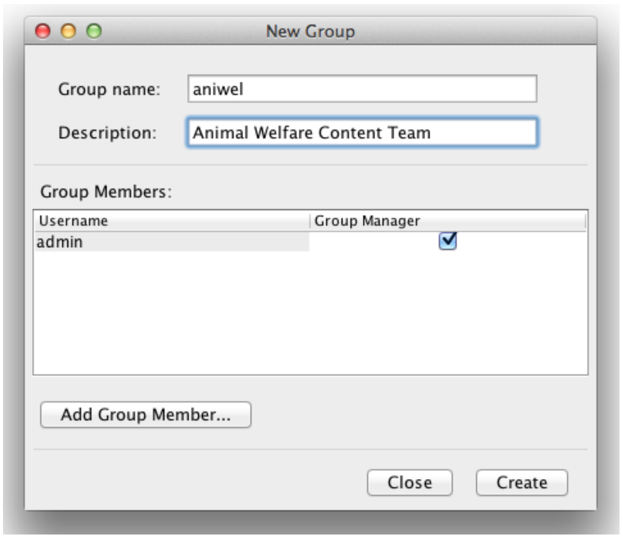


Group	Description
dba	Database Administrators
guest	Anonymous Users

Figure 8-6. Accessing the New Group dialog

When the New Group dialog appears, you need to complete the form and click the Create button. Keep the name of the group simple and remember that group names are case-sensitive in eXist. We recommend that you use short group names in lowercase. Regardless of the scheme you decide on, you cannot use punctuation or whitespace when naming a group.

As with the User Manager dialog, you can right-click on the Group Members list to add and remove group members and also to promote or demote them to or from group managers (Figure 8-7). By default, the creator of the group will be a member and manager of the group he creates.



Group name: aniwel

Description: Animal Welfare Content Team

Group Members:

Username	Group Manager
admin	✓

Figure 8-7. Creating a group

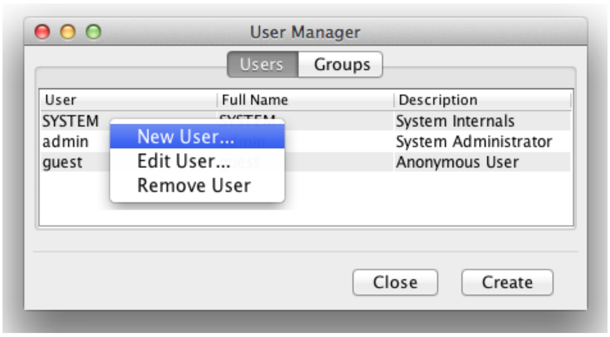
Once you have completed the form, click the Create button. The group will be created and you will be returned to the User Manager Groups list.

You can associate simple descriptive metadata (such as a description or email address) with groups and accounts in eXist, which can be useful for searching or managing them. The User Manager in the Java Admin Client and the dashboard both limit the metadata to just a description (and a name for accounts), but you can set and retrieve additional metadata properties using the XQuery functions in the Security Manager module, as discussed in `sm`.

Creating Users

Now we need to create user accounts for each member of the Animal Welfare Content Team and add them to the group we created for the team.

From the User Manager, click the Users tab; and then either right-click on an existing user in the list to get a pop-up menu and click the New User menu item, or simply click the Create button at the bottom of the User Manager (Figure 8-8).

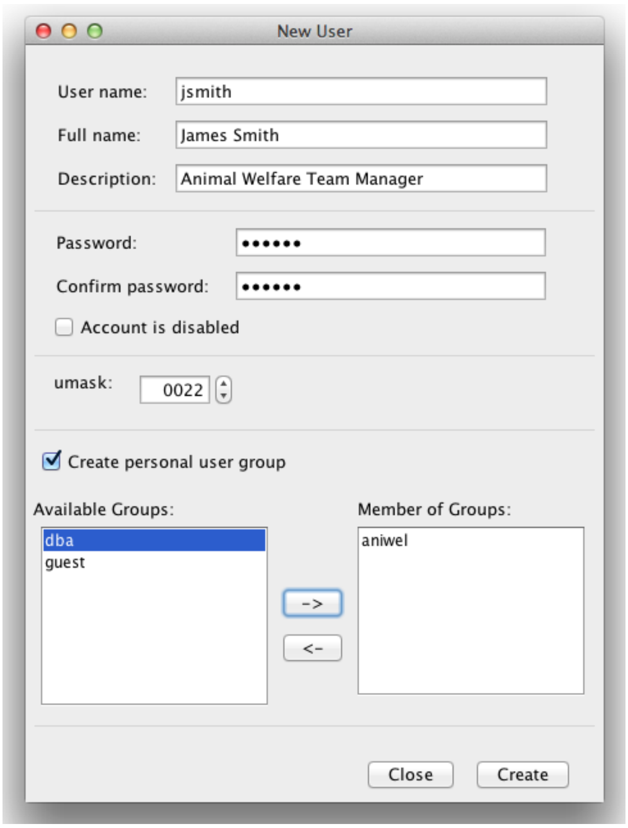


User	Full Name	Description
SYSTEM	SYSTEM	System Internals
admin	admin	System Administrator
guest	guest	Anonymous User

Figure 8-8. Accessing the New User dialog

When the New User dialog appears, you need to complete the form and click the Create button. Keep the name of the user simple and remember that user account names are case-sensitive in eXist. We recommend that you use short usernames in lowercase. Regardless of the scheme you decide on, you cannot use punctuation or whitespace when naming a user.

From the New User dialog you can see a list of available groups of which the user can become a member (providing you have permission to add her to the group). For the purposes of this scenario, we need to add the new user to our newly created group called `aniwel` (Figure 8-9).

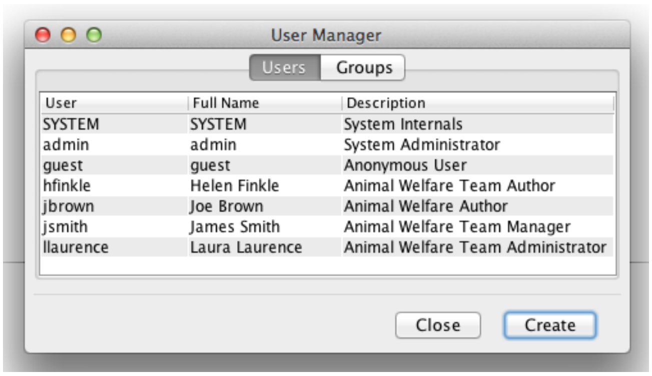


User name: jsmith
Full name: James Smith
Description: Animal Welfare Team Manager
Password: ••••••
Confirm password: ••••••
☐ Account is disabled
umask: 0022
☑ Create personal user group
Available Groups: dba, guest
Member of Groups: aniwel

Figure 8-9. Creating a user

Once you have completed the form, click the Create button. The user will be created and you will be returned to the User Manager Users list.

You should now repeat this process to create the other three users described in "Scenario" on page 145. The complete list is shown in Figure 8-10.



User	Full Name	Description
SYSTEM	SYSTEM	System Internals
admin	admin	System Administrator
guest	guest	Anonymous User
hfinkle	Helen Finkle	Animal Welfare Team Author
jbrown	Joe Brown	Animal Welfare Author
jsmith	James Smith	Animal Welfare Team Manager
llaurence	Laura Laurence	Animal Welfare Team Administrator

Figure 8-10. List of created users

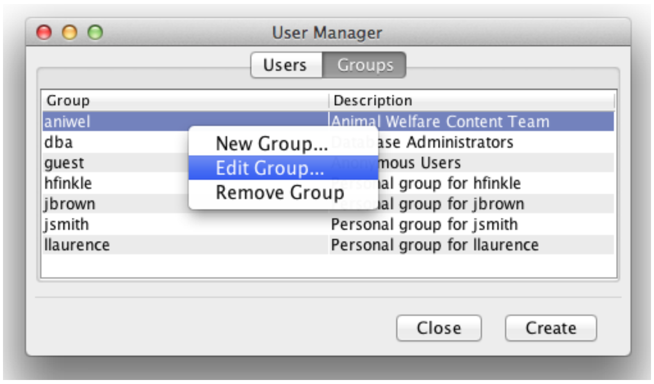
When you're creating a new user, the default option is to create a personal group for that user. This creates a group with the same name as the user, and sets the user as both a member and group manager of that group. This personal group is then also the default group for that user, which means that should the new user add a resource or collection to the database, the owner group of that resource or collection will be the user's personal group. Such an approach helps users avoid adding resources to the database and accidentally granting unintended access to other users of a different group.

When you are creating a new user, it is *always* recommended to also create the personal group.

Setting Group Managers

Finally, we need to set up the managers of the Animal Welfare Content Team as managers of the group that we created for the team.

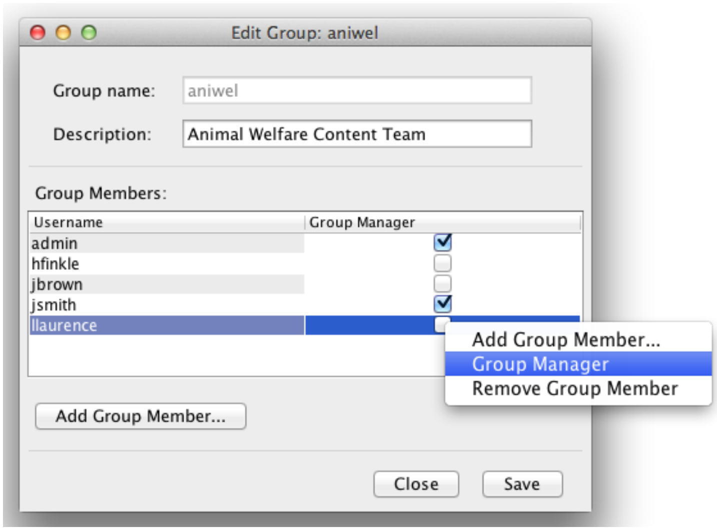
From the User Manager, click the Groups tab, and then select the `aniwel` group in the list. Next, right-click on the `aniwel` group to get a pop-up menu and click the Edit Group menu item (Figure 8-11).



Group	Description
aniwel	Animal Welfare Content Team
dba	ase Administrators
guest	mous Users
hfinkle	al group for hfinkle
jbrown	al group for jbrown
jsmith	Personal group for jsmith
llaurence	Personal group for llaurence

Figure 8-11. Accessing the Edit Group dialog

When the Edit Group dialog appears, you need to promote `jsmith` and `llaurence` to group managers of the `aniwel` group. Select the target group member in the list, and then right-click on the group member. Then click the Group Manager menu item (Figure 8-12).



Group name: aniwel

Description: Animal Welfare Content Team

Group Members:

Username	Group Manager
admin	✓
hfinkle	☐
jbrown	☐
jsmith	✓
llaurence	☐

Figure 8-12. Promoting group members to group managers

Once you have completed the changes to the group members, click the Save button. The group will be updated and you will be returned to the User Manager Groups list. You have now successfully created the users and groups and set up the group managers required in "Scenario" on page 145.

Managing Permissions

When resources are added to the database, the owner of those resources is the user adding the resources, the group of those resources is the primary group of the user adding the resources, and the mode is calculated from default permission modes subject to the user's umask, as described in "User masks" on page 139.

You may modify the permissions of a resource in the database at any time provided you have the appropriate rights to modify the permissions of that resource. Table 8-8 shows the ownership and group membership requirements for modifying the permissions of a collection or resource.

Table 8-8. Requirements to modify permissions

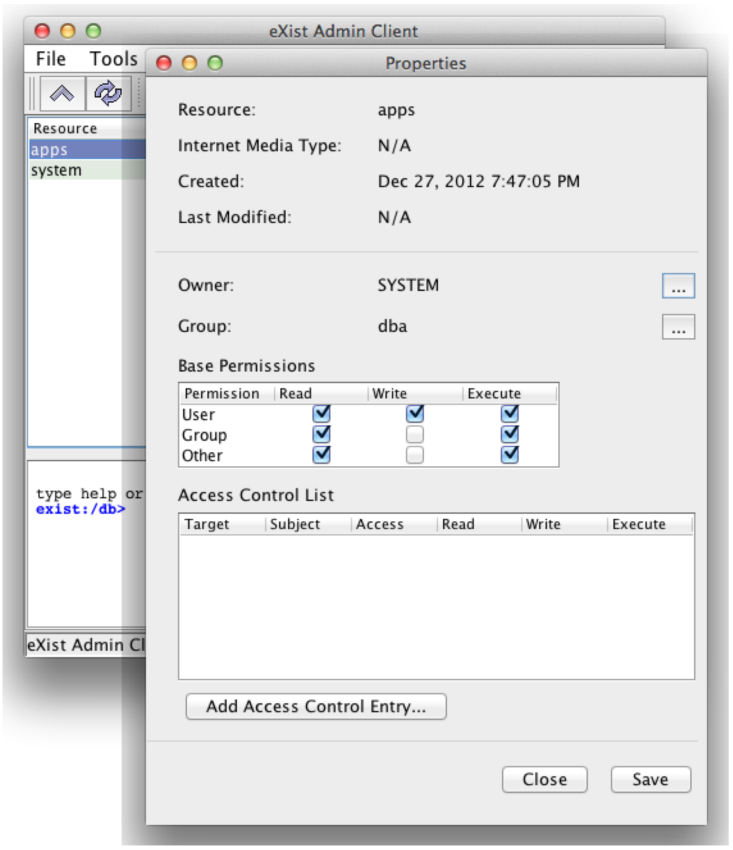
Action	Collection modification requirement	Resource modification requirement
Change owner	• Member of the dba group	• Member of the dba group
Change group	Either: • Owner and member of the destination group • Member of the dba group	Either: • Owner and member of the destination group • Member of the dba group
Change mode	Either: • Owner • Member of the dba group	Either: • Owner • Member of the dba group

Tools for Permission Management

As with managing users and groups, there are a number of approaches to managing permissions in eXist.

Using the Java Admin Client

The Java Admin Client (see "The Java Admin Client" on page 29) provides an excellent facility for modifying the permissions of collections and resources in the database. This is the tool we will focus on in "Permission Management with the Java Admin Client" on page 154. The "Resource properties" option on the File menu in the Java Admin Client opens the permissions management screen shown in Figure 8-13.



```
type help or
exist:/db>
```

Resource:	apps
Internet Media Type:	N/A
Created:	Dec 27, 2012 7:47:05 PM
Last Modified:	N/A
Owner:	SYSTEM
Group:	dba

Base Permissions

Permission	Read	Write	Execute
User	✓	✓	✓
Group	✓	☐	✓
Other	✓	☐	✓

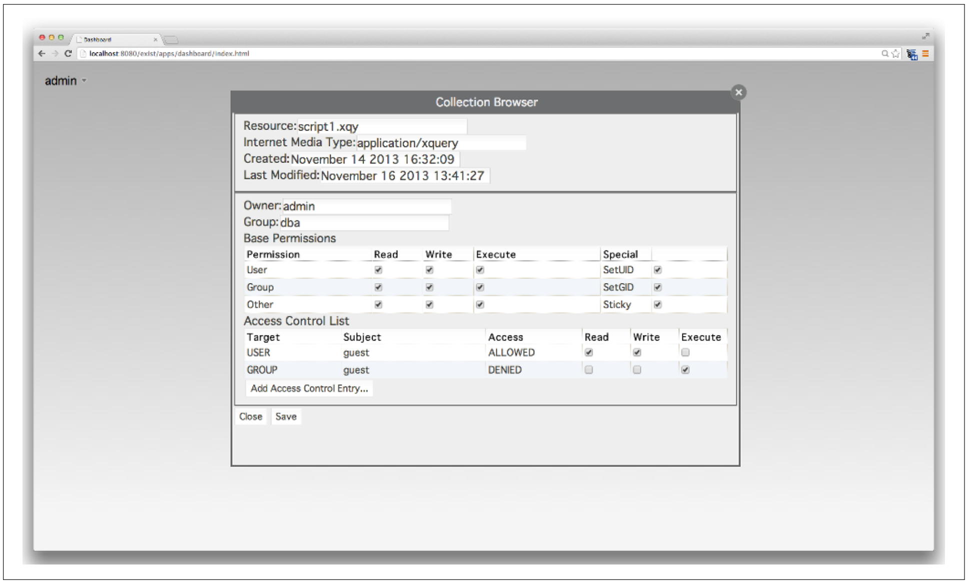
Access Control List

Target	Subject	Access	Read	Write	Execute

Figure 8-13. The Java Admin Client's Resource Properties dialog

Using the collection browser web app

eXist provides a collection browser app out of the box as part of its dashboard. This app has a Resource Properties dialog that provides identical functionality to the Resource Properties dialog in the Java Admin Client and looks almost identical (see Figure 8-14).


Dashboard
localhost:8080/exist/apps/dashboard/index.html
admin
Collection Browser

Resource: script1.xqy
Internet Media Type: application/xquery
Created: November 14 2013 16:32:09
Last Modified: November 16 2013 13:41:27

Owner: admin
Group: dba
Base Permissions

Permission	Read	Write	Execute	Special	
User	✓	✓	✓	SetUID	✓
Group	✓	✓	✓	SetGID	✓
Other	✓	✓	✓	Sticky	✓

Access Control List

Target	Subject	Access	Read	Write	Execute
USER	guest	ALLOWED	✓	✓	☐
GROUP	guest	DENIED	☐	☐	✓

Add Access Control Entry...
Close
Save

Figure 8-14. The Resource Properties dialog of the dashboard's collection browser app

Executing XQuery functions

eXist provides an XQuery module called `xmldb` (see `xmldb`). This module provides functions that allow you to retrieve the created and last-modified timestamps of resources, and allows you to retrieve or set the Internet media type of a resource.

Likewise, the previously mentioned Security Manager XQuery module provides functions for retrieving the permissions and changing the owner user, group, and mode of a resource (see `sm`).

Using the XML-RPC or XML:DB API

eXist's XML-RPC (see "XML-RPC API" on page 342) and XML:DB (see "XML:DB Remote API" on page 349) APIs allow you to programmatically connect to the database remotely (or even within an embedded JVM, if you're using XML:DB) and retrieve and modify the permissions of a resource. The use of these APIs is discussed more thoroughly in Chapter 13.

Using the eXist Ant tasks

eXist provides a series of tasks that can be used with Apache Ant, the basics of which are covered in "Using Ant with eXist" on page 381. These tasks allow you to change permissions (`chmod`), change ownership (`chown`), and add, edit, and remove users and groups.

The Ant tasks can be an excellent choice in a security-constrained environment, as they may be executed directly on the server console and thus do not require any external XML:DB network access to the host. You may, of course, also choose to execute these against a remote eXist server; however, that will have an impact on your security model, as you will also need to allow the XML:DB protocol across the network between your client and server.

Permission Management with the Java Admin Client

Modifying permissions in eXist is relatively simple with the Java Admin Client. First, you must select a resource or collection in the resource list by clicking on it; then you simply open the File menu and click the "Resource properties" menu item to display the Resource Properties dialog (see Figure 8-15).

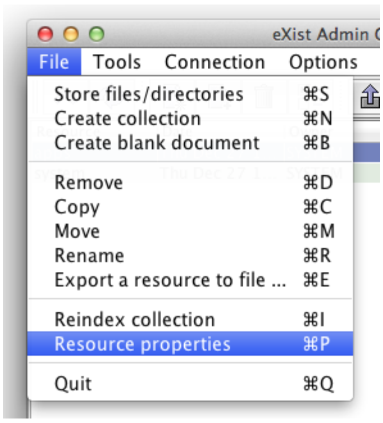
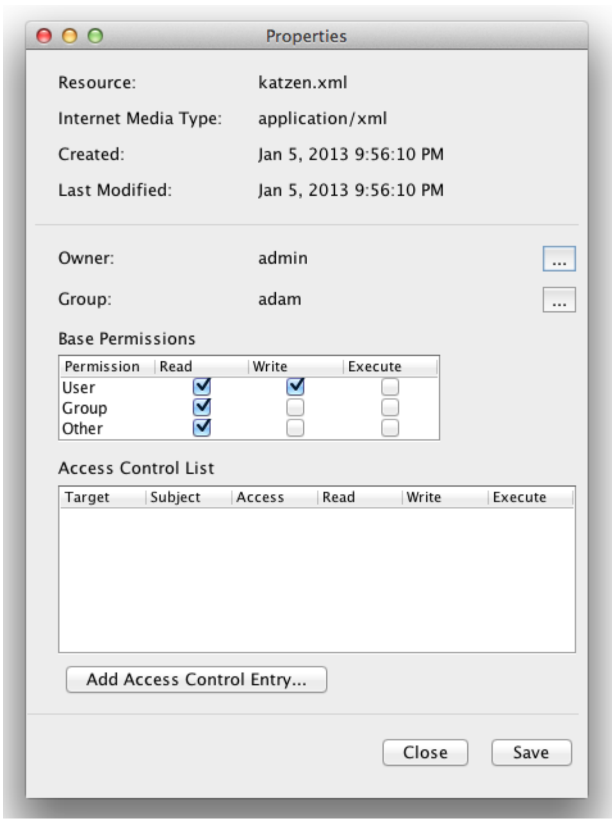


Figure 8-15. Accessing the "Resource properties" menu item

The Resource Properties dialog (Figure 8-16) allows you to change the permission mode applied to the resource using a convenient grid of checkboxes so that you need not remember how to describe the mode in octal format.



Resource:	katzen.xml
Internet Media Type:	application/xml
Created:	Jan 5, 2013 9:56:10 PM
Last Modified:	Jan 5, 2013 9:56:10 PM
Owner:	admin
Group:	adam

Base Permissions

Permission	Read	Write	Execute
User	✓	✓	
Group	✓		
Other	✓		

Access Control List

Target	Subject	Access	Read	Write	Execute

Figure 8-16. Resource Properties dialog

The Access Control List and Add Access Control Entry sections of the Resource Properties dialog are covered in "Access Control Lists" on page 156 and can be safely ignored for the moment.

You may also modify the owner or group of the resource by clicking the button labeled with an ellipsis (...) next to the name of the current owner or group. When modifying the owner or group of a resource, you will be presented with an autocompleted entry field that will only permit you to choose an existing user or group, respectively (see Figure 8-17).

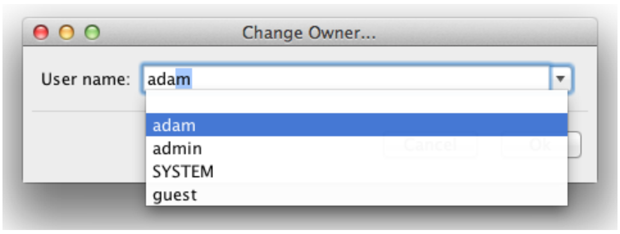


Figure 8-17. Change Owner dialog

Access Control Lists

Access control lists (ACLs) are an advanced permissions model that builds upon the basic Unix-style user and group permissions model that eXist uses. Each resource or collection in eXist may have an ACL in addition to its standard Unix-style permissions of owner, group, and mode.

ACLs are a very new feature in eXist, and familiarity with ACLs is certainly not a prerequisite to being able to understand and make use of eXist. ACLs are available to complement the basic permissions model when it does not provide quite enough for you. The main advantage of ACLs, when used correctly, is greater flexibility in controlling access to resources. The ACLs in eXist were inspired by those in the Network File System v4 (NFSv4) and Zettabyte File System (ZFS), and while they are much simpler than those in their progenitors, should you choose to use them you must clearly understand the evaluation order of ACLs when combined with eXist's Unix-style permissions.

Consider the following scenario: you have a group of 100 users who need to have access to two different collections: *CollectionA* and *CollectionB*. Using the Unix-style permissions, you would simply create a new security group, `GroupA`, add all 100 users to the group, and give that group access to (using an appropriate mode) both *CollectionA* and *CollectionB* and their subresources.

Now suppose that later you are asked to restrict *CollectionA* so that all but 10 members of the group can access it. This means that you now have to create a second group, `GroupB`, containing only 90 of the 100 users and change the permissions on *CollectionA* and its subresources. You now have two groups, `GroupA` and `GroupB`, where `GroupB` is applied to *CollectionA* and `GroupA` is applied to *CollectionB*. Ninety users are in both `GroupA` and `GroupB`, and `GroupA` also has an extra 10 users in it. This quickly becomes a bit of a mess to manage, and so far you have only been asked to make a single change to the security of the system!

Rather than duplicating 90 of the 100 users from `GroupA` into `GroupB`, you could simply have used ACLs. Creating a group of the 10 users for whom you want to remove access to `CollectionA` would allow you to add an access control entry (ACE; discussed in the next section) to the ACL for *CollectionA*, which would prevent that group from accessing it. This is much easier to manage.

Of course, most things that can be done with ACLs can also be done by subdividing permissions—that is, by creating new groups, adding users to them, and applying them with specific modes to individual resources and collections. However, the problem with that approach is that, with even a fairly simple system, you can quickly end up with a vast proliferation of small groups that are not only hard to name, but also hard to manage because you will need to remember why you applied these subdivided groups of users to various resources and collections.

When eXist evaluates the permissions of a resource or collection to determine whether to allow or deny access, *the ACL is evaluated before its Unix-style permissions*. If the ACL is empty, or does not explicitly allow or deny access, then the Unix-style permissions are evaluated.

Simply put, the ACL can override the Unix-style permissions, which is where its power is derived from.

Access Control Entries

Access control lists are composed of access control entries. The ACL of a resource or collection is considered empty if it has no ACEs; otherwise, it may have up to 255 ACEs. When you consider that each ACE may reference a group of users, this gives you a lot of scope for assigning permissions to resources.

An ACE is made up of several fields that describe the access rights to the resource to which the ACL belongs, as shown in Table 8-9.

Table 8-9. ACE fields

ACE field	Description
Target type	Indicates to the ACE whether the ID is that of a `USER` or `GROUP`.
ID	The identifier of the target; that is, the ID of the user or group.
Access type	The type of access applied to the target, either `ALLOWED` or `DENIED`.
Mode	The access mode of the target to the resource or collection. Three octets (e.g., `rwx`).

So, as we can see from the table, the nice thing about ACEs in ACLs is that they not only allow (`ALLOWED`) us to grant access to a resource or collection, but they also allow us to explicitly deny (`DENIED`) access.

The ordering of ACEs in an ACL is critically important.

The ACEs in an ACL are evaluated in order from the start of the list to the bottom of the list. The first ACE in an ACL that both matches a user directly (or indirectly, as a member of a group) and matches the requested access mode will be applied, and the evaluation of permissions will halt.

Consider an ACL with two ACEs in the following order:

1. Prevents a group of users (`GroupA`) from accessing a resource
2. Allows a user (`UserA`) from that group (`GroupA`) access to the resource

Once the ACL is applied, the user `UserA` will not be allowed access to the resource. If you want that user to have access to the resource, you should swap the order of the ACEs in the ACL.

ACLs by Example

It is perhaps easiest to explain ACLs by giving some concrete examples of how they might be used, and explaining the results of various configurations.

Allowing additional access

With the Unix-style permissions in eXist, you can only control access to a resource by the owner, a group of users, and all other users who are not the owner or within that group of users.

Imagine that you work for a small organization in the publishing industry. You have a security group of users who are *content editors* already configured in eXist, and that group has write access to many collections in the database, where each collection represents a different journal. However, one day, one of the editors is in an accident and will be away from the office for several weeks. During this time Bob Smith has to pick up that editor's work on the *Medical Neuroscience Journal*.

Currently, the *Medical Neuroscience Journal* collection is configured as follows:

Collection	Owner	Group	Mode
/db/journals/review/medical-neuroscience	`admin`	`editors`	`rwxrwx---`

So how do we allow Bob Smith to do his temporary editing work on the *medical-neuroscience* collection? There are several possible approaches:

- We could add Bob Smith to the `editors` group.

  Unfortunately, we may then have unintentionally given him access to other journal collections in the database, causing a security risk!

- We could create a new group called `editorsAndBobSmith`, add all the existing members of `editors` to the new group, add Bob Smith to the group, and then change the group applied to the *medical-neuroscience* collection from `editors` to `editorsAndBobSmith`.

  We have not caused any security issues here, but this seems like quite a lot of work and is quite messy! Not to mention, we would have to undo these changes when the original content editor returned.

- As Bob Smith is an exception to the rule encoded in the permissions of the collection, we could add an ACE to the ACL for the *medical-neuroscience* collection that allows Bob Smith access to that collection.

  This is most likely the simplest approach, and arguably the easiest to manage over time.

Let's see how the configuration for the *Medical Neuroscience Journal* might look if we solved this problem using an ACL:

Collection
/db/journals/review/medical-neuroscience

ACL	Target type	ID	Access type	Mode
ACE:0	`USER`	`bobsmith`	`ALLOWED`	`rwx`

Unix-style permissions		
Owner	Group	Mode
`admin`	`editors`	`rwxrwx---`

When Bob Smith tries to access the *medical-neuroscience* collection, eXist checks the collection's ACL, iterating through each ACE in turn until it finds one that matches Bob Smith. In this case the first ACE matches the user `bobsmith` and allows Bob Smith access to the collection with the mode `rwx`, allowing Bob to go about his temporary job.

When one of the *other* existing editors tries to access the *medical-neuroscience* collection, eXist still has to check the collection's ACL. If, after iterating through each ACE in turn it has not found one that matches the editor by user account or group, and so it falls through to the Unix-style permissions and find the editor in the `editors` group, allowing her access with the mode `rwx`.

Restricting access

This example is in some ways the inverse of the previous, but in addition it applies a different mode in the ACL than in the Unix-style permissions to solve a more complex problem.

This time at our small publishing organization, one of our editors has been misbehaving and not rigorously checking articles from an academic institution that he used to be involved with before accepting them to be published. The editor in question, Jason Green, has thus been placed on review and is no longer allowed to make edits to the *Nanotechnology Journal*; he is expected to just review the journal manually and propose changes by email to another editor for review first.

Currently, the *Nanotechnology Journal* collection is configured like so:

Collection	Owner	Group	Mode
/db/journals/review/nanotechnology	`admin`	`editors`	`rwxrwx---`

We could reconfigure this collection with an ACL to prevent Jason Green from changing the collection, while still allowing him the ability to view the collection. Such a configuration for the *nanotechnology* collection could look like this:

Collection
/db/journals/review/nanotechnology

ACL	Target type	ID	Access type	Mode
ACE:0	`USER`	`jasongreen`	`DENIED`	`-w-`

Unix-style permissions		
Owner	Group	Mode
`admin`	`editors`	`rwxrwx---`

When Jason Green tries to access the *nanotechnology* collection, eXist checks the collection's ACL, iterating through each ACE in turn until it finds one that matches Jason Green. In this case the first ACE matches the user `jasongreen`, so from here one of two things happens:

- If Jason Green is trying to write to the collection (add or remove a document), then the ACE *forbids* him from doing that, and he will *not* be allowed access.
- If Jason Green is trying to execute (open) and/or read (the contents of) the collection, then the ACE *does not* forbid him from doing that, as it only denies write access to the collection. This ACE *does not match* the access request, so we fall

through to the Unix-style permissions, which allow those in the `editors` group read and execute access to the collection. As Jason is still an editor, he is allowed access.

When one of the other existing editors tries to access the *nanotechnology* collection, eXist still has to check the collection's ACL. If, after iterating through each ACE in turn it has not found one that matches the editor by user account or group (which it won't, unless he is Jason Green), so it falls through to the Unix-style permissions and find the editor in the `editors` group, allowing him access with the mode `rwx`.

Allowing and restricting access

This example tries to combine aspects of the two previous examples to show a more complex and comprehensive ACL configuration.

This time at our small publishing organization, there are several new developments:

- There has been a recruitment drive, and a number of trainees have been recruited, some of whom will become editors. Management has decided that these trainees should only have read access to the system while they learn the ropes of the job. However, all trainees should have read access to any part of the system, so that they can easily learn more about the organization's business.
- The organization wishes to give read-only access to the printers of the journals so that they can pull the updated content directly from the system.
- Bob Ling at the printing organization will be allowed to modify the journal content to make stylistic changes for a better printed result.

This configuration is illustrated in Figure 8-18.

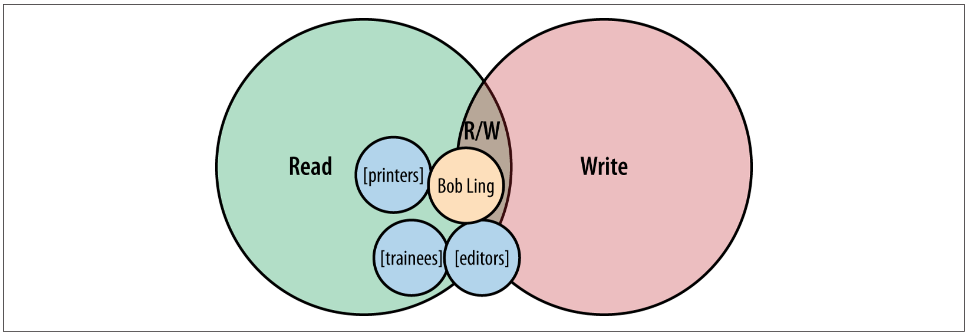


Figure 8-18. Venn diagram of required permissions for the nanotechnology collection

We'll use the *nanotechnology* collection configured in the previous section as a starting point:

Collection
/db/journals/review/nanotechnology

ACL	Target type	ID	Access type	Mode
ACE:0	USER	jasongreen	DENIED	-w-

Unix-style permissions		
Owner	Group	Mode
admin	editors	rwxrwx---

We can modify this configuration to suit our evolving needs. First we create a group called `trainees`, which will contain all of our newly recruited trainees. We also add our trainees to the groups for the roles that they will eventually fulfill (e.g., `editors`). We then add an ACL to prevent members of the `trainees` group from changing the collection, while still allowing them the ability to view the collection. Such a configuration for the *nanotechnology* collection could look like this:

Collection
/db/journals/review/nanotechnology

ACL	Target type	ID	Access type	Mode
ACE:0	USER	jasongreen	DENIED	-w-
ACE:1	GROUP	trainees	ALLOWED	r-x
ACE:2	GROUP	trainees	DENIED	-w-

Unix-style permissions		
Owner	Group	Mode
admin	editors	rwxrwx---

The new ACE at index `1` that we have added allows all trainees read access to the collection. The new ACE at index `2` prohibits any trainee from writing to the collection. While without this trainees who are not editors would not be able to write to the collection, this ensures that all trainees, including any user who is in both the `trainees` group and the `editors` group, cannot write to the collection (otherwise they would be able to, due to eventual fall through to the Unix-style permissions). This works because the ACL is evaluated *before* the Unix-style permission, so any user in both groups will be denied write access due to his membership in the `trainees` group. Neat, right?

The mode in an ACE is explicit rather than implicit. This means that an unset bit in the ACE mode (indicated by the `-` character) is not considered as either `ALLOWED` or `DENIED`, so processing moves to the next ACE in the ACL or finally falls through to the Unix-style permissions. When an ACE is checked, for it to be applied, it must match *both* the target and the requested access mode.

Finally, we need to create a group for all of the users at the printer that will need read access to the system; let's call it `printers`. We then add all our printer users to that group. Next, we add two more ACEs to the ACL on the *nanotechnology* collection. The first will permit Bob Ling, who is in the `printers` group, his extra write access, and the second will allow anyone in the `printers` group (including Bob Ling) read access. The updated configuration for the *nanotechnology* collection now looks like:

Collection
/db/journals/review/nanotechnology

ACL	Target type	ID	Access type	Mode
ACE:0	USER	jasongreen	DENIED	-w-
ACE:1	GROUP	trainees	ALLOWED	r-x
ACE:2	GROUP	trainees	DENIED	-w-
ACE:3	USER	bobling	ALLOWED	-w-
ACE:4	USER	printers	ALLOWED	r-x

Unix-style permissions		
Owner	Group	Mode
admin	editors	rwxrwx---

When Bob Ling tries to write to the *nanotechnology* collection, he will be allowed access by the ACE at index `3`; when he tries to read the collection, he will be allowed access by the ACE at index `4` because he is also a member of the `printers` group.

When another member of the printing staff (who is not Bob Ling) tries to read the collection, she will be allowed access by the ACE at index `4`; if she tries to write the collection, she will fall through to the Unix-style permission, which prohibits anyone apart from the `admin` or a member of the `editors` group from writing to the collection. As this member satisfies neither of these requirements, she will be denied access.

Managing ACLs

When resources or collections are added to the database, they start with an empty ACL. Accordingly, only the Unix-style permissions are in effect. However, you may add and remove ACEs to an ACL of a resource or collection at any time, providing you have the appropriate permissions to do so. Table 8-10 shows which users are permitted to modify the ACL of a collection or resource.

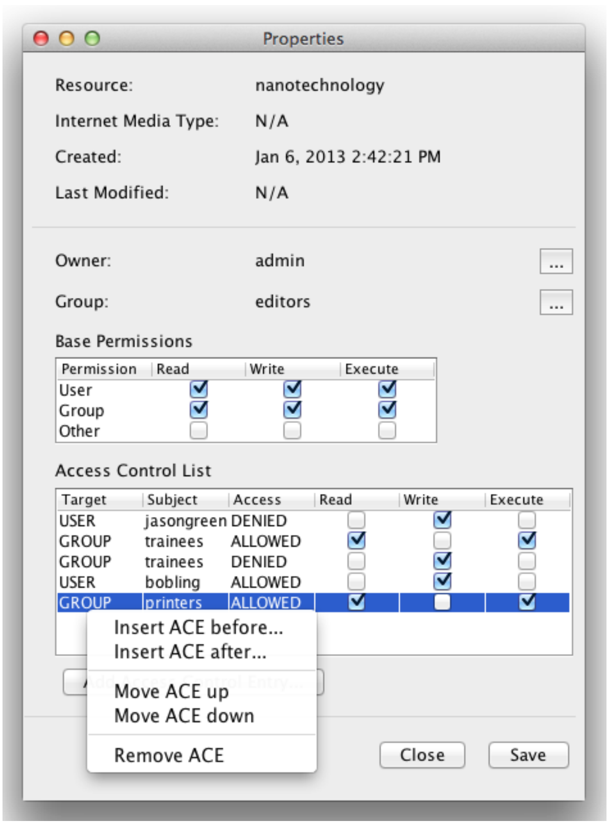
Table 8-10. Permissions to modify ACL

Action	Collection modification requirement	Resource modification requirement
Add, Update, Remove ACE	Either: • Member of the `dba` group • Owner of the collection with write access on the collection	Either: • Member of the `dba` group • Owner of the resource with write access on the resource

The same tools described in "Tools for Permission Management" on page 151 for managing permissions also provide the facilities for managing ACLs.

ACL management with the Java Admin Client

Once you've opened the Resource Properties dialog of the Java Admin Client, as described in "Permission Management with the Java Admin Client" on page 154, you can access the ACL for the resource. As order of ACEs in an ACL is significant, right-clicking on an existing ACE allows you not only to modify that ACE or remove it, but also to move it up or down in the ACL and to insert a new ACE before or after the selection (Figure 8-19).



Resource: nanotechnology
Internet Media Type: N/A
Created: Jan 6, 2013 2:42:21 PM
Last Modified: N/A

Owner: admin
Group: editors

Base Permissions

Permission	Read	Write	Execute
User	✓	✓	✓
Group	✓	✓	✓
Other			

Access Control List

Target	Subject	Access	Read	Write	Execute
USER	jasongreen	DENIED		✓	
GROUP	trainees	ALLOWED	✓		✓
GROUP	trainees	DENIED		✓	
USER	bobling	ALLOWED		✓	
GROUP	printers	ALLOWED	✓		✓

Figure 8-19. ACL management in the Java Admin Client

Clicking either the Add Access Control Entry button, the “Insert ACE before” menu item, or the “Insert ACE after” menu item will display a new dialog that allows you to configure the fields of a new ACE (see Figure 8-20).

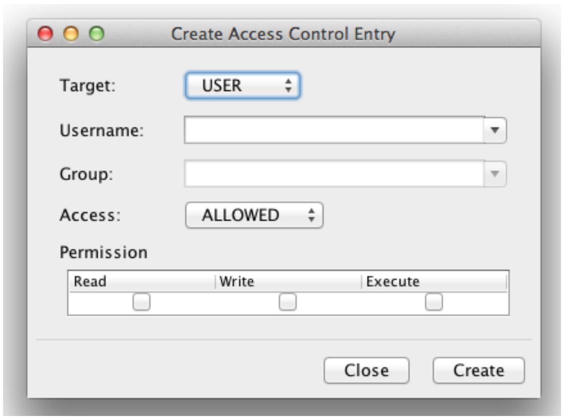


Figure 8-20. Creating an ACE in the Java Admin Client

Realms

As previously mentioned, the Security Manager in eXist permits pluggable modules that provide an authentication *realm* to the system. eXist comes with a default built-in internal realm that authenticates users and groups whose details are stored in a set of protected XML documents in the database.

In addition to the built-in internal realm, some more complex realm modules are available; these allow integration with authentication systems external to eXist. Each module is expected to provide a single authentication realm. You enable the configuration of these modules by modifying the following document in the database: */db/system/security/config.xml*. When multiple realms are configured, eXist will always consult its internal realm first, and then each additional realm in the order in which they are configured in the *config.xml* document. eXist's internal realm cannot be disabled, as it is required for the correct functioning of the system; however, you need not keep any user accounts in it apart from the built-in accounts of `SYSTEM`, `admin`, and `guest` that ship with eXist.

LDAP Realm Module

The Lightweight Directory Access Protocol (LDAP) realm module allows you to authenticate users of eXist against an LDAP directory. While traditionally LDAP was used in larger organizations for centralized user management, Microsoft's Active Directory (AD) technology (among other technologies) is built upon it.

The LDAP module in eXist is very flexible and can be configured to authenticate against almost any LDAP directory server, including domain controllers within a Microsoft Active Directory domain configuration. The LDAP module is shipped with eXist by default, so if you wish to use it simply add its realm configuration to the security configuration in */db/system/security/conf.xml*.

LDAP itself does not impose a structure on any particular directory system; rather, it allows you to create a directory structure of your own devising. Products like Red Hat IPA (Identity, Policy, and Audit) and Microsoft Active Directory typically impose a common proprietary structure on an LDAP directory implementation. The LDAP module is flexible enough to cope with any directory structure, but this flexibility comes at a price—namely, that the configuration of the LDAP module is more complicated than that of other such modules in eXist. However, it should not be too difficult for those familiar with LDAP.

The configuration options available for the LDAP module are comprehensive, so we will examine each and provide some explanation. We will also provide an example configuration for integrating with Microsoft Active Directory.

The current design of the LDAP module causes eXist to cache LDAP account credentials in the */db/system/security/ldap* collection.

This has a few implications that you should be aware of:

- A copy of your LDAP password will be kept securely (RIPEMD-160 hashed), which may or may not meet the security requirements of your organization.
- If a user is deleted or disabled in the LDAP directory, he will still have access to eXist until his cached credentials are *manually* removed from eXist.

LDAP configuration options

The configuration is specified in an LDAP realm configuration inside the security XML configuration file. An XML Schema 1.1 schema is provided with eXist for checking your LDAP configuration structure; you can find it in *$EXIST_HOME/extensions/security/ldap/ldap-realm.xsd*. An example for Microsoft Active Directory is provided in "LDAP configuration for Microsoft Active Directory" on page 172.

All configuration options in Table 8-11 are mandatory unless otherwise stated.

Table 8-11. Explanation of LDAP configuration options by category

Configuration option name	Description
context (LDAP context configuration)	
`principals-are-case-insensitive`	This can be either `true` or `false`. eXist itself is case-sensitive for security principals (i.e., usernames and group names), but your LDAP directory may not be case-sensitive (e.g., Active Directory is not). If your LDAP directory is not case-sensitive, then you should set this to `true` to ease interoperability.

`authentication`	This can be either `none`, `simple`, or `strong`:
	
	`none`
	No authentication is required before making queries of the LDAP server.
	
	`simple`
	Basic username and password authentication, sent in plain text across the network. This can be improved by using SSL.
	
	`strong`
	This is unsupported at this time in eXist.

`use-ssl`	This can be either `true` or `false`. It's optional, and defaults to `false`.
`url`	The URL of your LDAP server (e.g., `ldap://dir.mydomain.com:389`, or for SSL `ldaps://dir.mydomain.com:636`). For Active Directory, use the address of one of the domain controllers or the AD itself.
`domain`	The domain name that your LDAP directory describes (e.g., `mydomain.com`).
search (LDAP search configuration)	
`base`	The base of the LDAP context to search. Allows you to restrict the scope of searches within your LDAP directory to a specific *distinguished name* (e.g., `dc=dir,dc=mydomain-ever,dc=com`). Otherwise, if you wanted to, say, limit the search scope to just the office of engineers within your directory for your organization, you'd do so with `ou=engineers,ou=offices,dc=dir,dc=mydomain,dc=com`. In LDAP parlance, `dc` stands for domain component and `ou` for organization unit.
`default-username`	The default username, used if eXist needs to attach to the directory to perform a query for a system task it is carrying out. Typically the username of the user authenticating with eXist will be used instead of this. This account need only have minimal read access to the directory server.

Configuration option name	Description
context (LDAP context configuration)	
`default-password`	The default password, used with the default username.
account (LDAP account search and property mapping configuration)	
`search-filter-prefix`	The prefix to use when searching the LDAP directory. This should indicate the class of a user within the LDAP directory. For example, with Active Directory you would use the value `objectClass=user`. eXist would then construct an LDAP search string like `(&(objectClass=user)(`*`name`*`=`*`value`*`))`, where *`name`* will be substituted by the actual LDAP attribute indicated by a `search-attribute` and *`value`* will be substituted by the criteria of the thing you are trying to find. Trying to retrieve the user account for Bob Smith (`bsmith`) from Active Directory would, for example, cause eXist to produce the LDAP search string `(&(objectClass=user)(sAMAccountName=bsmith))`.
`search-attribute`	As an LDAP directory can come in any shape, eXist needs to know how to address certain properties of the user account in the directory. The `search-attribute` maps an account property that eXist can understand to an LDAP directory property. eXist requires `search-attribute` for the following account properties:

eXist account property	Map to (description)
`objectSid`	The property that holds a SID (Unique Security Identifier) for the account.
`primaryGroupID`	The property that holds the ID of the user account's primary group membership.
`name`	The property that holds the username of the account (i.e., the name used to log in).
`dn`	The property that holds the LDAP directory DN (distinguished name) of the account.
`memberOf`	The property that holds the list of groups that this account is a member of.

Configuration option name	Description
context (LDAP context configuration)	
`metadata-search-attribute`	eXist supports the notion of storing and retrieving metadata about a user account, and provides some support for searching for accounts using these metadata properties. For the purposes of LDAP, this metadata is read-only. However, if you want seamless integration with such functionality in eXist, then there are a number of properties that you must map to properties of the accounts in your LDAP directory. eXist currently supports the following metadata properties, all of which may be retrieved, but only some of which are used for search. We would recommend you configure at least the properties that are used for searching:

Metadata property	Used for search?
http://axschema.org/namePerson	Yes
http://axschema.org/namePerson/first	Yes
http://axschema.org/namePerson/last	Yes
http://axschema.org/namePerson/friendly	No
http://axschema.org/contact/email	No
http://axschema.org/contact/country/home	No
http://axschema.org/pref/language	No
http://axschema.org/pref/timezone	No
http://exist-db.org/security/description	No

Configuration option name	Description
`whitelist`	An optional whitelist of LDAP user accounts that are permitted access to eXist. The `blacklist` is *always* evaluated before the `whitelist`. If a `whitelist` is provided, a user must appear in the list to get access to eXist; otherwise, she will be denied access via LDAP.

group (LDAP group search and property mapping configuration)	
Configuration option name	**Description**
context (LDAP context configuration)	
`blacklist`	An optional blacklist of LDAP user accounts that are forbidden access to eXist. If a user is not in the `blacklist` and a `whitelist` is not provided, then he is given access via LDAP; otherwise, he is further checked against the `whitelist`.
`search-filter-prefix`	The prefix to use when searching the LDAP directory. This should indicate the class of a group within the LDAP directory. For example, with Active Directory you would use the value `objectClass=group`. eXist would then construct an LDAP search string like `(&(objectClass=group)(`*`name`*`=`*`value`*`))`, where *`name`* will be substituted by the actual LDAP attribute indicated by a `search-attribute` and *`value`* will be substituted by the criteria of the thing you are trying to find. Trying to retrieve the user group `editors` from Active Directory would, for example, cause eXist to produce the LDAP search string `(&(objectClass=group)(sAMAccountName=editors))`.
`search-attribute`	As an LDAP directory can come in any shape, eXist needs to know how to address certain properties of the user group in the directory. The `search-attribute` maps a group property that eXist can understand to an LDAP directory property. eXist requires `search-attribute` for the following group properties:

eXist group property	Map to (description)
`objectSid`	The property that holds a SID (Unique Security Identifier) for the group.
`name`	The property that holds the name of the group.
`dn`	The property that holds the LDAP directory DN (distinguished name) of the group.
`member`	The property that holds the list of members (i.e., user accounts) of this group.

Configuration option name	Description
context (LDAP context configuration)	
`metadata-search-attribute`	eXist supports the notion of storing and retrieving metadata about a group. For the purposes of LDAP, group metadata is currently unsupported. However, it is likely that this may be implemented in future versions of eXist if demand arises.
`whitelist`	An optional whitelist of LDAP groups that are permitted access to eXist. The `blacklist` is *always* evaluated before the `whitelist`. If a `whitelist` is provided, a group must appear in the list to get access to eXist; otherwise, its members will be denied access via LDAP.
`blacklist`	An optional blacklist of LDAP groups that are forbidden access to eXist. If a group is not in the `blacklist` and a `whitelist` is not provided, then its members are given access via LDAP; otherwise, they are further checked against the `whitelist`.
transformation (transformations applied to LDAP to aid integration)	
add-group	This optional transformation allows you to automatically add each LDAP user to a group known to eXist from another realm (e.g., its internal realm). For example, you could create a group in eXist called `businessUsers` and have all LDAP users automatically added to this group, and they would be granted access to any collections or resources that you have permitted the `businessUsers` group access to.

LDAP configuration for Microsoft Active Directory

Before attempting to configure eXist to authenticate with your Active Directory, it is highly recommended that you discuss this with your network administrators. In any case, they will need to provide you with the username and password for a low-privileged account to use in the `default-username` and `default-password` parts of the realm configuration.

It is also recommended that you first use a tool like Apache Directory Studio to ensure that you can connect to your Active Directory using LDAP with the username and password provided for the low-privileged account by your network administrator. Apache Directory Studio is a particularly good choice, as not only is it very easy to use and functional, but it is written in Java and uses the same underlying LDAP libraries that eXist will use to connect to your LDAP directory.

Remember that if you are using SSL, the LDAP port for your Active Directory connection will most likely be 636 and the scheme is `ldaps://`; if you are not using SSL, then it is most likely 389 with the scheme `ldap://` (as shown in Example 8-1).

Example 8-1 shows an example configuration.

Example 8-1. Security Manager configuration with an LDAP realm for Microsoft Active Directory

```
<security-manager xmlns="http://exist-db.org/Configuration"
    xmlns:xsi="http://www.w3.org/ 2001/XMLSchema-instance">
    <authentication-entry-point>/authentication/login</authentication-entry-point>
    <realm id="LDAP" version="1.0" principals-are-case-insensitive="true">
        <context>
            <authentication>simple</authentication>
            <use-ssl>false</use-ssl> ❶
            <url>ldap://ad.mydomain.com:389</url> ❷
            <domain>ad.mydomain.com</domain> ❸
            <search>
                <base>ou=mygroup,dc=ad,dc=mydomain,dc=com</base> ❹
                <default-username>account@ad.mydomain.com</default-username> ❺
                <default-password>XXXXXXX</default-password> ❻
                <account>
                    <search-filter-prefix>objectClass=user</search-filter-prefix>
                    <search-attribute key="objectSid">objectSid</search-attribute>
                    <search-attribute key="primaryGroupID">primaryGroupID
                    </search-attribute>
                    <search-attribute key="name">sAMAccountName</search-attribute>
                    <search-attribute key="dn">distinguishedName</search-attribute>
                    <search-attribute key="memberOf">memberOf</search-attribute>
                    <metadata-search-attribute
                        key="http://axschema.org/namePerson">name
                        </metadata-search-attribute>
                    <metadata-search-attribute
                        key="http://axschema.org/namePerson/last">sn
                        </metadata-search-attribute>
                    <metadata-search-attribute
                        key="http://axschema.org/namePerson/first">givenName
                        </metadata-search-attribute>
                    <metadata-search-attribute
                        key="http://axschema.org/contact/email">mail
                        </metadata-search-attribute>
                </account>
                <group>
                    <search-filter-prefix>objectClass=group</search-filter-prefix>
                    <search-attribute key="member">member</search-attribute>
                    <search-attribute key="objectSid">objectSid</search-attribute>
                    <search-attribute key="name">sAMAccountName</search-attribute>
                    <search-attribute key="dn">distinguishedName</search-attribute>
                </group>
            </search>
        </context>
    </realm>
</security-manager>
```

❶ `true` if you are using SSL, or `false` otherwise.

❷ The network host name of either the Active Directory domain or a domain controller within the domain, and the TCP port to talk to the Active Directory LDAP server on; TCP port 389 is usual, or 636 if you are using SSL.

❸ The fully qualified Active Directory domain name.

❹ The LDAP search base for your Active Directory, typically an organization unit followed by the Active Directory domain name components.

❺ The username of an Active Directory account, which may connect to the LDAP server and interrogate the LDAP store.

❻ The password of the Active Directory account being used.

Other Realm Modules

eXist also has authentication modules for OpenID and OAuth, but these are relatively new and not yet completely integrated into the eXist Security Manager. They are almost certainly not ready for production use; however, if you do need OpenID or OAuth support, they could serve as a starting point for further development or discussion with the eXist community.

The source code for these modules can be found in *$EXIST_HOME/extensions/security*. It is expected that these modules will be further developed in the near future and added to a subsequent release of eXist.

Hardening

The topic of *hardening* focuses on taking a standard eXist installation and modifying its defaults to make it more resilient to would-be intrusion. As eXist ships, it is in pretty good shape from a security perspective, but it also needs to be usable by a wide array of people for a variety of tasks, so some flexibility is afforded; there are several additional things that can be done to increase the security of your installation. This information is most pertinent if you are running an eXist server and providing access to others, for example as a website or web services.

Now, we certainly do not want our wonderful eXist server to be compromised by "the bad guys," and the eXist developers have gone to a lot of effort to try to ensure that eXist maintains the integrity and security of your data. However, you should never consider any computer system completely secure from intruders. As computer security expert Gene Spafford once said:

> The only truly secure system is one that is powered off, cast in a block of concrete, and sealed in a lead-lined room with armed guards.

With that in mind, we take a somewhat pessimistic view in this chapter and concede that your system could indeed be compromised. However, we explain how you can reduce the chances of this happening, and, should it happen, how you can limit the damage caused. The advice herein should be seen as a guide, and should not be substituted for the latest professional security advice. Certainly, there is always more that can be done.

Reducing Collateral Damage

Should your eXist-db installation be compromised, you want to limit the amount of damage that can be done by the intruder to the underlying server on which eXist is installed. One of the most effective ways to do this is to run eXist under an unprivileged service account that is created for this sole purpose.

The unprivileged account should have absolutely no login rights to your server or network. Further, the account should not be a member of any security groups on your server, apart from a personal group of which it is the only member.

The *$EXIST_HOME* folder and all directories and files therein should be owned by a secure system user, and the personal group of the unprivileged account should have only the access permissions to the files in *$EXIST_HOME* that it needs.

Typically, the personal group will need read access to all files in *$EXIST_HOME* and *only* write access to the following folders:

- *$EXIST_HOME/webapp/WEB-INF/data* (configured in *$EXIST_HOME/conf.xml*)
- *$EXIST_HOME/webapp/WEB-INF/logs* (configured in *$EXIST_HOME/conf.xml*)
- *$EXIST_HOME/tools/wrapper/bin* (for just *.pid* and *.status* files)
- *$EXIST_HOME/tools/wrapper/logs* (configured in *$EXIST_HOME/tools/wrapper/conf/wrapper.conf*)
- *$EXIST_HOME/tools/jetty/logs* (configured in *$EXIST_HOME/tools/jetty/etc/jetty.xml*)

- *$EXIST_HOME/tools/jetty/tmp* (configured in *$EXIST_HOME/tools/jetty/etc/jetty.xml*)
- *$EXIST_HOME/tools/jetty/work* (configured in *$EXIST_HOME/tools/jetty/etc/webdefault.xml*)

In addition, the personal group will need read and write access to the folder containing the *.pid* file if you are using the service wrapper (see *$EXIST_HOME/tools/wrapper/bin/exist.sh*) and the temporary folder used by your system's JVM (this varies depending on the JVM vendor, the version, and the operating system).

Depending on which operating system you installed eXist on, there are slightly different approaches to configuring eXist to run under an unprivileged account. We will assume that you have created the unprivileged account and set the permissions on the *$EXIST_HOME* folder and its contents correctly.

Linux platforms

The recommended way to run eXist as a service (system daemon) on Linux is using the Java Service Wrapper shipped in *$EXIST_HOME/tools/wrapper* and discussed in "Windows Linux and Other Unix" on page 407. If you are not taking that approach, we will assume that you know enough about Linux to complete the configuration yourself. If you are using the Java Service Wrapper, then you simply need to change the file *$EXIST_HOME/tools/wrapper/bin/exist.sh* by uncommenting the following line in that file and setting it to your unprivileged account (replace *`existsrv`* with the correct user):

```
RUN_AS_USER=existsrv
```

You will also need to ensure that this user has execute access on *$EXIST_HOME/tools/wrapper/bin/exist.sh*, as this will be called when the system service is started. It is also worth remembering that when installing the Java Service Wrapper, you must perform this as a user who has *root* access, either through *sudo* or directly, as the wrapper will install files into */etc*.

Solaris platforms

eXist ships with support for the Solaris SMF (Service Management Framework); see "Solaris" on page 406. The documentation provided in the *$EXIST_HOME/tools/Solaris/README.txt* file explains in detail how you can trivially configure the user account and group that eXist runs under.

Windows platforms

Providing that you opted to install eXist as a *service* during installation, you are already running eXist using the Java Service Wrapper integrated with Windows Services. If you did not install eXist as a service, you can do this by either using the appropriate icon in the eXist group on the Start menu, or following the instructions in "Windows Linux and Other Unix" on page 407.

By default the eXist service will run under the user account that you used to install eXist. Once you have set up your unprivileged account and the necessary permissions, you can reconfigure the service to run under your unprivileged account by opening the Services Manager, locating `eXist-db` in the Windows Services list, and editing its properties (double-click on the service). Go to the Log-On tab, change the setting from Local System Account to This Account, and specify the details of the unprivileged account that you created.

Reducing the Attack Surface

eXist comes as a very full-featured product with many things enabled by default, and several of these features are delivered as network services. In its default configuration eXist presents a rather large surface that an outside intruder could attempt to exploit. Fortunately, eXist is very configurable, and we can reduce the chance of an attack vector being found by disabling various features and services that eXist provides.

Disabling extension modules

eXist ships with extension modules, and many of these are enabled by default. These modules provide additional functionality to eXist in one of three areas:

- XQuery functions
- Security realms
- Indexing

Whatever the functionality, these extension modules have to be enabled in a two-step process. First, they have to be compiled into the eXist release, and second, they have to be enabled in eXist's configuration file (*$EXIST_HOME/conf.xml*) or in the Security Manager configuration (*/db/system/security/config.xml*).

While these extension modules provide a wealth of features, they are useful for different reasons and for different projects. It is unlikely that you will need to make use of many of these extension modules, and thus it is recommended that you only enable the extension modules that you absolutely require for your project. If you do not know if you are using an extension module, then most likely you are not.

To disable an extension module that was previously enabled, you can optionally remove its compiled code from your eXist installation (if you opted to install the source code); then you must disable it in eXist's configuration file.

Before changing eXist's configuration, you should always make sure that you have a recent and valid backup of your eXist database and its current configuration. "Backup and shutdown" on page 382 explains how to create backups.

To remove the code of a compiled extension module, you need to edit the *$EXIST_HOME/extensions/local.build.properties* file (if you do not have this file, you should create it from a copy of *$EXIST_HOME/extensions/build.properties*), find the appropriate `include` entry, and change it from `true` to `false`. Next, make sure that eXist is not running, and then run the following commands from the *$EXIST_HOME* directory:

```
$ ./build.sh clean
```

```
$ ./build.sh
```

This will recompile eXist in place, without the extension module(s) that you disabled in the *build.properties* file.

The preceding examples are for Unix-like platforms. If you are on a Windows platform, you should replace `.sh` with `.bat`, and you need not worry about the `./` part of the command.

To disable an XQuery or indexing extension module, you can simply modify *$EXIST_HOME/conf.xml* and comment out the extension module that you wish to disable. This file itself is well documented within, and the process should be self-explanatory. The configuration only takes effect at startup, so you should do this when eXist is not running, or you will need to restart eXist for the changes to take effect. To disable a Security Realm extension module, you need to remove its realm configuration from the */db/system/security/config.xml* XML document in the database; it is then recommended that you immediately restart eXist.

Performing an audit of both *$EXIST_HOME/extensions/build.properties* and *$EXIST_HOME/conf.xml* and disabling unused modules is always recommended before deploying a public or production instance of eXist.

There are several extension modules, listed in Table 8-12, that are of higher risk than others, and you should fully understand the implications of using them. With suitable precaution and understanding, it is perfectly fine to use these modules, but you should be aware of what an attacker could attempt with them.

Table 8-12. Security-sensitive XQuery extension modules

XQuery extension module	Security implications of use
Cache module	A user could potentially store a large quantity of information into the cache. This would exhaust the memory available to the JVM used by eXist and could cause the database to crash.
Counter module	A user could create many counters on disk, filling the available disk space of the server and thereby potentially crashing eXist or the underlying server.
File module	You have to be a user in the `dba` group in eXist to use the functions in the File module. However, should a user be able to obtain `dba` privileges, he will be able to read/write anywhere in the filesystem that the account under which the eXist server is running can read/write. Potentially, he could delete the *.dbx* files that make up your database, download data or sensitive files, or fill up your server's disk space.
FTP module(s)	The EXPath FTP module could allow a user to retrieve documents from remote sources by URI; you can reduce or alleviate this risk by using sensible firewall and network settings. This module could also potentially be used to launch a Denial of Service (DoS) attack against a remote service.
HTTP module(s)	eXist and EXPath HTTP module could allow a user to retrieve documents from remote sources by URI; you can reduce or alleviate this risk by using sensible firewall and network settings (e.g., forced transparent proxy). These modules could also potentially be used to launch a DoS attack against a remote service.
JNDI module	A user could potentially gain access to external JNDI sources, depending on how the authentication of the JNDI sources is configured by the container (e.g., Jetty, Tomcat, etc.).
Mail module	The Mail module could allow a user to send email either via a local *sendmail* or via a remote SMTP host. This could be used to send SPAM or launch a DoS attack against a remote SMTP server. Consequently, a local *sendmail* should not be used and remote SMTP servers should be secured appropriately to avoid such attacks.
Memcached module	A user could potentially store a large quantity of information into a Memcached instance, which could affect the stability of that instance and the server on which it resides.
Session module	A user could potentially store a large quantity of information into her session. This would exhaust the memory available to the JVM used by eXist and could cause the database to crash.
SQL module	A user could potentially open many connections against a remote SQL database, which could affect its ability to respond and, in effect, cause a DoS attack.
Util module	A user could use the `util:serialize` function to create many files on disk, filling the available disk space of the server and therefore potentially crashing eXist or the underlying server. A user could use the `util:wait` function to pause the current database broker for a long period. If this was done across many connections, the available brokers could rapidly be exhausted, effectively causing a DoS attack on eXist.

XQuery extension module	Security implications of use
XPath and XQuery Functions and Operators module	A user could use the `fn:doc` function to retrieve documents from the filesystem to which the user under which the eXist server is running has read access. There is also the potential to download documents from remote sources by URI. You can reduce or alleviate this risk by using sensible firewall and network settings. This module could also potentially be used to launch a DoS attack against a remote service.

Disabling the Java binding from XQuery

eXist provides a feature called *Java binding* (see "Java binding" on page 117 for further details) that allows Java code to be written inside XQuery code and executed by eXist's XQuery processor. This is undoubtedly a very powerful feature that could prove useful where an XQuery developer wishes to perform some task that is not possible in XQuery and for which no extension functions are provided. However, in the wrong hands, it gives an attacker access to a programming language that has a rich standard library.

By default, eXist is shipped with the Java binding disabled. You can configure this in *$EXIST_HOME/conf.xml* by changing the value of the `enable-java-binding` attribute on the `xquery` configuration element to either `yes` or `no`. It is *highly recommended* that the Java binding remain disabled. If you require additional functionality that XQuery does not provide, you should consider writing an extension module for XQuery in Java that encapsulates just the functionality that you require (see "Internal XQuery Library Modules" on page 467).

Disabling direct access to the REST Server

While it is recommended that you disable various XQuery modules and Java binding if you're not using their functionality, what can you do to limit what anonymous web users can exploit when you *do* require such functionality?

One possibility is to limit the ability for eXist's REST Server to directly receive web requests, including XQuery submissions that would otherwise be processed dynamically. Disabling this capability still allows you to place main modules and associated library modules written in XQuery into the database as binary documents and have them executed via URI calls, but access to database resources is instead controlled by XQuery URL Rewrite (see "URL Mapping Using URL Rewriting" on page 194). Such a restriction prevents anonymous and authenticated users from directly accessing database resources or sending in XQueries via eXist's REST Server, unless you permit it within your own XQuery controller.

To remove the REST Server's ability to directly receive web requests, you can modify the parameter `hidden` in *$EXIST_HOME/webapp/WEB-INF/web.xml*:

```
<init-param>
  <param-name>hidden</param-name>
  <param-value>true</param-value>
</init-param>
```

Changing this parameter means that the REST Server will not directly receive requests; rather, it will only receive requests forwarded from an XQuery controller (see "URL Mapping Using URL Rewriting" on page 194). You can then choose to filter such requests in your own XQuery controller.

Disabling network services and APIs

eXist provides several network services and APIs that are enabled by default; however, it is quite likely that you will not require all of those services in a production environment. Which services you require will depend on your application, but you should disable any that are not required by your application and/or users. You do so by commenting out the `servlet`, `filter`, `filter-mapping`, and `listener` declarations in *$EXIST_HOME/webapp/WEB-INF/web.xml*, as well as any servlet mappings to them in *$EXIST_HOME/webapp/WEB-INF/controller-config.xml*. See Table 8-13.

Table 8-13. eXist's network service components

Servlet	Provides	Impact of disabling
`org.exist.http.servlets.` **Log4jInit**	Initializes all logging services for eXist	eXist will not be able to log any messages.[a]
`org.exist.xmlrpc.` **RpcServlet**	XML-RPC and XML:DB Remote APIs; removes RPC and XML:DB Remote access	Will remove the ability to connect with the Java Admin Client, as this uses the XML:DB Remote API. May remove the ability to shut down or back up eXist if you use *shutdown.sh* or *backup.sh*, as these use the Java Admin Client library. It may be appropriate to disable this, depending on your environment and deployment of eXist.
`org.exist.http.servlets.` **EXistServlet**	REST Server and SOAP Server	Removes the REST Server and the ability to navigate the database automatically by URI and execute XQueries stored in the database by URI. Removes the ability to create SOAP web services of XQuery modules.
`org.exist.management.client.` **JMXServlet**	JMX information over HTTP as XML	Removes the ability to retrieve JMX information over HTTP as XML (see "JMX" on page 387). It is typically recommended to disable this API in production, or restrict access at a reverse proxy or firewall.

Servlet	Provides	Impact of disabling
`org.exist.webdav.` **MiltonWebDAVServlet**	WebDAV access to the eXist database	Disables WebDAV access to eXist. It is common to disable this API in a production web environment, or restrict access at a reverse proxy or firewall.
`org.exist.http.servlets.` **XQueryServlet**	Ability to execute XQuery main modules stored in files in eXist's `$EXIST_HOME/webapp` folder	Disables execution of XQuery from the filesystem. This *should* be disabled, as it is now deprecated and has been superseded by the REST Server.
`org.exist.http.urlrewrite.` **XQueryURLRewrite**	Disables XQuery-based URL rewriting	If you are not using XQuery URL rewriting with the REST Server or the eXist dashboard app, or you are using RESTXQ, then this can be disabled. If you disable this, you will need to translate the servlet mappings from *$EXIST_HOME/webapp/WEB-INF/controller-config.xml* to *$EXIST_HOME/webapp/WEB-INF/web.xml* as the URL rewriting controller is then no longer responsible for mapping URIs.
`org.exist.http.servlets.` **XSLTServlet**	XSLT transformations for XQuery URL rewriting	Disables XSLT transformations from XQuery URL rewriting. If you are not using XSLT transformations through forwarding in your XQuery URL Rewrite *controller.xql*, then this can be disabled. If you have disabled XQuery URL rewriting, then this should also be disabled.
`org.apache.axis.transport.http.` **AxisServlet** **AdminServlet**	SOAP API access to the eXist database	Disables eXist's SOAP API. If you are not making use of eXist's SOAP API, then both of these servlets should be disabled.
`org.exist.atom.http.` **AtomServlet**	Atom API access to the eXist database	Disables eXist's Atom API. If you are not making use of eXist's Atom API, then this should be disabled.
`org.exist.extensions.exquery.restxq.impl.` **RestXqServlet**	EXQuery RESTXQ framework	Disables the RESTXQ framework. If you are not using the RESTXQ framework in your XQuery modules, then this should be disabled.

Servlet	Provides	Impact of disabling
org.exist.webstart. **JnlpServlet**	Java WebStart of Java Admin Client	Disables the ability to download the Java Admin Client from eXist as a Java WebStart application. If you are not allowing remote users to download the Java Admin Client for eXist from your eXist server via Java WebStart, then this should be disabled.
org.directwebremoting.servlet. **DwrServlet** de.betterform.agent.web.servlet. **XFormsPostServlet** **FormsServlet** **XFormsInspectorServlet** **ResourceServlet** **ErrorServlet** **BfServletContextListener** de.betterform.agent.web.filter. **XFormsFilter**	XForms support through the betterForm engine	Disables server-side XForms support through betterForm. If you are not using XForms or are using XSLTForms or another XForms rendering engine, then you should disable these servlets, the filter, and the listener.

[a]Disabling the logging service is *not* recommended.

Disabling autodeployment of EXPath packages

eXist provides a mechanism to autodeploy applications, extension libraries, and data provided as EXPath packages to the database at startup time. Any EXPath package placed in the *$EXIST_HOME/autodeploy* folder will be loaded into the database at startup time.

While this is not a threat from within eXist, should someone compromise the user account under which you run the eXist server, he could potentially place a malicious package into the *autodeploy* folder that would be loaded the next time the database was restarted. Such a package might contain XQuery scripts that perform further nefarious actions, and these could potentially be invoked remotely.

You can disable the *autodeploy* folder by commenting out the `AutoDeploymentTrigger` line in *$EXIST_HOME/conf.xml*. For example:

```
<!-- trigger class="org.exist.repo.AutoDeploymentTrigger"/ -->
```

Subsequent to disabling the `AutoDeploymentTrigger`, you can also delete the *$EXIST_HOME/autodeploy* folder.

Removing preinstalled EXPath packages

By default, eXist ships with a number of EXPath packages providing libraries and applications that are installed by the autodeployment mechanism the first time eXist is started. These packages are excellent in a development environment, but less desirable in a production environment. Of particular concern are the dashboard and eXide applications, which provide facilities for remotely administering eXist through web pages.

If you have started eXist before disabling the autodeployment mechanism, as described in the previous section, you may remove these applications from a running eXist instance by executing the following XQuery (for example, from the Java Admin Client):

```
xquery version "1.0";

import module namespace repo = "http://exist-db.org/xquery/repo";

(:~
: This XQuery removes the preinstalled EXPath packages
: that ship with eXist 2.1
:)

declare variable $local:preinstalled-pkgs := (
    "http://exist-db.org/apps/shared",
    "http://exist-db.org/apps/dashboard",
    "http://exist-db.org/apps/eXide"
);

for $pkg in $local:preinstalled-pkgs return (
    repo:undeploy($pkg),
    repo:remove($pkg)
)
```

Securing eXist's network services

All of the network services and APIs discussed in "Disabling network services and APIs" on page 181 rely on an underlying Java application server to actually connect them to incoming network requests. eXist uses the Jetty application server by default, but can be deployed into any Java application server of your choosing.

eXist ships with Jetty, servicing incoming HTTP and XML-RPC requests on TCP port 8080 and HTTPS and XML-RPC over SSL requests on TCP port 8443. This configuration is held in the Jetty configuration file *$EXIST_HOME/tools/jetty/etc/jetty.xml*.

In a production environment, it is recommend that you disable the non-SSL interface and just use the SSL interface. By default eXist is configured with a temporary SSL certificate that serves to provide the encryption of a connection, but not verification of the server. If your users will connect directly to eXist by HTTPS or XML-RPC over SSL, then it is recommended that you generate or purchase your own genuine SSL certificate. The mechanics of SSL, SSL certificates, and configuring Jetty for custom certificates are considered beyond the scope of this book, and good online resources on these topics already exist, including Jetty's own documentation.

While we do not believe that "security through obscurity" adds any real security to a system, layering of protective mechanisms is always sensible. So, should you wish, you can also modify the TCP port numbers that eXist uses in the aforementioned Jetty configuration file.

Should you change the TCP port numbers (or disable the non-SSL protocols) that eXist uses, it is recommended that you also update the *$EXIST_HOME/client.properties* file. This is used by both the Java Admin Client and backup and restore scripts to connect to eXist when they are executed on the same server.

Reverse proxying

Reverse proxying is the process of placing a reverse proxy server between eXist and the client. The reverse proxy server receives a request from the client, creates a new request to eXist, and then returns the result it receives to the client (see Figure 8-21). The reverse proxy server may also add some optimization services such as caching, response compression, and load balancing.

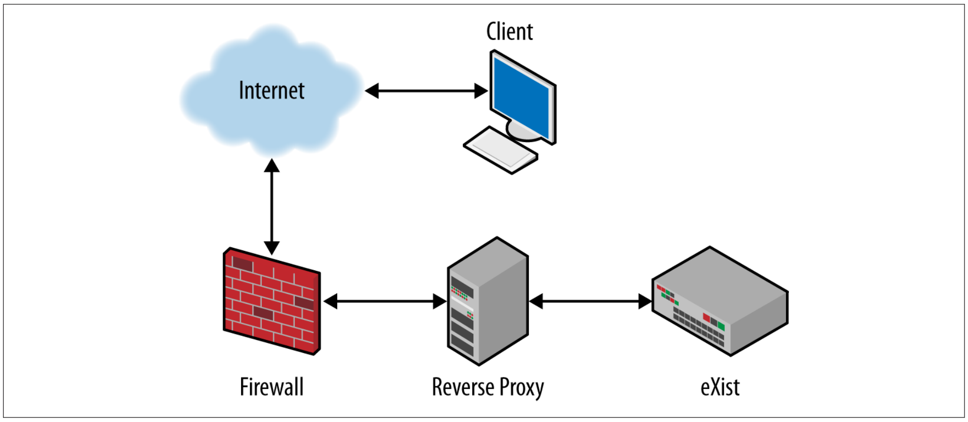


Figure 8-21. A reverse proxy network

While there are many good arguments for using reverse proxying in production environments, we will focus on the security aspect here. Many firewalls are capable of restricting access to a server for specific ports and protocols, but few firewalls operate at the application layer. A reverse proxy operates at the HTTP application layer (in this case), which means that it can filter based on characteristics of the HTTP requests.

The main security concerns for the reverse proxy server are:

Hiding the existence and characteristics of eXist from the client
: The client should only see the reverse proxy server. Should this be a malicious client and try to exploit the server directly, it will be exploiting the reverse proxy server and not eXist. Another concern, perhaps, is to allow the reverse proxy server to map an application or documents you have in eXist into your website's URI space, so that it is not obvious to attackers that part of your website is running in an application server (eXist)—for example, proxying the public URI *http://www.mywebsite.com/widgets/* to the private URI *http://my-exist-server:8080/exist/apps/widgets/*.

Controlling (and limiting) access to eXist by the client
: This allows you to limit which HTTP requests will reach eXist based on various criteria such as URI, headers, and cookies. A useful example is to limit access to the URIs */exist/xmlrpc* and */exist/webdav*, while allowing access to */exist/apps* from your web users.

There are many options for reverse proxy servers, but two of the most popular ones are the Apache HTTP Server Project (Apache httpd) and Nginx. Apache httpd is a large, feature-rich, and very popular web server, an example of whose use is described in "Proxying eXist Behind a Web Server" on page 207. Nginx (shown in Example 8-2) is a modern, very lightweight, and incredibly fast web server. It is perfect if you are looking for a reverse proxy for eXist because it is much simpler to install and configure than Apache httpd, lending it a smaller attack surface.

Example 8-2. Nginx configuration for reverse proxying eXist

```
proxy_set_header    Host              $host; ❶
proxy_set_header    X-Real-IP         $remote_addr; ❷
proxy_set_header    X-Forwarded-For   $proxy_add_x_forwarded_for; ❸
proxy_set_header    nginx-request-uri $request_uri; ❹

server {
    listen 80; ❺
    server_name .mywebsite.com; ❻
    charset utf-8;
    access_log /srv/www/vhosts/mywebsite.com/logs/access.log;
    location / {
        proxy_pass http://localhost:8080/exist/rest/db/mywebsite.com/; ❼
    }
}
```

❶ The first line, the `proxy_set_header` directive, takes the host header from the HTTP request sent from the client and sets it in the new HTTP request that Nginx sends to eXist. This can be useful for informing eXist which virtual host it is servicing.

❷❸ These `proxy_set_header` directives set new headers in the HTTP request that Nginx sends to eXist. eXist automatically uses these when behind a reverse proxy to understand the request from the client correctly.

❹ This `proxy_set_header` directive sets a new header in the HTTP request that Nginx sends to eXist, which allows a user in an XQuery to find the original URI requested by the client using `request:get-header("nginx-request-uri")`.

❺❻ Requests to any URI that starts **.mywebsite.com* on TCP port 80 (standard HTTP) will be proxied.

❼ Matched URIs are forwarded to the collection */db/mywebsite.com* in eXist via the REST Server, so *http://www.mywebsite.com/documents/2012* would be forwarded to *http://localhost:8080/exist/rest/db/mywebsite.com/documents/2012*.

User Authentication in XQuery

eXist provides an extension module of XQuery functions, called the *Security Manager module*, for handling security concerns programmatically. This module, discussed in the entry for `sm` in Appendix A, allows you to manipulate users, groups, and group managers, and also to manage both resource and collection permissions. However, the Security Manager module has a major intentional omission: if you wish to authenticate users in your own XQuery modules (perhaps in response to a login form), you can't do so through this module; rather, you have to use the `xmldb` module (see the entry for `xmldb` in Appendix A).

It may seem strange that authentication and login in XQuery are provided by the `xmldb` module and not the Security Manager module. The reason for this is that the Security Manager module is a relatively new addition to eXist. The developers felt that the login and authentication mechanisms from XQuery could be improved, and unfortunately it was not possible to accomplish this work before eXist 2.1 was released. It is likely that new and improved authentication and login mechanisms will be added to the Security Manager module in the future, deprecating those in the `xmldb` module.

xmldb:authenticate

The `xmldb:authenticate` function allows you to validate that a user has access to a collection in the database. The function takes three arguments: a collection URI, the username of a user, and the corresponding password of that user. It then performs two distinct steps:

1. It attempts to authenticate the user, using her username and password, with the Security Manager.
2. It attempts to open the collection given in the collection URI argument; that is to say, it verifies whether the authenticated user has execute access to the collection.

If either of the two steps fails, the function returns `false`; if both steps succeed in succession, then the function returns `true`.

This function is basic, but can be useful when you want to authenticate a user but do not wish to perform actions on that user's behalf.

xmldb:login

The `xmldb:login` function takes the same arguments as the `xmldb:authenticate` function and performs exactly the same steps. Assuming those steps succeed, it then performs two additional steps:

1. It changes the user account of the XQuery that called the function to be that described by the username passed to the function. Subsequently, the calling XQuery has the same access rights to resources and collections in the database as the newly logged-in user.
2. If the XQuery is operating in an HTTP context (i.e., executed via the REST Server, SOAP Server, or XQuery Servlet), then the logged-in user is cached in the HTTP session, if one exists (you can control creation of the session by calling the four-argument version of `xmldb:login` or using the `session:create` function explicitly). Subsequent XQuery invocations from within the same HTTP session will reuse the cached user from this login until the session is invalidated or the user logs out.

The `login` function is very useful when you want to present a user with a login form, authenticate him, and subsequently allow him to access and manipulate database resources and collections with the permissions he has previously been allocated.

You may be wondering how to control user logout after a user has logged into eXist. You have three options for this, each of which involves manipulating the user's HTTP session:

1. Remove the cached user credentials attribute from the HTTP session by calling `session:remove-attribute("_eXist_xmldb_user")`.
2. Clear all attributes of the user's HTTP session by calling `session:clear()`.
3. Invalidate the user's HTTP session by calling `session:invalidate()`. Invalidating the user's session means that you will have to create a new session if you wish to store further attributes in the session.

For further details on how HTTP sessions are managed, see the entry for `session` in Appendix A.

Backups

While the reasons for performing backups are not solely related to security but also to disaster recovery, the importance of backups cannot be stressed enough. First, ensure that you have backups, and secondly, ensure that you can restore your system from them! Whatever approach you take, backups should be something that you perform frequently, test regularly, and have confidence in as your last line of defense. For further information, see "Backup and Restore" on page 396.

CHAPTER 9

Building Applications

eXist is a great platform not only for storing and querying XML data, but also for building web applications. Returning pages in (X)HTML is a breeze, since XHTML *is* XML and XML is eXist's bread and butter. Conveniently, because your data is in XML and the programming language is XML-aware, there is no "impedance" between the layers (data storage, processing, presentation). Piece of cake, isn't it?

Well, maybe a marketing department (if one existed) would try to sell you this high-tech fairy tale. But as we all know, programming is hard work, whatever your environment or toolset. Silver-bullet environments don't exist.

However, that being said, eXist does represent an excellent platform on which to build web applications. For applications that work with XML (or can be redesigned to work with XML), increases in development speed and decreases in code base size are often realized when compared to undertaking the same projects in Java or PHP for example. It's not a panacea, but it can certainly help you in specific situations.

Overview

A web application is an application that is accessed over the Internet (or an intranet) and uses a web browser as its client. URLs passed from the browser are mapped to something on the server (a static page, a script that accesses a database, a CSS file, an image, etc.). Together this constitutes some meaningful functionality to the application's users, such as accounting, playing games, or making friends. An application residing on a server typically consists of multiple files/resources. Some of these are scripts that, once executed, do something like querying or updating a database. Others are static, like CSS files and images. Another category is internal data, which is not shown directly to the user but used internally for configuration, lookup tables, and more.

Which Technology to Use?

Somewhat confusing perhaps is that eXist has two very different technologies for mapping URLs in HTTP requests to functionality (e.g., executing XQuery or XSLT code or retrieving a particular resource). They are *URL rewriting* and *RESTXQ*:

- URL rewriting works by intercepting every HTTP request and passing it first through a centralized controller XQuery script that you provide. This script decides what to do with the request and either passes it on to another specific XQuery script, performs a redirect, or rejects it. Read more about this in "URL Mapping Using URL Rewriting" on page 194.
- RESTXQ works with XQuery 3.0 function annotations that tell eXist the function that must be executed when certain HTTP requests come along. Read more about this in "Building Applications with RESTXQ" on page 215.

URL rewriting is the older and more mature of the two. RESTXQ is younger and easier to use, but might not provide all the necessary functionality yet. The approaches cannot be mixed!

So, which one to choose? RESTXQ is probably the simpler approach to get started with, and also the more platform-independent choice. However, there are some limitations in RESTXQ at the moment:

- RESTXQ allows little nondeclarative access to the HTTP request: the eXist HTTP `request` module is not supported.
- RESTXQ has no session (in other words, the eXist `session` module is not supported).
- There is no support for processing HTTP multipart requests or responses in RESTXQ.

Therefore, you can handle more advanced tasks with URL rewriting (currently). However, few people need such advanced functionality, and missing features are likely to be added to RESTXQ in the near future.

Application Aspects

A web application is a multifaceted thing, and there are many aspects you have to deal with to make everything run smoothly. This is the case for all web technologies, and eXist is no exception. What are the eXist-specific aspects we have to talk about? In this chapter, we'll explore the following topics:

Where to store your files/resources
: Older versions of eXist gave you the choice of storing the application's files/resources in the database or on the filesystem. Although the option of using the

filesystem still exists, using the database is absolutely preferred. See "Where to Store Your Application?" on page 194.

The URL rewriting mapping mechanism
: How does the URL rewriting mechanism map HTTP requests to functionality? See "URL Mapping Using URL Rewriting" on page 194.

Cleaning up URLs for URL rewriting
: Up to now we have only seen ugly URLs like *http://localhost:8080/exist/rest/db/myapp/*. For a real application, you probably want to change these into something like *http://www.myapp.com/*. How to achieve this with URL rewriting is described in "Changing the URL for URL Rewriting" on page 205.

Requests, sessions, and responses
: Inside your application you'll want to inspect the requests, keep data alive between requests, and control the server's responses. Information about this can be found in "Requests, Sessions, and Responses" on page 209.

Security for applications
: How to handle the user base and add an extra layer of security using eXist's native mechanisms is described in "Application Security" on page 212.

Global error pages
: How to create specific pages that handle HTTP `400` (bad request), `404` (not found), and other response error codes is explained in "Global Error Pages" on page 215.

Using RESTXQ
: RESTXQ is substantially different from URL rewriting. You can read more about it in "Building Applications with RESTXQ" on page 215.

Packaging
: How to use the eXist packaging mechanism to easily distribute your application is described in "Packaging" on page 227.

Getting Started, Quickly?

This chapter will teach you the basic mechanisms for eXist applications: how they work and how to customize them to your needs. However, this might be too much information if you want to write an application quickly and are not really interested in what lies underneath.

In that case, we advise you to use eXide's (eXist's internal IDE's) application framework. This allows you to quickly set up small- to medium-sized applications without any fuss. Please refer to "eXide" on page 374 for more information.

Where to Store Your Application?

A real-world application consists of data, which for eXist is always stored in the database, and the application itself, which consists of many files—not only (XQuery) scripts but also images, stylesheets, static HTML pages, and more. In times past (v1.4 and before), eXist gave you a choice of where to store these files: either in the filesystem (underneath a subdirectory of *$EXIST_HOME/webapp/myapp*) or in the database.

Although the option to use the filesystem for your application is still supported, it has been deprecated for some time and should not be used. Security and deployment are the major drivers for this, and additionally, applications stored in the database are part of standard database backups. The database's security system lets you tag certain resources for general usage and others for use by specific users and/or groups. This provides an extra layer of security on top of what is built into the application logic. Deployment on a live database is a breeze; a simple backup/restore does the trick. If you need more functionality, packaging mechanisms are available (see "Packaging" on page 227). There are several tools that make working with resources in the database feel almost exactly like working with files on the filesystem (as you'll read about in Chapter 14). RESTXQ (see "Building Applications with RESTXQ" on page 215) only works from within the database. The only remaining issue is version control, but with a bit of scripting that can be solved too (for instance, using eXist Ant scripting, as described in "Ant and eXist" on page 379).

So, develop your application in the database and reap the benefits of the security and deployment mechanisms offered. Do not use the filesystem (any longer).

URL Mapping Using URL Rewriting

URL mapping is all about providing meaningful URLs to your users and keeping them consistent. For instance, in a wiki, you might want the user to visit a subject with a URL like *http://.../wiki/subjectname*. But, of course, there will not be an XQuery script for every subject. Likely, there will be a single script handling all subjects based on some parameter, like in *http://.../wiki/handlepage?subject=subjectname*.

eXist's oldest and most mature mechanism for mapping URLs to functionality is called *URL rewriting*. URL rewriting works by intercepting the HTTP request and passing control to a single entry point. This entry point is an XQuery script, always called *controller.xql*. Because it is XQuery, you can do whatever you want in it. It must return an XML fragment that describes what eXist should do next.

Anatomy of a URL Rewriting-Based Application

This section will take you through the anatomy of a mini demo application that uses URL rewriting. Although tiny, it shows you the important components and characteristics.

The demo application is part of the example code for this book; if you have installed that correctly you can start it with this URL (don't forget the terminating slash): *http://localhost:8080/exist/apps/exist-book/building-applications/mini-application/*.

Don't let the URL format annoy you. We'll talk about creating more user-friendly URLs soon, in "Changing the URL for URL Rewriting" on page 205.

You should see something like Figure 9-1.

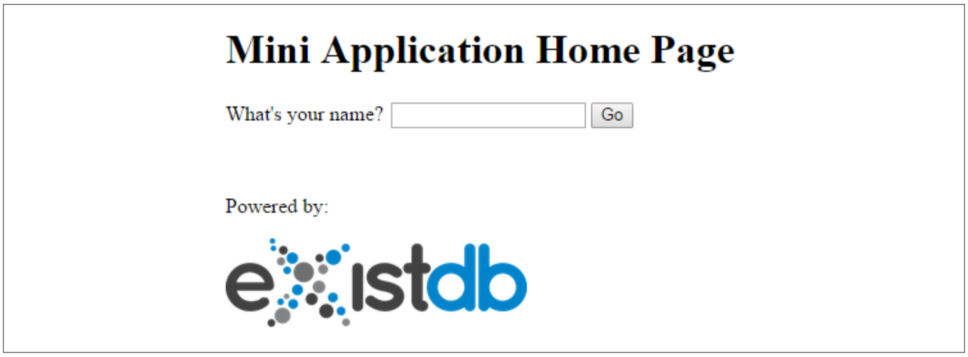


Figure 9-1. The home screen of our example mini application

Notice that the URL visible in your browser changed and now ends in */home*. It brings up a page generated by an XQuery script, but strangely enough, the URL doesn't end in *.xq*.

Typing your name and pressing Submit brings up a similar "Hello <*name*>" screen; nothing particularly fancy is going on. So what makes this a typical eXist URL rewriting application?

A URL rewriting–based application has a central XQuery script as a single point of entrance for *all* requests. This is always called *controller.xql* and located in the root collection of your application. For Example 9-1, it is in */db/apps/exist-book/building-applications/mini-application/controller.xql*.

Example 9-1. The URL rewriting controller code for the example application

```
xquery version "1.0" encoding "UTF-8";

(:~
: Example URL Rewriting Controller
:)

(: External variables available to the controller: :)
declare variable $exist:path external;
declare variable $exist:resource external;
declare variable $exist:controller external;

(: Other variables :)
declare variable $home-page-url := "home";

(: Function to get the extension of a filename: :)
declare function local:get-extension($filename as xs:string) as xs:string {
    let $name := replace($filename, ".*[/\\]([^/\\]+)$", "$1")
    return
        if(contains($name, "."))
        then replace($name, ".*\.([^\.]+)$", "$1")
        else ""
};

(: If there is no resource specified, go to the home page.
   This is a redirect, forcing the browser to perform a redirect. So this request
   will pass through the controller again... :)
if($exist:resource eq "")then
    <dispatch xmlns="http://exist.sourceforge.net/NS/exist">
        <redirect url="{$home-page-url}"/>
    </dispatch>

(: Check if there is no extension. If not, assume it is an XQuery file and forward
   to this. Because we use forward here, the browser will not be informed of the
   change and the user will still see a URL without a .xq extension. :)
else if (local:get-extension($exist:resource) eq "")then
    <dispatch xmlns="http://exist.sourceforge.net/NS/exist">
        <forward url="{concat($exist:controller, $exist:path, ".xq")}"/>
    </dispatch>

(: Anything else, pass through: :)
else
    <ignore xmlns="http://exist.sourceforge.net/NS/exist">
        <cache-control cache="yes"/>
    </ignore>
```

This example is intended to give you a first rough idea about what's going on. We're going to talk about URL rewriting in detail later, but here are some important characteristics:

- Notice that the result of the script is always an XML fragment in an eXist-specific namespace. This fragment determines what eXist will do next—for instance, redirect the browser to another page, or silently (and invisibly to the user) forward to some other URL. The examples here are quite simple, but you can do some amazingly complex things, like pipelining results through XLST stylesheets.
- The top of the script declares a number of external variables. eXist uses these to pass important information about the call to the script.
- There is a function that extracts the extension part from a filename (e.g., *xq* from *home.xq*), by using a regular expression. This is, of course, not unique to URL rewriting, but you'll often see regular expressions in a *controller.xql* inspecting parts of the URL.
- The main part of the script examines the request by inspecting the external variables. The first `if` clause determines if the URL contains a resource name. If not, it *redirects* the browser to the home page. This causes a browser redirect (visible because the URL visible in your browser changes). So again, an HTTP request travels through our URL rewriting controller, but now with */home* appended.
- The second `if` clause checks whether the resource part of the URL has an extension (like *.xq* or *.png*). If not, it assumes that it is an XQuery script and *forwards* the request, invisibly to the browser, to an XQuery script in the database. So, for example, the second time around, after */home* has been appended to the URI, *home* is interpreted as `home.xq`, and therefore *home.xq* is called.
- The last `else` is a catchall that passes on the full URL to the appropriate handler, necessary for displaying images and more.

You'll also want to look at the security settings of the application files. You can view these by, for instance, using eXist's Java Admin Client:

- Other users have execute permissions for *controller.xql*. This will always be the case for a URL rewriting controller, because it is the general entry point to your application. Otherwise, this is the error message you'll see:

  ```
  Subject 'guest' does not have '--------x' access to resource
  '/db/apps/exist-book/building-applications/mini-application/controller.xql'
  ```

- For our application, *other* users have execute permissions for other XQuery files too, so anybody can use them.
- Other users also have execute permissions for the *images* subcollection. If you were to revoke this permission, the browser would not be able to load the eXist logo.

- Finally, other users have read permissions to view the eXist logo in the *images/existdb.png* file. This makes it viewable by everyone.

For more restricted applications, you could limit these execute and other permissions to certain database users and/or groups. It's easy to provide the user with a login page that changes its database identity, allowing you to fine-tune access. See "Application Security" on page 212 for more about this.

Our last comment on this example involves inspecting the request and passing parameters. When you have a look at the code of the *hello.xq* file, you can see that it calls the eXist extension function `request:get-parameter` to read the name of the person invoking the request:

```
<p>Hello <i>{request:get-parameter('personname', '?')}</i></p>
```

The `request` extension module can get you a lot more information; see "The request Extension Module" on page 209.

How eXist Finds the Controller

To test a URL controller, you can use the URL *http://localhost:8080/exist/apps/<path-toyourapp>*, as in the beginning of the previous section. We'll do a sneak preview here of information to come (in "The controller-config.xml Configuration File" on page 206) to make sure you understand how this works and how eXist finds the controller:

eXist has a configuration file called *$EXIST_HOME/webapp/WEB-INF/controller-config.xml*. In it are entries like this:

```
<root pattern="/apps" path="xmldb:exist:///db/apps"/>
```

When you request a page that starts with *http://localhost:8080/exist/apps/*, the following happens:

- Jetty recognizes the eXist prefix */exist* and passes control to the eXist main servlet.
- This servlet sees a URL starting with */apps*. It tries to match this with an entry in *controller-config.xml*.
- If a match is found, the value of its `path` attribute is used to try to locate a controller. So, in this case, eXist will first look for a controller in *xmldb:exist:///db/apps/controller.xql*.
- If no match is found, it uses the rest of the URL to try to find the controller. It starts at the most specific path and works backward until it finds a controller.

So, for instance, a URL ending with */apps/myapp/a/b/c.xq* will have eXist looking for a *controller.xql* file in */db/apps/myapp/a/b*, */db/apps/myapp/a*, and finally in */db/apps/myapp* (where it will most probably be).

- If eXist does not find a controller, it uses the URL as a path into the database and tries to find a matching resource.

Only one controller will be applied to a given request. It is not possible to pass control from one controller to another (or back to the same).

However, be aware that when your controller asks for a *redirect* (using the `redirect` element, as discussed in "Redirecting the request" on page 201), the *browser* will fire a *new request* and the whole circus of finding and possibly running a controller will start again. This creates the potential for redirect loops, so be careful!

The URL Rewriting Controller's Environment

The URL rewriting controller in *controller.xql* gets information about the request through five external variables. You do not need to explicitly declare them, but if you do it should look like this:

```
declare variable $exist:path external;
declare variable $exist:resource external;
declare variable $exist:controller external;
declare variable $exist:prefix external;
declare variable $exist:root external;
```

Besides using the special controller external variables, you can also use the functions in the `request` extension module (see "The request Extension Module" on page 209) to find out more about the request and the URL.

If you want to play with these variables, the collection */db/apps/exist-book/building-applications/show-controller-variables* contains an example that passes the values of the external variables to the *show-controller-variables.xq* script, which displays them on an HTML page.

You can use this little application to inspect the values of the URL rewriting controller's external variables. Use your browser to visit *http://localhost:8080/exist/apps/exist-book/building-applications/show-controller-variables/<any-path-you-like>*, and the values of the variables will be displayed.

Here are the definitions of the variables. For the examples, we assume you have browsed to *http://localhost:8080/exist/apps/exist-book/building-applications/show-controller-variables/a/b/c.xq*:

`$exist:path`
: The part of the URL after the part that led to the controller. For example: *a/b/c.xq*.

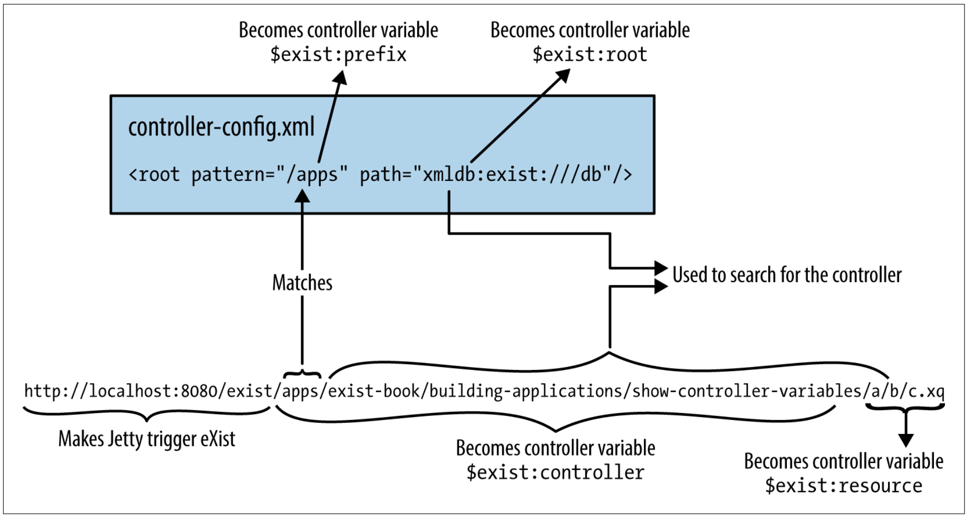
`$exist:resource`
: The part of the URL after the last / character, usually pointing to a resource. For example: *c.xq*.

`$exist:controller`
: The part of the URL leading from the prefix (see below) to the controller script. For example: */exist-book/building-applications/show-controller-variables*.

`$exist:prefix`
: The URL prefix that caused the URL rewriting controller to become active. This is defined in the *controller-config.xml* configuration file. For example: */apps*.

`$exist:root`
: The root path used for finding the controller, as defined in the *controller-config.xml* configuration file. This path can be on the filesystem or in the database. In our example it is *xmldb:exist:///db*.

Figure 9-2 summarizes all this.



Becomes controller variable $exist:prefix
Becomes controller variable $exist:root
controller-config.xml
<root pattern="/apps" path="xmldb:exist:///db"/>
Used to search for the controller
Matches
http://localhost:8080/exist/apps/exist-book/building-applications/show-controller-variables/a/b/c.xq
Makes Jetty trigger eXist
Becomes controller variable $exist:controller
Becomes controller variable $exist:resource

Figure 9-2. Going from a URL to controller variables

The Controller's Output XML Format

A URL rewriting controller must output an XML fragment. This fragment determines what eXist will do next.

Ignoring the request

If you don't want the controller to do anything and simply pass the request on for normal processing, either output nothing or use an `ignore` element. Skipping any URL rewriting is mostly used for "miscellaneous" requests, like for images or stylesheets. The format is:

```
<ignore xmlns="http://exist.sourceforge.net/NS/exist">
  cache-control?
</ignore>
```

Cache control is explained in "URL rewrite caching" on page 202. When a request is ignored, cache control is usually on.

Redirecting the request

If you want the controller to *redirect the client* to another URL, use a `dispatch` element with a `redirect` child element. This will cause the client to issue a new request, potentially triggering the controller again. The format is:

```
<dispatch xmlns="http://exist.sourceforge.net/NS/exist">
  redirect
  cache-control?
</dispatch>
```

The `redirect` element is defined as:

```
<redirect url = string >
```

Cache control is explained in "URL rewrite caching" on page 202.

Forwarding the request

If you want the request forwarded to a specific resource *on the server*, use a `dispatch` element with a `forward` child. The format is:

```
<dispatch xmlns="http://exist.sourceforge.net/NS/exist">
  <forward url = string ❶
           servlet? = string ❷
           absolute? = "yes" | "no" ❸
           method? = "POST" | "GET" | "PUT" | "DELETE" > ❹
    ( add-parameter | set-attribute | clear-attribute | set-header )*
  </forward>
</dispatch>
```

❶ `url` directs the request to a new request path. This is equivalent to directly requesting this path, but without a controller present.

A relative path will be resolved relative to the original request path.

An absolute path will be resolved relative to the path that triggered the controller. For example, if the original URL started with *http://localhost:8080/exist/apps/...*

and a forward was done to */ui/login.xq*, the resulting request would be to *http://localhost:8080/exist/apps/ui/login.xq*.

❷ `servlet` passes control to another servlet. Read more about this in "Advanced URL Control" on page 203.

❸ If `absolute` is set to `"yes"`, interpret the `url` attribute as a path on the filesystem, relative to the *$EXIST_HOME/webapp* directory, even when the controller is stored in the database. The default is `"no"`.

For instance, `<forward url="/extra/admin.xq/" absolute="yes"/>` will forward control to *$EXIST_HOME/webapp/extra/admin.xq*.

❹ `method` sets the HTTP method to use when passing the request to the next step in a pipeline. More about pipelines can be found in "Advanced URL Control" on page 203. The default is `"POST"`.

A `forward` element can contain the following additional children:

- The `add-parameter` element lets you add or override a request parameter:

```
<add-parameter name = string
               value = string />
```

- The `set-attribute` element sets a request *attribute*:

  ```
  <set-attribute name = string
                 value = string />
  ```

  You can inspect request attributes through eXist's `request` extension module. There's more about request attributes and how they differ from parameters in "The request Extension Module" on page 209.

- The `clear-attribute` element clears a request attribute:

  ```
  <clear-attribute name = string />
  ```

  In rare circumstances this is necessary when constructing a pipeline. Read more about pipelines in "Advanced URL Control" on page 203.

- The `set-header` element sets an HTTP header field:

```
<set-header name = string
            value = string />
```

URL rewrite caching

You can enable URL rewrite caching by adding a `cache-control` child element to the `dispatch` element:

```
<cache-control cache = "yes" | "no" />
```

Setting the cache attribute to "yes" adds an entry for the dispatch rule to an internal map and prevents the controller from being triggered again for the input URL. For instance:

```
<dispatch xmlns="http://exist.sourceforge.net/NS/exist">
  <redirect url="home"/>
  <cache-control cache="yes"/>
</dispatch>
```

URL rewrite caching has nothing to do with HTTP caching; only the dispatch *rule* is cached, not the response.

Advanced URL Control

URL rewriting is capable of more than just passing on or redirecting a request. It can also pass on the results of a forwarded request to a *pipeline* (a.k.a. sequence or view) of additional processing steps (usually XQuery and/or XSLT scripts).

The most common use case for this is probably the Model-View-Controller or MVC pattern, separating the application logic from its presentation. In the case of URL rewriting, *controller.xql* is the controller in the MVC pattern. Then we create an XML document, describing the contents of the response (but not its presentation). This becomes the model in the MVC pattern. Subsequent processing steps add the presentation to this, usually by transforming it to (X)HTML. This is the view in the MVC pattern.

URL rewriting allows you to specify such actions in the XML fragment output of the URL rewriting controller. To do this, add a view element after the forward element, containing the additional processing steps.

Here is a simple example of such an XML fragment. You can see this example in action by browsing to *http://localhost:8080/exist/apps/exist-book/building-applications/views/*:

```
<dispatch xmlns="http://exist.sourceforge.net/NS/exist">
  <forward url="{concat($exist:controller, "/createmodel.xq")}"/>
  <view>
    <forward servlet="XSLTServlet">
      <set-attribute name="xslt.stylesheet"
          value="{concat($exist:root, $exist:controller, "/xslt/view1.xslt")}"/>
    </forward>
  </view>
</dispatch>
```

In this example, the request is first passed to the *createmodel.xq* script. This creates some XML that is subsequently passed to the *view1.xsl* XSLT stylesheet for transformation into HTML.

Another example uses two stylesheets in a pipeline:

```
<dispatch xmlns="http://exist.sourceforge.net/NS/exist">
  <forward url="{concat($exist:controller, "/createmodel.xq")}"/>
  <view>
     <forward servlet="XSLTServlet">
      <set-attribute name="xslt.stylesheet"
         value="{concat($exist:root, $exist:controller, "/xslt/view2a.xslt")}"/>
    </forward>
    <forward servlet="XSLTServlet">
      <set-attribute name="xslt.stylesheet"
         value="{concat($exist:root, $exist:controller, "/xslt/view2b.xslt")}"/>
    </forward>
  </view>
</dispatch>
```

This will first create XML by calling *createmodel.xq*. This is passed to the *view2a.xslt* XSLT stylesheet and processed into something else. Finally, the *view2b.xslt* XSLT stylesheet which transforms it into HTML.

We pass the name of the stylesheet by setting the `xslt.stylesheet` request attribute. Notice that we do a bit of filename juggling there: `concat($exist:root, $exist:controller, "/xsl/view1.xslt")`. This is necessary because stylesheets are expected to be on the filesystem by default. To execute stylesheets from the database, we have to explicitly prepend their paths with `xmldb:exist:///db/`, and `$exist:root` starts with this. You can, of course, hardcode this, but eXist passes enough information in the controller variables to build this path dynamically, which somewhat isolates you from possible changes in future.

eXist has multiple servlets, but the one that is useful in this scenario is the XSLT servlet, named `XSLTServlet`. It is controlled by means of the following attributes:

`xslt.stylesheet`
: The path and name of the XSLT stylesheet to execute. By default, the filesystem is used. If you want to use a stylesheet stored in the database, prepend this value with `xmldb:exist:///db/`.

`xslt.user`, `xslt.password`
: The username and password of a database user, used during execution of the XSLT script when it accesses the database.

`xslt.*`
: Any other attributes starting with `xslt.` will be passed as stylesheet parameters. For instance, an attribute called `xslt.extra` will be available to the stylesheet as

global parameter `$xslt.extra`. Not all XDM types are supported, so it's best to limit yourself to strings.

Changing the URL for URL Rewriting

We've only seen ugly URLs for referencing our application so far, like:

http://localhost:8080/exist/rest/db/myapp/
: For a resource stored in the database underneath */db/myapp*

http://localhost:8080/exist/apps/building-applications/
: For an application with a URL rewriting controller stored in the database underneath */db/apps/myapp*

It's time to clean up our act and make way for nice URLs like *http://localhost/myapp* that use port 80 and don't need the */exist* prefix—or even better, use a DNS name like *http://www.myapp.com/*.

Changing the URL has everything to do with how eXist processes a URL:

- The Jetty web server is the main receiver of the request. It listens on a certain TCP port (by default, 8080) for HTTP requests. It examines the request and, based on its URL, passes it on to a servlet. The default configuration tells Jetty that *all* requests (with a URL starting with */exist*) should be passed to the `XQueryUrlRewrite` servlet, serving as the central entry point.
- The `XQueryUrlRewrite` servlet matches the remainder of the URL (the part after */exist*) to entries in the mapping file *$EXIST_HOME/webapp/WEB-INF/controller-config.xml*. This tells `XQueryUrlRewrite` what to do: look for a URL rewriting controller somewhere or pass it directly to another servlet.
- If a URL rewriting controller is involved, it inspects the URL and passes control for further processing (or tells the browser to redirect to another page).

We talked about this last step first (see "URL Mapping Using URL Rewriting" on page 194), because it's so crucial for understanding how applications work in eXist. Now we're going to talk about the first two stages.

Changing Jetty Settings: Port Number and URL Prefix

The Jetty settings determine the TCP port number used (by default, 8080) and the prefix of the URL (by default, */exist*). These settings are configured in *$EXIST_HOME/tools/jetty/etc/jetty.xml*.

Change TCP port number
: To change the TCP port number eXist listens on, find the following entries, change the port numbers, and restart eXist:

```
<SystemProperty
          name="jetty.port" default="8080"/>

<SystemProperty
          name="jetty.port.ssl" default="8443"/>
```

Be aware that the second entry appears twice!

On Unix and Linux systems, only a process running under the `root` user's account can open ports beneath 1024. While web servers typically operate on port 80 and/or 443, as discussed in Chapter 8, it is better to run eXist as an unprivileged user (see "Hardening" on page 174) and instead reverse proxy eXist through an existing web server (see "Reverse proxying" on page 185).

URL prefix
: To remove the */exist* URL prefix, find the entry `<Set name="contextPath">/exist</Set>`, change its value to `/`, and restart eXist.

The controller-config.xml Configuration File

The next step eXist takes is examining the remainder of the URL (the part after the URL prefix, if any). This is done by the `XQueryUrlRewrite` servlet using the entries in the *$EXIST_HOME/webapp/WEB-INF/controller-config.xml* file.

If you want to, you can change the location of the *controller-config.xml* file, even to somewhere *inside* the database. This can be beneficial for security or backup reasons.

Open *$EXIST_HOME/webapp/WEB-INF/web.xml* and search for the entry that mentions *controller-config.xml*. It should look like `<param-value>WEB-INF/controller-config.xml</param-value>`. Change this to, for instance, `<param-value>xmldb:exist:///db/controller-config.xml</param-value>` and store your *controller-config.xml* in the `/db` database collection. Restart eXist.

Example 9-2 is a simplified and annotated version of this file.

Example 9-2. Example controller-config.xml file

```
<configuration xmlns="http://exist.sourceforge.net/NS/exist">
  <!-- Forward URLs starting with rest or servlet to the REST servlet: -->
  <forward pattern="/(rest|servlet)/" servlet="EXistServlet"/>
  <!-- Patterns starting with /apps should look for a URL rewriting controller: -->
  <root pattern="/apps" path="xmldb:exist:///db/apps"/>
  <!-- My url www.myapp.com should map to my application stored underneath
```

```
        /db/myapp in the database: -->
  <root server-name="www.myapp.com" pattern="/*"
    path="xmldb:exist:///db/apps/myapp/"/>
  <!-- Anything else, pass on to the XQueryServlet for default executing
        from the filesystem: -->
  <forward pattern=".*\.(xq|xql|xqy|xquery)$" servlet="XQueryServlet"/>
</configuration>
```

The content of the *controller-config.xml* file must be in the `http://exist.sourceforge.net/NS/exist` namespace. The format is:

```
<configuration xmlns="http://exist.sourceforge.net/NS/exist">
  ( forward | root )+
</configuration>
```

What happens is that all entries in the *controller-config.xml* file are examined from top to bottom. If the remainder of the URL (the part after */exist*) matches with a `pattern` attribute (which is a regular expression), this entry is used.

- A `forward` element passes control directly to a given servlet:

  ```
  <forward pattern = string
           servlet = string />
  ```

- A `root` element triggers the URL rewriting controller:

  ```
  <root pattern = string
        server-name? = string
        path = string />
  ```

  The `path` attribute tells eXist where to look for the URL rewriting controller, as explained in "How eXist Finds the Controller" on page 198. The default location is within the filesystem, but if you want it to point to a location in the database, start its value with `xmldb:exist:///db/`.

  When a `server-name` attribute is present (e.g., `server-name="www.myapp.com"`), this must match also, allowing you to associate a DNS name with your application.

Proxying eXist Behind a Web Server

Another way of cleaning the URLs is by running eXist behind another web server, as a proxy. This web server—we'll use Apache as an example—catches requests for eXist, passes them on, and sends the responses back to the user.

Although this sounds like a bit of a detour, it is actually quite useful in certain situations:

- Sometimes you're running on a server in a mixed environment. Besides the eXist application there can be other applications active, based on PHP, CGI, Perl, and more. The easiest way to handle this is to use a workhorse like Apache and proxy eXist behind Apache.
- Apache is more flexible in configuration than Jetty + eXist and contains more functionality as a web server.
- System managers are probably more used to running Apache as a frontend than Jetty. They have things like tools and scripts hanging around and know the commands by heart. Keep them happy!
- Apache is used more than Jetty, so there are lots of third-party tools for things like analyzing web traffic.

There is more than one way to handle this, but here is a recipe for proxying an eXist application behind Apache:

1. Leave the Jetty settings at the defaults (i.e., TCP port 8080 and a URL prefix of */exist*).
2. Adapt the *controller-config.xml* file so that the URL to your application points to the right collection (or directory). For example:

   ```
   <root server-name="www.myapp.com" pattern=".*"
     path="xmldb:exist:///db/myapp/"/>
   ```

3. Enable the `mod_proxy` module in Apache.
4. Add the configuration shown in Example 9-3 to Apache (this is a minimal example; you'll probably want to add logging and other functionality).

Example 9-3. Apache configuration for proxying eXist

```
<VirtualHost *:80>

    ServerAdmin your-admin-email@your-domain.com

    # The URLs for this application:
    ServerName www.myapp.nl
    ServerAlias myapp.com
    ProxyRequests Off
    <Proxy *>
        Order deny,allow
        Allow from all
    </Proxy>

    ProxyPass        / http://www.myapp.nl:8080/exist/
    ProxyPassReverse / http://www.myapp.nl:8080/exist/
```

```
    # Cookies must be adapted to allow the session mechanism to work:
    ProxyPassReverseCookiePath /exist /
    ProxyPassReverseCookieDomain localhost myapp.com

    RewriteEngine   on
    RewriteRule     ^/(.*)$     /$1   [PT]

</VirtualHost>
```

There was another example of proxying earlier in the book (using the Nginx web server), which focused on security; see "Reverse proxying" on page 185.

Requests, Sessions, and Responses

An entry to a web application starts with a *request* from a web client. A request consists of a URL but might also contain, for example, parameters or an uploaded file. In between requests you probably want to keep information for the current user in a *session*. The answer to a request is called a *response*, and there are several things you might want to control here too.

To work with requests, sessions, and responses, eXist uses extension modules. This section will provide you with an overview of the functionality found in these modules (for the full details, please refer to the function documentation browser). Along the way we'll reveal some tips and tricks.

The request Extension Module

All details about an incoming HTTP request can be accessed through the `request` extension module. This module is really just a very simple XQuery wrapper around the underlying `HttpServletRequest` Java class that eXist handles for you. For instance:

- `request:get-uri` will give you the original URI as received from the client.
- There are other functions for inspecting details, like `request:get-remote-port` for checking the TCP port number.
- The functions `request:get-parameter-names` and `request:get-parameter` give you access to the request parameters.
- `request:get-cookie-names` and `request:get-cookie-value` let you access the data stored in cookies.

Request parameters and attributes

If you browse the functions of the `request` extension module, you might notice that both request parameters and attributes are mentioned:

- A request *parameter* is a name/value pair that was passed in from the client—for instance, as part of the URL or as an input field of an HTML form. Parameter values are always strings.
- A request *attribute* is a name/value pair that was set on the server. This was most likely done by the URL controller (see "URL Mapping Using URL Rewriting" on page 194), but if needed you can do it anywhere in your code using the `request:set-attribute` function. Attribute values can be anything from simple strings to complex XML fragments.

  Request attributes are useful for internal communication between parts of your application code when processing a request. They are also used by some internal mechanisms as parameters to servlets (for an example of this, see "Advanced URL Control" on page 203).

Uploading files

The `request` extension module can also be used for uploading files to the server. For example, assume you want to upload a binary file to your server and store this in the database. The page that offers this functionality must contain a form with encoding type `multipart/form-data`, as in this HTML fragment:

```
<form enctype="multipart/form-data" method="post" action="upload1-process.xq">
  <p>Upload binary file:
    <input type="file" size="80" name="FileUpload"/>
    <br/>
    <input type="submit"/>
  </p>
</form>
```

Access to the uploaded file is via the `request:get-uploaded-file-data` function. You can store the result in the database by using the `xmldb:store` function, as in this XQuery fragment:

```
let $stored-file as xs:string? := xmldb:store($store-collection, $store-resource,
  request:get-uploaded-file-data($field-name), 'application/octet-stream')
```

This returns the path of the file as stored in the database. Other functions that might be of interest here are `request:get-uploaded-file-name` for getting the original file name and `request:get-uploaded-file-size` for getting the size of the file (and optionally rejecting it if it is too large).

The session Extension Module

A session represents the interaction with a server for a specific client over a period of time. You may store data in the session that is available across requests for the same client. Each client *may* have a distinct session with the server. A session is accessed with the `session` extension module. This module is really just a very simple XQuery wrapper around the underlying `HttpSession` Java class that eXist handles for you. Some usage hints:

- A session must be created with the `session:create` function. Lots of other functions that do something with a session create it implicitly for you, but it can never hurt to create it explicitly with `session:create`. If the session already exists, the call is ignored.
- A session can hold attributes that are name/value pairs. Attribute values can be anything from simple strings to complex XML fragments. Use the functions `session:set-attribute` and `session:get-attribute` to work with these.
- Sessions invalidate automatically after not being accessed for a certain amount of time. You can control this interval using the `session:get-max-inactive-interval` and `session:set-max-inactive-interval` functions.

The response Extension Module

You control the response to a request via the `response` extension module. This module is really just a very simple XQuery wrapper around the underlying `HttpServletResponse` Java class that eXist handles for you. Useful functionality here includes:

- Setting cookies with the `response:set-cookie` function
- Explicitly setting response headers and the overall status code with the `response:set-header` and `response:set-status-code` functions
- Redirecting the client to another page with `response:redirect-to`
- Streaming data directly to the output with the `response:stream` and `response:stream-binary` functions (useful for creating download functionality, as described next)

Creating "download XML file" functionality

Creating a download function for an XML file, in which the browser asks you where to store it instead of displaying it, is not as easy as it may sound. You have to trick the browser into believing the file is not XML. The following code fragment forces an XML download:

```
response:stream-binary(
  util:string-to-binary(
    util:serialize(<Hello/>, 'method=xml'),
    'UTF-8'
  ),
  'application/octet-stream',
  'download.xml'
)
```

- An XML fragment (here, simply `<Hello/>`) is forced into a string via `util:serialize`.
- This string is then forced into binary data via `util:string-to-binary`.
- This is passed to `response:stream-binary` with an Internet media type set to `application/octet-stream`.
- A filename (in this example, *download.xml*) is passed as the preferred filename for storage (the user can change this).

As a result, the browser sees a binary response, which it cannot display. It therefore asks the user where it should be stored.

Application Security

Unless you're creating a fully public website, your application will have to deal with security. Such functionality may include creating and maintaining a user base, managing login and logout, and restricting access to parts of the application to certain user groups.

The usual way to implement this for an eXist application is to concentrate the security checks in the central controller (see "URL Mapping Using URL Rewriting" on page 194). The controller can check the user's identity, restrict access, map URLs to different pages based on the user's credentials, and more. Because the controller handles it all, your other code can be relatively security-code-free and concentrate on what it should be doing.

To make this all happen you need some kind of user/group administration system, and you might be tempted to set up one of your own—just some XML file with users, passwords, and additional information. There are several functions that take care of this, allowing users to log in and storing the identity of the current user in the session. The application can work with this information when pages are requested to allow or deny access.

However, we strongly advise against this approach. eXist already has an excellent security system that allows you to create users, log them in, organize them in groups, restrict access to scripts and data based on their credentials, and so on. This security system and how to work with it are described in detail in Chapter 8.

If you base your application's security on top of eXist's security, you have to write, debug, and maintain less code. It also creates two levels of security:

- Your controller or other parts of your application can work with eXist's security settings through functions in the `xmldb` and `securitymanager` extension modules. This allows for programmatically asking questions like "Is this user allowed to execute this XQuery module?" or "Is the current user allowed to see this data?" If not, you could redirect the user to the appropriate error or login page.
- On top of that, eXist takes guard. So, if your application is flawed and tries to access a nonauthorized page or data file, this is simply not allowed.

Therefore, our advice is to base your application's security on top of eXist's security. Here are some tips and tricks:

- Create at least one specific user group for your application, and make all the application's users a member of this group. Nonpublic pages and data should be accessible by members of this group only. You can extend this mechanism with multiple user groups if your application needs more fine-grained authorization.
- When you log somebody in, check whether this user is a member of the right user group(s) first! Sometimes you have multiple applications running on the same server, and you don't want users of Application A being able to log in to Application B (and running into trouble afterward because the security settings won't allow them to do anything).

  Here is little login function that checks whether a user is part of a list of user groups before attempting the login:

  ```
  declare function local:login(
    $user-groups as xs:string*,
    $user as xs:string,
    $password as xs:string
  ) as xs:boolean
  {
    let $users-in-groups as xs:string* :=
      for $group in $user-groups return xmldb:get-users($group)
    return
      if(empty($user-groups) or ($user = $users-in-groups)) then
        xmldb:login('/db', $user, $password, true())
      else
        false()
  };
  ```

- A handy function for checking the access rights of the current user for a certain resource or collection is `sm:has-access`. You can check against a partial mode string like `r-x` or `x`. For instance:

```
if(sm:has-access('/db/myapp/securepage.xq', 'r-x')) then
  (: forward to this page :)
else
  (: redirect to error page :)
```

- There is no explicit logout function. The safest way to log out is to return the current user's identity back to `guest` and to invalidate the session:

```
xmldb:login('/db', 'guest', 'guest'),
session:invalidate()
```

Running with Extra Permissions

You've set up an application and paid special attention to security, so when a user runs an XQuery, it runs with minimum permissions and is not allowed to access those parts of the database that it doesn't need to. However, suddenly you realize this user has to create/update the user base, a global logfile, or some other part of the database you don't want to make accessible in normal circumstances. What to do?

This is a frequently occurring problem. Luckily, eXist allows you to switch to another user for a single XQuery statement (which can, of course, also be a function call, so you can do whatever complicated stuff you like).

The function call for this is in eXist's `system` extension module:

```
system:as-user($username as xs:string, $password as xs:string?,
  $code-block as item()*) as item()*
```

`system:as-user` runs `$code-block` with the credentials of the given user. It returns whatever `$code-block` returns.

So, you set up a user with enough privileges and run the offending command with `system:as-user`. For example, the following creates a new user group called `appusers` with a member `erik`:

```
let $create-group-result := system:as-user('privuser', 'verysecret',
  xmldb:create-group('appusers', 'erik') )
```

As you probably have noticed, this creates a new security problem: you'll have to provide the `system:as-user` function with the username and password of a privileged user, so this data must be defined somewhere in your XQuery code or read from a data file. Unfortunately, there is not (yet) a watertight solution for this. The best you can do now is store this information somewhere in the database and set the security measures for the resource as tight as possible.

Global Error Pages

When something goes wrong, eXist generates an error page with the appropriate HTTP status code—for instance, a page with status 500 for an XQuery script that contains an error, or a status 404 for a nonexistent resource. You might want to prevent the user from seeing this and redirect these error responses to some kind of "Oops, sorry" page.

Unfortunately, eXist has no means of defining these kinds of error pages on an application level. You can only define them at the Jetty level, making them global for the full eXist instance.

To add an error page, edit the *$EXIST_HOME/webapp/WEB-INF/web.xml* file and add the following XML fragment as a child of the root web-app node:

```
<error-page>
  <error-code>http-error-code</error-code> ❶
  <location>uri-to-error-page</location> ❷
</error-page>
```

❶ The error-code element contains the integer HTTP status code you want to catch (e.g., 500 or 404).

❷ The location element contains the URL to the page you want to display if such an error pops up. This must be the part of a valid eXist URL that comes after */exist*. For instance: */rest/db/central/page404.xq*.

So, if your *web.xml* file contains:

```
<error-page>
  <error-code>404</error-code>
  <location>/rest/db/central/page404.xq</location>
</error-page>
```

all responses with an HTTP 404 status code will be forwarded to the *page404.xq* script.

Note that you have to restart eXist for the changes to take effect.

Building Applications with RESTXQ

RESTXQ is a standard developed by the EXQuery community that allows you to declare interactions between HTTP requests and XQuery functions. RESTXQ takes a very different approach from that of XQuery URL rewriting in eXist, instead using XQuery 3.0 annotations to declare your HTTP intentions within function declarations. XQuery functions that declare RESTXQ annotations are known as *resource functions* due to the fact that they expose some sort of resource over HTTP.

RESTXQ was inspired by the JAX-RS specification JSR-311. RESTXQ attempts to be nondisruptive by allowing you to annotate existing functions, which will then become HTTP-aware in a web-enabled XQuery processor or continue to work fine in a standalone processor. While XQuery URL rewriting is specific to eXist, RESTXQ attempts to create a standard XQuery 3.0 approach to servicing HTTP requests with XQuery, thereby allowing you to execute your XQuery web applications on any RESTXQ-compatible XQuery processor.

RESTXQ is a relatively young project: an implementation for eXist started in early 2012, and a beta version became part of eXist 2.0. Progress is still being made toward a final RESTXQ 1.0 version, but it is already very usable in eXist and many people are doing so. There are also implementations available in BaseX, Zorba, and MarkLogic.

RESTXQ offers great potential, and need not be solely limited to HTTP in the future; ultimately, RESTXQ might enable XQuery URL rewriting and the REST Server to be reimplemented in XQuery as a set of resource functions. Documentation for RESTXQ is fairly limited at the moment, and the best source of information is most likely Chapter 4 of the paper "RESTful XQuery" from the conference proceedings of XML Prague 2012.[1] Next, we will demonstrate how to make use of the features of RESTXQ, and you'll find a more complete example of using it in "RESTXQ" on page 353.

Configuring RESTXQ

RESTXQ monitors the eXist database, and when XQueries are stored that contain RESTXQ annotations, RESTXQ is configured to route matching HTTP requests to the identified resource functions. RESTXQ accomplishes this monitoring by means of a *trigger*, which is enabled by default on all database collections via the collection configuration in */db/system/config/db/collection.xconf*. You may enable or disable RESTXQ monitoring by adding its trigger configuration to or removing it from the configuration for a specific collection. For more details, see "System Collections" on page 90 and "Database Triggers" on page 449.

1 Adam Retter, "RESTful XQuery: Standardised XQuery 3.0 Annotations for REST," XML Prague 2012—Conference Proceedings (2012): 91-123.

RESTXQ maintains a registry of resource functions that it has detected. In eXist this registry is persisted on disk in the file *$EXIST_HOME/webapp/WEB-INF/data/restxq.registry*. You can see the list of known resource functions by looking in this file or by executing the XQuery function `rest:resource-functions`. While the format of this file is plain text, it is not recommended that you modify the file manually. However, removing this file when eXist is not running can be a good way to clear out the RESTXQ registry during development and testing.

When eXist starts up, RESTXQ reads its registry of known resource functions by means of a *startup trigger* (see "Startup Triggers" on page 446), which is enabled globally by default in *$EXIST_HOME/conf.xml*. Disabling this startup trigger along with removing references to RESTXQ from all collection configuration documents effectively disables RESTXQ in eXist.

All RESTXQ resource functions are relative to an implementation-defined *base URI*. In eXist, the default base URI is typically */restxq* (i.e., *http://localhost:8080/exist/restxq*). You may reconfigure the base URI by making changes to the forward pattern for `RestXqServlet` in *$EXIST_HOME/webapp/WEB-INF/controller-config.xml*. If you wish to map this into an existing domain space, one option would be to use reverse proxying, as described in "Proxying eXist Behind a Web Server" on page 207 and "Reverse proxying" on page 185. This would, for example, allow you to map *http://www.something.com/customer/1234* on to *http://localhost:8080/exist/restxq/customer/1234* (assuming that you have a resource function with a path annotation like `%rest:path("/customer/{$id}")`).

RESTXQ Annotations

RESTXQ defines a set of XQuery 3.0 annotations that, when added to an XQuery function, produce a resource function. This resource function can service an HTTP request and return an HTTP response. The exact mechanics of marshaling and demarshaling HTTP to XQuery are implementation-specific; RESTXQ just defines how various HTTP properties should be mapped into and out of an XQuery function. RESTXQ annotations can be used on any XQuery function; that is, functions in a main module or library module.

RESTXQ provides two classes of XQuery 3.0 annotations for use on resource functions:

Constraint annotations
: Constraint annotations identify and limit the scope of HTTP requests that may be processed by a resource function. Constraint annotations allow you to specify, for example, the URI, HTTP method, and Internet media types that your function is interested in processing.

Parameter annotations
: Parameter annotations extract properties of an HTTP request (matching the constraint annotations) and inject the values as parameters to your resource function. Parameter annotations allow you to extract parameters from the URI query, HTTP header, HTTP cookie, and `POST`ed HTML forms.

HTTP method constraint annotations

A resource function may have one or more *method constraint annotations*. A method constraint annotation constrains the HTTP methods that a resource function may process. RESTXQ currently supports the HTTP methods `GET`, `HEAD`, `POST`, `PUT`, and `DELETE`. See Example 9-4.

Example 9-4. Simple resource function that services all incoming GET requests

```
xquery version "3.0";

module namespace ex = "http://example/restxq/1";

import module namespace rest = "http://exquery.org/ns/restxq";

declare
    %rest:GET ❶
function ex:not-found() {
    <result>The requested page could not be found!</result>
};
```

❶ A RESTXQ resource constraint annotation for the HTTP method `GET`.

The XQuery in Example 9-4 is perhaps the simplest example of using RESTXQ; it will simply return a response for *any* HTTP `GET` request to eXist's RESTXQ Server.

By storing the XQuery anywhere in the database and granting it *execute* rights, you may then access it by requesting by HTTP `GET` any URI under *http://localhost:8080/exist/restxq*. For example, using cURL:

```
$ curl -v http://localhost:8080/exist/restxq/any/thing/at/all
```

results in:

```
* About to connect() to localhost port 8080 (#0)
*   Trying ::1...
* Adding handle: conn: 0x7f8091007200
```

```
    * Adding handle: send: 0
    * Adding handle: recv: 0
    * Curl_addHandleToPipeline: length: 1
    * - Conn 0 (0x7f8091007200) send_pipe: 1, recv_pipe: 0
    * Connected to localhost (::1) port 8080 (#0)
    > GET /exist/restxq/any/thing/at/all HTTP/1.1
    > User-Agent: curl/7.32.0
    > Host: localhost:8080
    > Accept: */*
    >
    < HTTP/1.1 200 OK ❶
    < Date: Sun, 20 Oct 2013 13:01:17 GMT
    < Set-Cookie: JSESSIONID=bzhhe8x66jqb1wremvc814vah;Path=/exist
    < Expires: Thu, 01 Jan 1970 00:00:00 GMT
    < Content-Type: application/xml ❷
    < Transfer-Encoding: chunked
    * Server Jetty(8.1.9.v20130131) is not blacklisted
    < Server: Jetty(8.1.9.v20130131)
    <
    * Connection #0 to host localhost left intact
    <result>The requested page could not be found!</result> ❸
```

❶ Note the HTTP response is `200 OK`. This is not ideal for when we do not find a document; `404 Not Found` would be more appropriate!

❷ The response media type is `application/xml`, which is the default of RESTXQ.

❸ The result of the XQuery function.

Example 9-4 has some shortcomings, in that it only handles HTTP `GET` requests, it returns the wrong HTTP status code when a document is not found, and it assumes that the client wants an XML response when a document is not found. Example 9-5 shows an improved version.

Example 9-5. Simple resource function that creates an HTML response

```
xquery version "3.0";

module namespace ex = "http://example/restxq/2";

import module namespace rest = "http://exquery.org/ns/restxq";
declare namespace output = "http://www.w3.org/2010/xslt-xquery-serialization";

declare
    %rest:GET ❶
    %rest:HEAD
    %rest:POST
    %rest:PUT
    %rest:DELETE
    %output:method("html5") ❷
```

```
function ex:not-found() {
    ( ❸
    <rest:response>
        <http:response status="404"/> ❹
    </rest:response>
    , ❺
    <html> ❻
        <head><title>Document not found!</title></head>
        <body>
            <p>Sorry, we could not find the document that you requested :-(</p>
        </body>
    </html>
    )
};
```

❶ We declare that we wish to process all supported HTTP methods.

❷ We use an *output annotation* to specify that the body of the response should be serialized as HTML5. Output annotations are part of the RESTXQ specification and simply provide an annotation syntax for the XSLT and XQuery 3.0 Serialization specification.

❸ We start a *sequence*, which allows us to control the response from RESTXQ and provide a response body.

❹ We declare that the HTTP response code should be set to `404 Not Found`.

❺ Note the comma that separates the first item in the sequence, which controls the RESTXQ response, from our response body (the second item in the sequence).

❻ We construct an HTML document for our response body.

Example 9-5 addresses the problems of Example 9-4 by handling all methods and explicitly defining properties of the HTTP response. We can store it into the database, replacing the first example, and then access it by requesting any URI (by any HTTP method) under *http://localhost:8080/exist/restxq*. For example, using cURL:

```
$ curl -v -X POST http://localhost:8080/exist/restxq/any/thing/at/all
```

results in:

```
* About to connect() to localhost port 8080 (#0)
*   Trying ::1...
* Adding handle: conn: 0x7f947b007200
* Adding handle: send: 0
* Adding handle: recv: 0
* Curl_addHandleToPipeline: length: 1
* - Conn 0 (0x7f947b007200) send_pipe: 1, recv_pipe: 0
* Connected to localhost (::1) port 8080 (#0)
```

```
> POST /exist/restxq/any/thing/at/all HTTP/1.1
> User-Agent: curl/7.32.0
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 404 Not Found ❶
< Date: Sun, 20 Oct 2013 13:29:14 GMT
< Set-Cookie: JSESSIONID=1dkp1kr1w2zdbrdjfycz9qtaa;Path=/exist
< Expires: Thu, 01 Jan 1970 00:00:00 GMT
< Content-Type: text/html;charset=UTF-8 ❷
< Transfer-Encoding: chunked
* Server Jetty(8.1.9.v20130131) is not blacklisted
< Server: Jetty(8.1.9.v20130131)
<
<!DOCTYPE html> ❸
<html> ❹
    <head>
        <title>Document not found!</title>
    </head>
    <body>
        <p>Sorry, we could not find the document that you requested :-(</p>
    </body>
</html>
* Connection #0 to host localhost left intact
```

❶ The HTTP response is now `404 Not Found`, as declared in our `rest:response`.

❷ The response media type is `text/html`, which is set by RESTXQ by default when an HTML output serialization is declared.

❸ The HTML5 `doctype` has been inserted by the HTML5 output serializer, as declared by our `output:method` annotation.

❹ The response body aspect of the sequence results from our XQuery function.

So far, each of the examples that we have looked at has used simple HTTP requests, but what happens when a `POST` or `PUT` request is received that contains a request body? If you wish to extract the body of the HTTP request, you can declare this intention on `POST` and `PUT` methods by specifying the name of the function parameter that the body should be injected into. See Example 9-6.

Example 9-6. Resource function extracting a request body

```
xquery version "3.0";

module namespace ex = "http://example/restxq/3";

import module namespace rest = "http://exquery.org/ns/restxq";
```

```
declare
  %rest:POST("{$body}") ❶
function ex:echo($body) { ❷
  <received>{$body}</received> ❸
};
```

❶ We declare that we wish to process only HTTP `POST` requests, and that any request body should be extracted and injected into the function parameter named `$body`.

❷ This parameter will be set to the value of the request body declared when invoked by RESTXQ.

❸ The request body will be output as part of the response.

By storing the XQuery anywhere in the database and granting it execute rights, you may then access it by requesting by HTTP `POST` any URI under *http://localhost:8080/exist/restxq*. For example, given the following simple XML file:

```
<test>123</test>
```

using cURL to `POST` the XML file:

```
$ curl -X POST -H 'Content-Type: application/xml' -d @/tmp/test.xml
http://localhost:8080/exist/restxq/something
```

results in:

```
<received>
  <test>123</test> ❶
</received>
```

❶ Note that the content of *test.xml* has been received by the server and placed inside the `received` element for the response by our XQuery function.

When extracting the HTTP request body for a `POST` or `PUT`, RESTXQ will attempt to automatically process the request body and provide the correct data type for you. The process for automatically converting the request body is as follows:

1. Is there an HTTP `Content-Type` header indicating that the content is of a binary type (looked up in *$EXIST_HOME/mime-types.xml*)? If so, return an `xs:base64Binary` value of the request body.
2. Try to parse the request body as XML; is it XML? If so, return it as a `document-node()` value.
3. Return the body as an `xs:string`.

URI path constraint annotation

A resource function may have a *path constraint annotation*, `%rest:path`, as shown in Example 9-7. A path constraint annotation constrains the URI path of an HTTP request that a resource function may process. The URI path may itself contain templates that are extracted and injected as parameters to the function. A URI path constraint may not be used by itself; it always requires at least one HTTP method constraint annotation to also be present on the resource function. The URI path is always relative to the base URI of the RESTXQ Server (see "Configuring RESTXQ" on page 216).

Example 9-7. Resource function saying hello with URI templating

```
xquery version "3.0";

module namespace ex = "http://example/restxq/4";

import module namespace rest = "http://exquery.org/ns/restxq";

declare
  %rest:GET
  %rest:path("/hello/{$name}") ❶
function ex:say-hello($name) { ❷
  <greeting>Hi there {$name}!</greeting> ❸
};
```

❶ We declare that we wish to only process paths that start with */hello* followed by a path segment template, which should be extracted and injected into the function parameter `$name`.

❷ This parameter will be set to the value of the URI template declared by *%rest:path* when invoked by RESTXQ.

❸ The value of the path segment will be output as part of the response.

This simple example declares the URI path to service, extracts a single URI segment using templating, and returns a result showing the value extracted from the URI.

By storing the XQuery anywhere in the database and granting it execute rights, you may then access it by an HTTP `GET` to the URI *http://localhost:8080/exist/restxq/hello/myName*. For example, using cURL:

```
$ curl http://localhost:8080/exist/restxq/hello/Liz
<greeting>Hi there Liz!</greeting>
```

Note the name has been extracted from the URI; changing the last segment of the URI changes the greeting!

URI paths may be much more complicated and have several templates within them; for example:

```
%rest:path("/country/{$country-code}/organization/{$org-id}/person/{$person-id}")
```

Consumes constraint annotation

A resource function may be constrained by the media types of HTTP requests that it is willing to process. You can achieve this by using one or more *consumes constraint annotations*, `%rest:consumes` (see Example 9-8). If no consumes constraint annotations are present on a resource function, then the function is assumed to process all content types. Consumes constraint annotations make the most sense in the context of `POST` and `PUT` requests, where you wish to control the `POST`ed/`PUT`ed resources that your resource function processes. A consumes annotation is compared against the `Content-Type` header from an incoming HTTP request.

Example 9-8. Resource function restricting request processing by Content-Type

```
xquery version "3.0";

module namespace ex = "http://example/restxq/5";

import module namespace rest = "http://exquery.org/ns/restxq";

declare
  %rest:POST("{$body}")
  %rest:consumes("application/xml", "text/xml") ❶
function ex:echo($body) {
  <received>{$body}</received>
};
```

❶ We declare that we wish to only process incoming HTTP requests that have a `Content-Type` of either `application/xml` or `text/xml`. You may specify as many media types as you wish within a consumes constraint annotation, or use multiple consumes constraint annotations.

Produces constraint annotation

The *produces constraint annotation*, `%rest:produces` (see Example 9-9), is the counterpart to the consumes constraint annotation: a resource function may be constrained by the media types that a client is willing to accept in an HTTP response. If no produces constraint annotations are present on a resource function, then the function is assumed to create a response that is compatible with any client. Produces constraint annotations are used for *content negotiation* scenarios, where the client informs the server which media types it accepts. A produces constraint annotation is compared against the `Accept` header from an incoming HTTP request.

Example 9-9. Resource function restricting request processing by Accept

```
xquery version "3.0";

module namespace ex = "http://example/restxq/6";

import module namespace rest = "http://exquery.org/ns/restxq";

declare
  %rest:POST("{$body}")
  %rest:consumes("application/xml", "text/xml")
  %rest:produces("application/xml") ❶
function ex:echo($body) {
  <received>{$body}</received>
};
```

❶ We declare that we wish to only process incoming HTTP requests that will accept a response of type `application/xml`. You may specify as many media types as you wish within a produces constraint annotation, or use multiple produces constraint annotations.

Parameter annotations

RESTXQ provides four different parameter annotations; however, their behavior is almost identical. It is the source of the parameter extraction that is the main difference. The annotations are:

Query parameters
: `%rest:query-param` extracts a parameter from the URI query string of the HTTP request. The value extracted may be an empty sequence (if the parameter is not present) or a sequence of one or more values, as it is possible to have URI query parameters with the same name and different values.

Header parameters
: `%rest:header-param` extracts a parameter from an HTTP header of the HTTP request. The value extracted may be an empty sequence (if the header is not present), or the value of the header.

Cookie parameters
: `%rest:cookie-param` extracts a parameter from a cookie in the HTTP header of the HTTP request. The value extracted may be an empty sequence (if the cookie is not present), or the value of the cookie.

Form field parameters
: `%rest:form-param` extracts a parameter from a `POST`ed or `GET`ed HTML form, so this can only be used in combination with `%rest:POST` and/or `%rest:GET`. The value extracted may be an empty sequence (if the form field is not present) or a

sequence of one or more values, as it is possible to have form fields with the same name and different values!

All of the parameter annotations have the same two forms:

- `%rest:source-param(parameter-name,function-parameter-reference)`
- `%rest:source-param(parameter-name,function-parameter-reference, default-value)`

The second form allows you to specify a default value to be injected into the named function parameter in case no matching parameter is available in the HTTP request (see Example 9-10).

Example 9-10. Resource function extracting request parameters

```
xquery version "3.0";

module namespace ex = "http://example/restxq/7";

import module namespace rest = "http://exquery.org/ns/restxq";

declare
  %rest:GET
  %rest:path("/hello") ❶
  %rest:query-param("name", "{$name}", "stranger") ❷
function ex:say-hello($name) { ❸
  <greeting>Hi there {$name}!</greeting> ❹
};
```

❶ We declare that we wish to only process paths that end with */hello* relative to the RESTXQ base URI.

❷ We declare that we wish to extract the value of the `name` URI query parameter and, if it is not available, to use the default value `stranger`. We declare that the value should be injected into the function parameter `$name`.

❸ This parameter will be set to the value of the URI query parameter declared by `%rest:path` when invoked by RESTXQ.

❹ The value of the URI query parameter will be output as part of the response.

By storing the XQuery anywhere in the database and granting it execute rights, you may then access it by an HTTP `GET` to the URI *http://localhost:8080/exist/restxq/hello*. For example, using cURL:

```
curl -v http://localhost:8080/exist/restxq/hello
```

results in:

```
<greeting>Hi there stranger!</greeting>
```

Note that the default value of `stranger` is provided in the response because we did not specify a `name` URI query parameter.

Alternatively, the following:

```
curl -v http://localhost:8080/exist/restxq/hello?name=Adam
```

results in:

```
<greeting>Hi there Adam!</greeting>
```

A more complete example of using RESTXQ can be seen in "RESTXQ" on page 353.

RESTXQ XQuery Extension Functions

RESTXQ attempts to take a minimal approach to providing extensions to XQuery, so it currently defines only three external XQuery functions:

`rest:base-uri() as xs:anyURI`
: Returns the base URI of the RESTXQ Server

`rest:uri() as xs:anyURI`
: Returns the URI of the HTTP request that led to the resource function being invoked

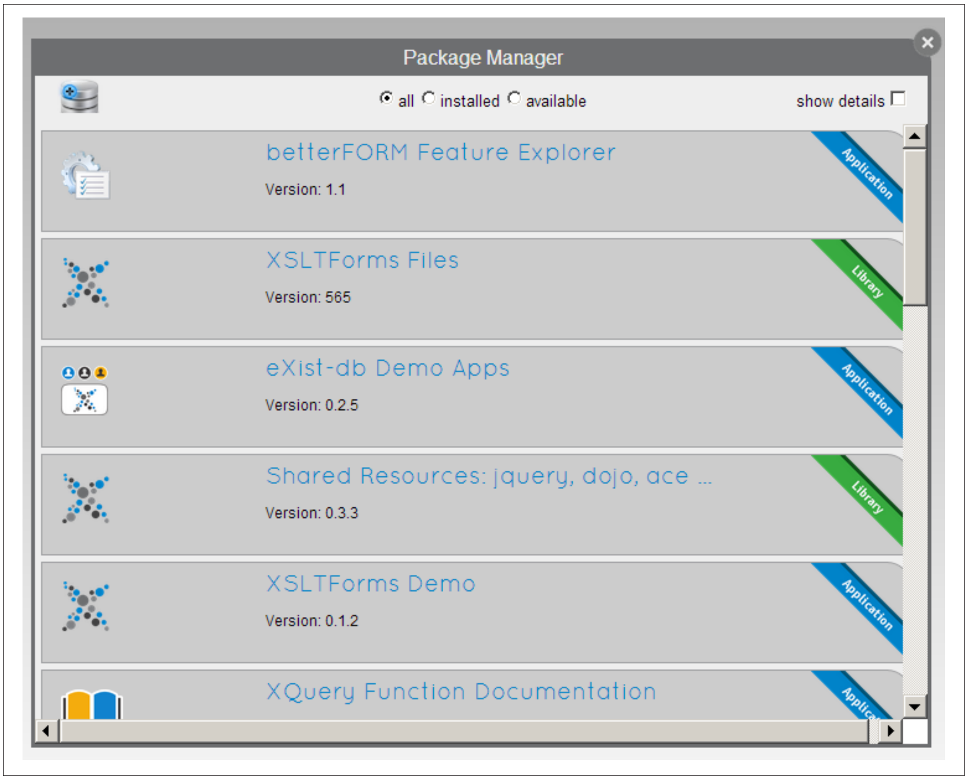
`rest:resource-functions() as document-node(element(rest:resource-functions))`
: Returns an XML document describing the resource functions that are known to the RESTXQ Server

Packaging

Once an application or library is finished, there is often a need to distribute it to others. For instance, something for the public should be easily distributable to and installable by everyone who wants to download and use it. Likewise more often than not, private applications need some kind of distribution too—for instance, moving from a development server to the test server, and after that to production.

To aid in this, eXist can work with *packages*. A package is an application (or library) bundled into a single archive ZIP file, together with machine-processable information on how to distribute and install it. You can work with packages through eXist's *Package Repository*, a core component of eXist since version 2.0. The Package Repository can install/uninstall, update, and launch packages.

eXist also contains an extension module, for use from within your XQuery scripts for working with the repository (see `repo`). Most of the time, however, you will work with the Package Repository through its user interface, the *Package Manager* (Figure 9-3), which is available through the dashboard.



betterFORM Feature Explorer
Version: 1.1

XSLTForms Files
Version: 565

eXist-db Demo Apps
Version: 0.2.5

Shared Resources: jquery, dojo, ace ...
Version: 0.3.3

XSLTForms Demo
Version: 0.1.2

XQuery Function Documentation

Figure 9-3. The eXist Package Manager

This whole packaging idea is based on the EXPath packaging system. This specification was designed to work across different XQuery implementations and is targeted at managing extension libraries (including XQuery, Java, or XSLT code modules). eXist extends this by adding a facility for the automatic deployment of entire applications into the database.

eXist packages come in two categories:

Applications
: An application is anything with a web interface. It will produce a tile on the dashboard and will start when the user clicks the tile.

Library packages
Library packages contain data, libraries, or resources used by other packages. They can also contain Java JAR files to load into eXist's classpath.

You might not have realized it, but you've used the result of the packaging system already quite intensively: most dashboard applications *are* packages. When you browse through the list of installed packages, you'll likely recognize some of them from the dashboard.

Examples

We need not provide you with extensive examples of packages in this section, because there are many practical examples already in existence that you can learn from:

- The example code for this book is distributed as a package, and you can look inside the package file or at the installed version in the database (in the */db/apps/exist-book* collection) to see how it is structured. Besides examples of all the packaging configuration files, there is some interesting code in the *installer* subcollection also.
- The directory *$EXIST_HOME/webapp/WEB-INF/data/expathrepo* holds the unpacked files of each installed package and is a good source of varied examples.

The Packaging Format

An eXist/EXPath package is a ZIP archive file, containing all the package's resources in directories which follow their collection structure. The file extension is, by convention, not *.zip* but *.xar*.

The root of the archive contains some packaging configuration files:

expath-pkg.xml
The standard EXPath descriptor file. It contains information on things like the package's name, version, and dependencies. See "The expath-pkg.xml file" on page 230.

repo.xml
The eXist-specific deployment descriptor file. It contains additional metadata and controls how the package will be deployed into the database. See "The repo.xml file" on page 231.

exist.xml
An eXist adaptation for loading extension modules written in Java. For more information about this, refer to the Package Repository documentation (available through the dashboard's Function Documentation browser).

The expath-pkg.xml file

The eXist-specific version of the EXPath descriptor file *expath-pkg.xml* is as follows. The full definition of *expath-pkg.xml* offers some further options, but these are currently ignored by eXist:

```
<package xmlns = "http://expath.org/ns/pkg" ❶
         name = uri ❷
         abbrev = string ❸
         version = string ❹
         spec = "1.0" > ❺
  title
  dependency*
  xquery*
</package>
```

❶ An *expath-pkg.xml* file is an XML document whose content is in the `http://expath.org/ns/pkg` namespace.

❷ `name` is a URI which is used to globally and uniquely identify the package.

❸ `abbrev` contains a short abbreviation for the package. Since the Package Manager uses this for filename creation, it is best to choose something without spaces and/or punctuation characters.

❹ `version` contains the version number or name of the package. To allow the Package Manager to work with this to its fullest extent, you should use what is called the *semantic version number format*: `x.y.z` (where `x`, `y`, and `z` are integers; e.g., `1.2.3`). See also the upcoming description of the `dependency` element.

❺ `spec` is the version of the EXPath specification and always contains, for now, `1.0`.

The child elements of the `package` root element are:

`title`
: The `title` element contains a descriptive title of the package. This is what will be displayed to the user in the dashboard.

`dependency`
: The `dependency` element defines other packages that this package is dependent on:

```
<dependency package = uri

  version = string
  - OR -
  semver = string
  - OR -
```

```
    semver-min = string
    semver-max = string
  />
```

The `package` attribute holds the URI of a package that this package depends upon (the value of the necessary package's `name` attribute from its `expath-pkg.xml`).

You can specify the version in one of three ways:

- Define the absolute version of the dependency with the `version` attribute.
- Define the version of the dependency in semantic version number format (*x.y.z*) using the `semver` attribute. This allows the packaging system, for instance, to select the highest version within a release (e.g., `semver="1.2"` will satisfy all versions starting with `1.2`, like `1.2.3`, `1.2.16`, etc.).
- Use one or both of the `semver-min` and `semver-max` attributes to set the minimum and maximum version number of the dependency using semantic version number format (*x.y.z*).

Using the semantic version number format is highly recommended.

`xquery`

Use the `xquery` child element to register one or more library modules with eXist. These modules then become globally available for your XQuery scripts, and your code or other packages can use them without knowing where they are stored. In other words, you don't have to use the `at` clause within the `import module` declaration for this module in your XQuery script's prolog:

```
<xquery>
  <namespace>namespace of xquery module</namespace>
  <file>filename without path of xquery module</file>
</xquery>
```

For a package library module like this, the XQuery module itself *must* be stored in the */content* subdirectory of the package.

The repo.xml file

The *repo.xml* file contains additional metadata which eXist uses to determine how to install and present the package:

```
<meta xmlns = "http://exist-db.org/xquery/repo" > ❶
  description ❷
  author+
  website
  status
  license
  copyright
  type ❸
  target ❹
  prepare ❺
  finish
  permissions ❻
</meta>
```

❶ A *repo.xml* file is an XML document whose content is in the `http://exist-db.org/xquery/repo` namespace.

❷ Child elements `description`, `author`, `website`, `status`, `license`, and `copyright` contain additional (string-type) metadata about the package.

❸ The `type` childelement tells eXist what kind of package this is. It contains either the value `library` or the value `application`.

❹ The `target` child element tells eXist where to store the package in the database. This must be a relative path, and eXist prepends this with the root collection where the repository manager stores installed packages. This is, by default, */db/apps*. So, a package with `<target>xyz</target>` specified in *repo.xml* will be stored in */db/apps/xyz*.

If you want to change this root directory, you can find its definition in *$EXIST_HOME/conf.xml*: `<repository root="/db/apps"/>`.

❺ The `prepare` and `finish` child elements can contain the name of an XQuery script that runs before and after the package is installed, respectively. Further information about these scripts can be found in the next section.

Allowed values are either empty (no script), or a relative filename (relative to the root of the package). In the past, usually these scripts were called *pre-install.xql* and *post-install.xql* and stored in the root of the package, but you're free to deviate from this convention, and we would now recommend using the *.xq* file extensions instead (see "XQuery Filename Conventions" on page 13).

❻ The `permissions` child element sets for which user and under which permissions the package is loaded:

```
<permissions user = string
   password = string
   group = string
   mode = string />
```

- `user`, `password`, and `group` define the ownership of the loaded package files.

- `mode` defines the permissions on the loaded files in permission string format (e.g., `rw-rw-r--`). For XQuery files, the `x` permission will be automatically set in addition.

  In most cases you'll want your package loaded under `admin` privileges. Since the Package Manager runs as `admin` anyway, you don't have to specify the `admin` password. So, the usual contents of the `permissions` element are:

  ```
  <permissions user="admin" password="" group="dba" mode="rw-rw-r--"/>
  ```

The Prepare and Finish Scripts

In *repo.xml*, in the `prepare` and `finish` child elements, you can define two optional XQuery scripts that run directly before (preinstall) and directly after (post-install) a package is installed. Typical tasks for these scripts could include:

- Installing indexing or other definitions in a *collection.xconf* resource underneath */db/system/config*. There are two approaches you can take:
  - Do this in the preinstall script (before loading the data). The data will then be indexed on load.
  - Do it in the post-install script and programmatically reindex (using the extension function `xmldb:reindex`). This is how the package for the book's example files works. There is some generic code for this in the */db/apps/exist-book/installer/installer.xqm* module that could be reused in other packages if so desired.
- Creating users and groups for your application (or checking whether they exist).
- Creating data collections and resources outside the application's collection structure.

Both the pre- and the post-install scripts can make use of some external variables whose values will be provided by eXist during execution. The variables available for you to declare in your query's prolog are:

```
declare variable $home external;
declare variable $dir external;
declare variable $target external;
```

`$home`
: The directory where eXist is installed (i.e., *$EXIST_HOME*)

`$dir`
: The directory containing the unpacked version of your packages *.xar* file

`$target`
: The collection where your package will be installed

The pre- and post-install scripts can be difficult to debug as their output is not visible to the end user, and errors are logged only when they are very severe (which usually isn't the case). The advice here is to test them standalone (as much as possible) before you try them out as part of a package install. If you need to see what your scripts are doing, consider writing some XML to the database from your script to act as a logfile during execution, or make frequent calls to `util:log`.

Creating Packages

A *.xar* file is a ZIP file, so it's easy enough to create one manually, simply: create the right directory structure and contents on disk, and then zip it all up.

However, you'll probably create your package when developing inside/with eXist. So, the collection structure, the code, and the data will initially be *inside* eXist. To manually create a package, you'll have to export it all to disk, zip it, and so on. Not exactly impossible; just boring, repetitious work.

There is an easier solution to this, from the dashboard, start eXide. In eXide, open one of the resources in your package's home directory (e.g., */db/apps/mypackage/repo.xml)*. Choose the menu option Application→Download App, and *voilà*, you are presented with your package.

Additional Remarks About Packages

Packages must be developed in such a way that they're independent of their final location in the database. So, your application's code must be able to find out where it is (the path to itself). There are two ways to achieve this:

- You can use the extension function `system:get-module-load-path`. Unfortunately for our purposes, this function returns a collection path with the string `embedded-eXist-server` prefixed, for instance:

  ```
  xmldb:exist://embedded-eXist-server/db/apps/myapp/installer/
    installer.xqm
  ```

  To turn this into a usable collection path, you can use this regular expression code:

  ```
  replace(system:get-module-load-path(),
    '^(xmldb:exist://)?(embedded-eXist-server)?(.+)$', '$3')
  ```

  This code will work even if this strange string should disappear in a future release.

- You can also search the repository manager's root collection for the right package. Here is an example that returns the path to the book example application:

  ```
  declare namespace expath="http://expath.org/ns/pkg";

  let $descriptor := collection(repo:get-root())
    //expath:package[@name eq "http://www.exist-db.org/exist-book"]
  return
    util:collection-name($descriptor)
  ```

CHAPTER 10

Other XML Technologies

In previous chapters we have focused on eXist combined with XQuery, with only lip service paid to other XML technologies. But eXist is a full-blown XML application platform and has many other interesting and useful technologies available. One of its greatest strengths is the ability to mix and match different approaches, using the right technology for the problem at hand.

This chapter delves into technologies such as XSLT, XSL-FO, XInclude, XML validation, collations, and XForms and explains how to use them in eXist.

We do *not* explain the technologies themselves; that is to say, this chapter does not contain crash courses on XSLT, XInclude, XForms, and so on. Rather, we assume that if you need one of the aforementioned technologies, you already know how to use it (or are able to learn how elsewhere). Only the relationship with eXist is explained. If you need more information about the technologies themselves, please refer to "Additional Resources" on page 16.

A notable missing technology in this chapter is XProc. Although eXist does contain some support for using XProc pipelines, this is still rather experimental and subject to change. There is a connector to the open source XProc processor XML Calabash available; see `xmlcalabash`.

As of early 2014, there is an XProc module under development, but this will not run on eXist v2.1. You'll need to wait for v2.2 (or use the development branch from GitHub) to be able to use it.

XSLT

XQuery is a powerful language, but there are tasks that can be solved just as well with XSLT, and sometimes even more easily. For instance, for complex XML transforms you can either use XQuery (with `typeswitch` constructions) or XSLT. Whether transformations are more easily achieved in XQuery or XSLT is a contentious issue, with many experts firmly preferring one over the other. Fortunately eXist supports both, so you may decide for yourself which you find easier. The most basic examples of using XSLT in eXist can be found in "Hello XSLT" on page 33.

For executing XSLT, eXist 2.1 uses Saxon HE (Home Edition) version 9.4.0.7 by default. If you need to upgrade to Saxon's commercial PE (Professional Edition) or EE (Enterprise Edition), you can replace the existing Saxon libraries in *$EXIST_HOME/lib/endorsed* with their respective PE or EE counterparts and the accompanying license file. If you need a different XSLT processor, you can configure it in *$EXIST_HOME/conf.xml*.

An important consequence of using an external XSLT processor (and not one that is truly part of the eXist core) is that XSLT scripts run in isolation from the rest of the environment. The documents the XSLT processor works on are passed wholesale, but their database context (most importantly, indexes) is lost. No index-based query optimization is performed. So, be careful in designing the interaction between XQuery and XSLT: it's best to leave the querying to your XQuery scripts and use XSLT for transformation only.

Embedding Stylesheets or Not

Stylesheets can be fully embedded in your XQuery code. Example 10-1 shows an XQuery script that runs an embedded stylesheet for checking the XSLT system property information (and can find out whether the Saxon version changed in the eXist version you're using).

Example 10-1. Get the XSLT processor information

```
xquery version "1.0" encoding "UTF-8";

declare option exist:serialize "method=html media-type=text/html indent=no";

declare variable $page-title as xs:string := "XSLT processor information";

declare variable $xslt as document-node() := document {
    <xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns:fn="http://www.w3.org/2005/xpath-functions"
      exclude-result-prefixes="#all">
```

```
        <xsl:variable name="SystemProperties" as="xs:string+"
             select="('xsl:vendor',
             'xsl:vendor-url',
             'xsl:product-name',
             'xsl:product-version')"/>
         <xsl:template match="/">
             <XsltInfo>
                 <xsl:for-each select="$SystemProperties">
                     <Info property="">
                         <xsl:value-of select="system-property(.)"/>
                     </Info>
                 </xsl:for-each>
             </XsltInfo>
         </xsl:template>

    </xsl:stylesheet>
};

<html>
    <head>
        <meta HTTP-EQUIV="Content-Type" content="text/html; charset=UTF-8"/>
        <title>{$page-title}</title>
    </head>
    <body>
        <h1>{$page-title}</h1>
        <ul>
        {
            for $info in transform:transform(<dummy/>, $xslt, ())//Info
            return
                <li>{string($info/@property)} = {string($info)}</li>
        }
        </ul>
    </body>
</html>
```

Note the *double* curly braces in the XSLT stylesheet in `<Info property="">`. This is because we want to use the XSLT attribute-value template mechanism here, but if we use a single curly brace, XQuery kicks in and tries to interpret the contents as an XQuery expression. You can work around this by using double curly braces, which are passed as single curly braces to the XML document we're defining.

We embedded the XSLT stylesheet in our XQuery script here to show you how this works. This is useful for small scripts, but if your XSLT is longer, it is better to store it in a resource of its own and reference it (see "Invoking XSLT with the Transform Extension Module" on page 240 for details). There are also other advantages to not embedding XSLT within your XQuery:

- Most XML IDEs provide content suggestion/completion when editing XSLT stylesheets (e.g., by proposing elements or showing function declarations). If you write your stylesheet embedded in an XQuery script, the IDE most likely cannot provide such help due to the mixed-content model.
- When your code becomes sufficiently complex, you will probably want to test the stylesheet separately from the surrounding XQuery code. This is much easier when the stylesheet is a separate resource; for example, you may want to use XSpec to execute your stylesheet against a series of behavior-driven development (BBD)–style tests.
- When your stylesheet is separate, it is possible to run it through an XSLT debugger when you are trying to diagnose a problem. Such a debugger is available for Saxon in the oXygen XML Editor.
- Separate XSLT stylesheets can often have their compiled form cached, making repeated invocations faster.

Invoking XSLT with the Transform Extension Module

Performing XSLT transformations from your XQuery code can be done with eXist's `transform` extension module. For instance:

```
transform:transform(
  <input><text>hello XSLT</text></input>,
  'xmldb:exist:///db/myapp/convertinput.xslt',
  <parameters><param name="type" value="basic"/></parameters>
)
```

The first argument is the node tree to transform; the second is the URI or document element of the stylesheet. The third parameter passes the external parameter `type=basic` to the stylesheet (which you can reference in the XSLT with a global `<xsl:param name="type"/>`).

The `transform` extension module offers two approaches for doing a transformation:

`transform:stream-transform`
: Directly streams the result of the transformation to the output stream, returning the empty sequence `()`. It is most commonly used as the final transformation step for converting XML into HTML.

  The only thing you'll see in your output is the output of `transform:stream-transform`; everything else is ignored. So, this is usually the last statement in a script.

`transform:stream-transform` works only from within the REST Server; it does not work in a RESTXQ context. See "Building Applications with RESTXQ" on page 215.

`transform:transform`
: Passes the result of the transformation back to you as a node tree.

All functions have the same parameter list:

`$node-tree as node()*`
: The node tree to transform.

At present eXist relies on an external XSLT processor, so the node tree has to be serialized to a byte stream, passed to the XSLT processor, and reparsed before it can be processed. This adds some overhead to the transformation and can have an impact when you're using XSLT on very large documents from eXist.

`$stylesheet as item()`
: The stylesheet to apply. This can be either a node tree containing a valid XSLT stylesheet, or a URI referencing an XSLT stylesheet. URIs to stylesheets residing in the database *must* be specified as XMLDB URIs (i.e., start with `xmldb:exist://`).

  When you're passing stylesheets by URIs, the stylesheet is cached, speeding up performance of the invocations that follow.

`$parameters as node()`
: Optional parameters for the stylesheet, as described in the next section.

`$serialization-options as xs:string` *(optional)*
: Optional serialization options to apply to the result. Must be in the same format as the `exist:serialize` option (refer to "Serialization" on page 115).

  There is one additional serialization option: `xinclude-path`. This specifies the base path for expanding XIncludes (if any). More information about XInclude can be found in "XInclude" on page 243.

Passing XSLT Parameters

You can pass (string) parameters to your stylesheet by constructing an XML fragment as follows and passing it as the third argument to the `transform` function:

```
<parameters>
  <param name="par1" value="value of par1"/>
  <param name="par2" value="value of par2"/>
  <!-- ...any further parameters -->
</parameters>
```

You reference these in your stylesheet by specifying global parameters with the same names. For instance:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:param name="par1"/>
  <xsl:param name="par2"/>
  <xsl:template match="/">
    <p>Values passed were <xsl:value-of select="$par1"/>
       and <xsl:value-of select="$par2"/></p>
  </xsl:template>
</xsl:stylesheet>
```

There are two special parameters defined:

`exist:stop-on-error`
: If this parameter is present and set to `yes`, an XQuery error is generated if the XSLT processor reports an error.

`exist:stop-on-warning`
: If this parameter is present and set to `yes`, an XQuery error is generated if the XSLT processor reports a warning.

Most errors emitted by the XSLT processor are of the category `fatal` and will stop the processing anyway.

Invoking XSLT by Processing Instruction

Another way of invoking XSLT is by adding a `<?xml-stylesheet type="text/xsl" href="..."?>` processing instruction at the top of your output. The `href` attribute should contain a reference to an XSLT stylesheet. Relative names are taken from the location of the originating XML.

Although normal use for this processing instruction is for triggering *client-side* XSLT processing (in the browser), in an eXist context it triggers *server-side* XSLT processing. This means the client will see the output of the transformation.

There is a nice example of this in the eXist demo application. If you look in */db/apps/demo/data*, you'll find some Shakespeare plays marked up in XML. At the top of these XML files is the processing instruction `<?xml-stylesheet href="shakes.xsl" type="text/xsl"?>`. The same collection holds the referenced *shakes.xsl* document, which contains an XSLT stylesheet that renders a play in HTML. See this in action by browsing to, for instance, *http://localhost:8080/exist/rest/db/apps/demo/data/macbeth.xml*.

Checking for and invoking an XSLT stylesheet by processing instruction is enabled by default. You can, however, disable it with serialization options; see "Post-processing serialization options" on page 120.

Stylesheet Details

Here are some final details about using XSLT from within eXist:

- If you get a puzzling Java `NullPointerException` when trying to transform something using a URI for the stylesheet, it probably means that the URI was incorrect.This rather unspecific error message makes it hard to find out why the code isn't working. A common mistake here is to forget to start the URI with `xmldb:exist://` when you want to specify a stylesheet in the database!
- The serialization options set in the XSLT stylesheet itself (with `xsl:output`) will not be used.
- `xsl:include` and `xsl:import` work as expected. Relative filenames are resolved against the location of the stylesheet.
- The XPath `doc` function in a stylesheet works as expected: it loads a document. Relative paths are resolved against the location of the stylesheet (not the location of the calling XQuery file). Like in XQuery, `doc` silently returns an empty sequence, `()`, when the referenced document is not an XML file.
- The `collection` function does not behave as you would expect: it does not return any direct information about the collections in the database. It can, however, be used as Saxon intended it. For more information about this, please refer to the Saxon documentation.
- You can use the XSLT `xsl:result-document` instruction to create files on the filesystem only; writing to the database (with an XMLDB URI) results in an error. Relative paths are resolved against *$EXIST_HOME* (so `<xsl:result-document href="test.xml">` will result in a file *$EXIST_HOME/test.xml*, which is probably not where you want an output file to be written).

XInclude

By default, eXist performs XInclude processing during the serialization phase. In a nutshell, this means that `xi:include` elements are replaced with what they refer to (the namespace prefix `xi` must be bound to the namespace `http://www.w3.org/2001/XInclude`). XInclude is an official W3C standard.

XInclude is primarily intended for reusing XML or XHTML code fragments. Therefore, a use case for XInclude is an application that outputs pages with a fixed menu and navigation bar: you can insert the XHTML code for these parts using XInclude. However, in eXist more complicated scenarios are also possible, like including the output of XQuery scripts, or partial documents.

A first simple XInclude example can be found in "Hello XInclude" on page 35.

XInclude processing is switched on by default. If you want to turn it off, you can do so by using serialization options; see "Post-processing serialization options" on page 120.

eXist's implementation of the XInclude standard is not complete. Its limitations are:

- You can't use it to include raw text. Only XML is supported.
- XPointers are restricted to XPath. Additional features, like points and ranges, are not supported.

Including Documents

Including a document is easy—just add a reference to it in the `href` attribute, as in the following examples:

```
<xi:include href="includethisxml.xml"/>
<xi:include href="file:///file/from/filesystem.xml"/>
<xi:include href="http://somewhere.com/xmlfeed"/>
```

If you don't include a scheme (like `http://` or `file://`), the document is included using the same scheme as the master document (either database or filesystem). Relative paths are resolved from the location of the master document.

You can limit the output of the `include` by using an `xpointer` attribute holding a limited XPointer expression. eXist supports two constructions:

Node identifier

If you only specify the identifier of a node, the output will be limited to that node. An identifier can be set by an `xml:id` attribute or an attribute that is marked as type `ID` by an attached DTD. For instance, if we have the document:

```
<Lines>
  <Line xml:id="L1">Line 1</Line>
  <Line xml:id="L2">Line 2</Line>
</Lines>
```

Specifying the XInclude as `<xi:include href="..." xpointer="L1"/>` (assuming the `href` is correct) will only include the first `Line` element.

XPath

The other construction supported is passing an XPath expression with the so-called `xpointer` scheme. Here are some examples:

```
<xi:include href="includethisxml.xml" xpointer="xpointer(//Line[1])"/>
<xi:include href="file:///file/from/filesystem.xml"
   xpointer="xpointer(//customer[@id eq '123'])"/>
```

Including Query Results

If the `href` attribute references an XQuery script stored in the database, the script will be executed and the results included.

The executing script can get information about the master document with two variables (you don't have to explicitly declare them `as external` unless you wish to):

`$xinclude:current-doc`
: The name of master document without the collection path

`$xinclude:current-collection`
: The collection for the master document

These variables are implicitly declared by the XInclude processor. This means that if you want to use the same script outside of the XInclude context, you can't use (reference) them!

Passing your own parameters is also possible. For instance, when you have an XInclude that looks like this:

```
<xi:include href="script.xq?par1=abcdef"/>
```

you can reference the `par1` parameter in your script as `$par1` by declaring it as an external variable:

```
declare variable $par1 external;
```

Limiting the output by using XPointer, as when including documents, is not possible for XQuery results.

Error Handling and Fallback

If you try to include a resource that doesn't exist, an error will be generated. You can prevent this by specifying an `xi:fallback` element:

```
<xi:include href="includethisxml.xml">
  <xi:fallback><p>XML not found!</p></xi:fallback>
</xi:include>
```

Validation

Validation involves checking an XML document against a *grammar document,* like a DTD (document type definition) or an XML schema, and determining whether it conforms to this grammar. eXist can validate documents in two ways:

Implicit validation
: Validates an XML document when it is stored into the database. Any parsing error stops the document from being stored.

Explicit validation
: Validates documents from within XQuery code, using the `validation` extension module.

Implicit Validation

Implicit validation is (if turned on) performed when a document is being stored into the database. eXist will search for an appropriate grammar document, validate the incoming XML document against it, and reject or accept the incoming document based on the validation results. You can turn implicit validation on or off for the full database or specific collections.

Implicit validation is useful when you want to make absolutely sure that all the stored content is valid. However, some interesting limitations apply:

- Implicit validation can only be performed using XML schemas or DTDs, and not for instance with RELAX NG.
- The catalogs used for finding the grammar documents are globally defined for the entire database. There is no way to use a specific catalog for a particular collection or application.
- You cannot specify that a certain collection should only accept documents validating against a specific grammar. The only thing you can specify is that the documents must be valid according to the global set of grammar documents available in the catalog. So, a scenario forcing a collection to hold, for instance, only DocBook files is not possible with this mechanism.

This makes implicit validation a somewhat coarse-grained mechanism. However, there are use cases where it can help make your application more robust, for instance, when your database is serving a single application and all data files must validate against a specific set of XML schemas or DTDs.

Controlling implicit validation

To turn implicit validation on or off for the full database, you have to edit the *$EXIST_HOME/conf.xml* configuration file (and restart eXist afterward). Search for the following fragment:

```
<validation mode="no">
  <entity-resolver>
    <catalog uri="..."/>
  </entity-resolver>
</validation>
```

The `mode` attribute determines whether implicit validation is on or off. It can have one of the following three values:

`no`
: Implicit validation is off.

`yes`
: Implicit validation is on. All XML documents are validated and rejected if they do not pass. If an appropriate XML schema or DTD cannot be found (see "Specifying catalogs for implicit validation" on page 247), the document is rejected also.

`auto`
: Implicit validation is applied only when an appropriate XML schema or DTD can be found. Otherwise, the document is accepted.

To tune implicit validation for a specific collection (and its subcollections), you have to do the following:

1. The database has a system collection, */db/system/config*. Repeat the database collection structure here, leading up to the collection for which you want to specify the implicit validation. So, when you want to turn on implicit validation for */db/myapp/data*, create the collection */db/system/config/db/myapp/data*.
2. Create an XML file here called *collection.xconf* with the following contents:

   ```
   <collection xmlns="http://exist-db.org/collection-config/1.0">
     <validation mode="..."/>
   </collection>
   ```

3. Fill in the appropriate value for the `mode` attribute, as described earlier.

Specifying catalogs for implicit validation

The `validation` element in the *$EXIST_HOME/conf.xml* file also contains the URIs of the *catalog files* that eXist uses for implicitly validating documents. A catalog file specifies:

- Mappings from system or public IDs to DTD grammar documents
- Mappings from namespaces to XML schema grammar documents

eXist works with v1.0 OASIS catalog files. For an example, have a look at the default eXist catalog file in *$EXIST_HOME/webapp/WEB-INF/catalog.xml*.

Catalog files must be specified in the *$EXIST_HOME/conf.xml* file as in this example:

```
<validation mode="no">
  <entity-resolver>
    <catalog uri="${WEBAPP_HOME}/WEB-INF/catalog.xml"/>
    <catalog uri="xmldb:exist:///db/myapp/myapp-catalog.xml"/>
  </entity-resolver>
</validation>
```

All `uri` attributes must point to valid OASIS catalog files:

- By default, the catalogs are on the filesystem. Use an XMLDB URI (like in the second `catalog` element in the preceding example) to specify a catalog that is stored in the database.
- You can use `${EXIST_HOME}` to point to *$EXIST_HOME* and `${WEBAPP_HOME}` to point to *$EXIST_HOME/webapp*.

Explicit Validation

Explicit validation allows you to perform validation from your code using the `validation` extension module. eXist provides three different parsers for this:

JAXP
: This parser (called JAXP because internally it uses the Java `javax.xml.parsers` interface) validates documents using Xerces2. XML schemas (v1.0) and DTDs are supported through Xerces. Implicit validation uses the JAXP validator.

JAXV
: This parser (called JAXV because internally it uses the Java `java.xml.validation` interface) validates documents using the validation facility built into the Java standard library. Only XML schemas are supported.

Jing
: This validation is based upon James Clark's Jing parser. It supports XML Schema, RELAX NG (both full and compact), Schematron (v1.5), and Namespace-based Validation Dispatching Language (NVDL).

This leaves you with the problem of which one to use. If you are unsure, the general advice is to use JAXP for XML schema– and DTD-based validation and Jing for all other types.

Performing explicit validation

You can perform explicit validation using the functions from the `validate` extension module. This module has separate functions for all three parsers (JAXP, JAXV, and Jing).

There are also generic validation functions that try to select the best parser for you, based on the type of grammar document that you provide. We'll describe these generic functions here; the functions for the specific parsing types are more or less the same. For specifics, please refer to the online XQuery function documentation.

There are two generic validation functions:

`validation:validate`
: This will validate your input document and return `true` or `false` depending on the result.

`validation:validate-report`
: This will validate your input document and return an XML fragment describing the result. Use this if, for instance, you have to provide detailed feedback to the end user about the validity of some input.

These functions will choose a parser based on the type of grammar document: if this is a DTD or XML schema, the JAXP (Xerces) parser is used, otherwise Jing is used.

These functions do not produce the correct results for XML where the root element is not in a namespace. In that case, use the specific functions instead (e.g., `validation:jing-report`).

Both functions have the following arguments:

`$instance as item()`
: This is the input document to validate. You can specify either a URI (data type `xs:anyURI`), an element, or a document node.

`$grammar as xs:anyURI`
: This argument is optional. It is used to determine the grammar document. Specifying the grammar in `$grammar` can be done in one of four ways:

  - If you don't specify `$grammar`, the catalogs defined for implicit validation (see "Specifying catalogs for implicit validation" on page 247) are used.
  - If the `$grammar` URI ends with *.dtd* (DTD), *.xsd* (XML Schema), *.rng* (RELAX NG), *.rnc* (RELAX NG Compact), *.sch* (Schematron), or *.nvdl* (NVDL), it is assumed to reference a grammar document of that type.

- If the `$grammar` URI ends with *.xml* it is assumed to be an OASIS catalog file, which is used to further determine the grammar document.
- If the `$grammar` URI ends with a /, it is assumed to be the name of a collection. eXist will search for an appropriate grammar in that collection and its subcollections.

The report returned by `validation:validate-report` for a valid document will look like this:

```
<report>
  <status>valid</status>
  <namespace>http://myapp.com/namespace</namespace>
  <duration unit="msec">51</duration>
</report>
```

For an invalid document, it will contain one or more error messages. For example:

```
<report>
  <status>invalid</status>
  <namespace>http://myapp.com/namespace</namespace>
  <duration unit="msec">6</duration>
  <message level="Error" line="10" column="29">
     cvc-complex-type.2.4.a: Invalid content was found ...
  </message>
</report>
```

Grammar management in the JAXP (Xerces) parser

This applies to the JAXP (Xerces) parser type only: to speed up validation, grammar documents are loaded, compiled, and held in the cache. This is usually fine, but there might be situations where you want a little bit more control over this (for instance, when you're developing grammars).

eXist provides the following XQuery functions for working with the grammar cache:

`validation:clear-grammar-cache`
: Clears the grammar cache and returns the number of deleted grammars

`validation:pre-parse-grammar`
: Parses one or more XML schema documents or DTDs and adds them to the grammar cache

`validation:show-grammar-cache`
: Returns an XML fragment describing the contents of the grammar cache

Collations

Collations are the mechanism used for comparing strings. By specifying a collation, you can make comparing strings language-specific (a.k.a. locale-specific). For instance, in a default comparison of `'në'` with `'ni'`, `'ni'` comes first, because the Unicode code for an `ë` is greater than the Unicode code for an `i`. However, if you compare these words with a collation for a language that uses diacriticals (like Dutch, German, or French), things get reversed because the `ë` is treated like an `e`.

Supported Collations

eXist supports the following collations:

`http://www.w3.org/2005/xpath-functions/collation/codepoint`
: This is the default collation that uses the Unicode code points. Internally, the basic Java string comparison and search functions are used.

`http://exist-db.org/collation?lang=...&strength=...&decomposition=...`
: Or for short: `?lang=...&strength=...&decomposition=...` (the `strength` and `decomposition` parameters are optional). This specifies a language-specific collation:

  - The `lang` parameter selects the language using an ISO 639-1 language code like `en`, `en-US`, `de`, `nl-NL`, or `fr`.

    You can find out which languages are supported by calling the `util:collations` extension function.
  - The `strength` parameter value must be one of `primary`, `secondary`, `tertiary`, or `identical`.
  - The `decomposition` parameter value must be one of `none`, `full`, or `standard`.

  What exactly these parameters do is a deep and rather separate subject that we're not going to handle here. It has to do with the way Unicode is built up, and canonization of Unicode accented characters. Most likely, if you don't know what this is about, you probably don't need to. A good place to start looking for more information is the Unicode site.

Specifying Collations

There are several ways to work with collations:

- You can specify a default collation for your XQuery script in its prolog:

```
declare default collation "?lang=de-DE";
```

- The FLWOR expression's `order by` clause has a `collation` keyword for specifying the collation:

```
for $w in $list-of-words
order by $w collation "http://exist-db.org/collation?lang=nl-NL"
return
  $w
```

- Several functions have collation arguments—for instance, `contains` and `ends-with`.

Lots of standard string functions, like `contains` and `ends-with`, accept an optional third collation parameter. Although you can certainly use this functionality, it may stop the expression from being optimized and indexes from being exploited!

XSL-FO

XSL Formatting Objects (XSL-FO) is an XML vocabulary to transform XML into formatted media, often PDF. To turn XSL-FO XML into PDF you need an XSL-FO formatter (or renderer), such as the open source Apache FOP or a commercial one like Antenna House Formatter or RenderX XEP. eXist has the ability to connect directly with several XSL-FO renderers. Our examples will use the open source Apache FOP formatter, but they should work for any formatter supported by eXist.

eXist has standard connectors for the Apache FOP, RenderX, and Antenna House formatters. You can change formatters by placing the JAR files in *$EXIST_HOME/lib/user* and changing the processor adapter within the definition of the `XSLFOModule` module configuration in *$EXIST_HOME/conf.xml*.

It is also possible to add support for any third-party FO processor to eXist by writing a simple SAX adapter in Java that implements `org.exist.xquery.modules.xslfo.ProcessorAdapter` and making it available on the classpath (e.g., adding it to *$EXIST_HOME/lib/user*).

Usually, an (XML) application that wants to present something in PDF creates, from some data source, the XSL-FO XML. This can be done by XSLT, XQuery, or any other way you like. The resulting XSL-FO document is then passed to the XSL-FO formatter for final processing. If your XSL-FO document doesn't contain any errors, a PDF is produced.

Performing the final XSL-FO transformation is trivial. We assume here that the XSL-FO document creation is already complete and that the final XSL-FO document is available from somewhere in the database. The following example transforms this document to PDF and displays it in your browser:

```
let $xsl-fo-document as document-node() := doc('some-xsl-fo.xml')
let $media-type as xs:string := 'application/pdf'
return
  response:stream-binary(
    xslfo:render($xsl-fo-document, $media-type, ()),
    $media-type,
    'output.pdf'
)
```

The `xslfo:render` function does the trick: it transforms the XSL-FO instructions into a (binary, `xs:base64binary`) PDF document and returns this. This is picked up by `response:stream-binary`, which sends it to your browser. Because the Internet media type is set to `application/pdf`, it will (hopefully) show up as a nicely formatted PDF document. There is an example in the accompanying source code (see "Accompanying Source Code" on page 15) that does exactly this.

The parameters for the `xslfo:render` function are:

`$document as node()`
: The XSL-FO document to render.

`$mime-type as xs:string`
: The requested output's Internet media type. In most cases this will be `application/pdf`. Please refer to your XSL-FO formatter's documentation to find out if other Internet media types are supported also.

`$parameters as node()`
: Parameters to pass to the formatter. The format, a `parameter` element with `param` children, is exactly the same as that used for passing parameters to an XSLT transformation (see "Passing XSLT Parameters" on page 241). Recognized rendering parameters are `author`, `title`, `keywords`, and `dpi`.

`$config-file as node()` *(optional)*
: An optional formatter configuration file. Please refer to your XSL-FO formatter's documentation for more information about this.

XForms

XML Forms (XForms) is an XML standard developed by the W3C to provide the next generation of forms for the Web: it splits apart the data model from the presentation of that data model, so that you may focus on each independently. XForms are a major component of XRX web application architectures.

XForms are not freestanding expressions; rather, they must be embedded into a *host document*. Originally, XForms were expected to become the forms for XHTML 2.0. While the XHTML 2.0 Working Group has expired and XHTML 2.0 has been superseded by HTML5 and its XHTML expression, the XForms standard is still being actively developed and of course may be embedded into HTML5 documents as an alternative or complement to HTML forms.

You may be wondering when you would consider using XForms instead of HTML forms. Our advice would be to use XForms when you need to collect anything more than a couple of trivial fields. XForms can provide automated validation and correction hinting of form values and enable you to collect your form responses into a complex, structured XML document that can be saved directly into eXist and/or further processed with XQuery or XSLT. One of the great advantages of XForms is that the same form that is used for collecting data can later be used to edit that data (when provided with the collected XML document as its instance). Another advantage is that if your forms need to perform calculations, either for display or within an instance for submission (such as calculating totals in a spreadsheet), XForms provides clever dependency rules that enable these values to be automatically recalculated when a dependency in the graph changes.

There are two main classes of XForms processors:

Server-side processors
: A server-side XForms processor renders XForms markup into another form (such as HTML, CSS, or JavaScript) on demand, when a request for a form is made to the server. The main advantage of the server-side approach is that you only transmit a rendered representation of the XForm to the client; you don't need to transmit all of the data of the model behind the form. Server-side processors often blur the strict client/server boundary, as the JavaScript (or other code) that some of them generate on the server in fact often runs on the client.

Client-side processors
: A client-side XForms processor instead typically operates inside a web browser, as either a plug-in or a JavaScript library. When the browser receives XForms markup, the XForms processor modifies the page DOM to enable the browser to render the XForm as intended and handle interactions and events. The main advantages of the client-side approach are that you do not require any special server-side processing, and you distribute the processing of forms to the client.

This chapter is not a comprehensive explanation of XForms itself, but instead is meant to show how you can use XForms with eXist. For in-depth information about XForms, check out Micah Dubinko's book *XForms Essentials* (O'Reilly) and Dan McCreary's XForms wikibook.

eXist provides facilities for both server-side processing through betterForm, which is embedded in eXist, and client-side processing through XSLTForms, which is available as an EXPath package for eXist. We will take a look at how each of these may be configured and used shortly.

At this point it is also worth mentioning the excellent Orbeon Forms. Orbeon is an open source (LGPL v2.1–licensed) server-side XForms processor that ships with an embedded eXist instance. There is also a commercial and supported version available. One of the major features of Orbeon is that it provides a pipeline language called XPL that enables you to easily create XForms, deliver them over the Web, and then save the results into eXist. It is also possible to configure Orbeon to use a separate eXist server instead of its own embedded instance. Orbeon is a separate project that deserves a book in its own right, so it will not be discussed further here; however, if you are interested in XForms and eXist, it is well worth evaluating.

XForms Instances

An XForm may have one or more *instances* within its *model*; these instances define the model aspect of the MVC architecture behind XForms. Simply put, each instance can be considered a standalone XML document that provides data to the form, for the purposes of display, capture, or influencing behavior. Ultimately, it is usually an instance (XML document) that is stored into eXist when the user submits the form. Typically, in a simple XForm the instances are hardcoded as either documents that have structure but no content, or documents with content that is to be edited; however, with eXist, you have several ways to make the instance data available to your form in a more dynamic manner.

Instances and the REST Server

Each instance within the model of an XForm need not be inlined. Rather, an instance can be retrieved from an external URI—and what better place to retrieve your XML instance documents from than a native XML database like eXist?

So, rather than constructing your instance inline like so:

```
<xf:model>
    <xf:instance xmlns="">
        <company>
            <name>eXist Solutions GmbH</name>
            <registration>HRB 89454, Amtsgericht Darmstadt</registration>
            <vatId>DE273180763</vatId>
            <taxNum>007 232 51397 </taxNum>
        </company>
      </xf:instance>

      ...

    </xf:model>
```

you could instead store your instance document into eXist and construct your instance like so:

```
    <xf:model>
      <xf:instance xmlns=""
       src="http://localhost:8080/exist/rest/db/companies/exist-solutions.xml"/>

      ...

    </xf:model>
```

While the end result is the same, there are several advantages to be gained from using the latter approach:

Content reuse
: Your instance data can be reused in different applications, and may not necessarily be exclusive to your XForm.

Security
: eXist provides an extensive security system and offers authorization and authentication for resources in the database. Therefore, you can separately manage the security constraints of your data and your forms, which may have different requirements.

Architecture
: While referencing the URL of the instance still provides a static instance, it is a pattern that we can reuse to provide a dynamic instance instead.

Instances and XQuery

We have seen how you may request an instance from eXist's REST Server with XForms rather than inlining the instance content, but up to this point the instances have been static. Here we look at generating an instance dynamically using XQuery.

Imagine that in the database we have a collection of XML documents (*/db/weather*), one for each day, that describes the weather for that day. In our form we may wish to display some information about today's weather. By sending a small piece of XQuery to the REST Server as part of an HTTP `GET` request, we can retrieve the correct weather document for our instance. Such a request to the REST Server may look like:

```
    http://localhost:8080/exist/rest/db/weather?_query=
      /weather[@date eq current-date()]
```

An XForms instance declaration to retrieve this instance would then look like Example 10-2.

Example 10-2. Instance retrieval by query submission

```
<xf:instance xmlns=""
  src="http://localhost:8080/exist/rest/db/weather?_query=%2Fweather%5B%40date%20eq
    %20current-date()%5D&_wrap=no"/>
```

In the URL used in the `xf:instance/@src` we need to *URL-encode* the query string parameter for the XQuery used to ensure that it is correctly transmitted. We have also added the parameter `wrap=no`, as we want the matched XML document for our instance; otherwise, by default it would have been wrapped in an `exist:result` element by the REST Server!

See "Querying the database" on page 326 for further information on submitting an XQuery to the REST Server.

Our next example is a reworking of Example 10-2, but rather than sending the XQuery to the database, it instead relies on the fact that we have already stored the XQuery into the database. Doing so allows us to later invoke the query from the REST Server by URI and have it executed.

So, if you were to store the following XQuery into the database at */db/weather.xq*:

```
xquery version "1.0";
collection("/db/weather")/weather[@date eq current-date()]
```

An XForms instance declaration to retrieve this instance would then look like Example 10-3:

Example 10-3. Instance retrieval by stored query

```
<xf:instance xmlns=""
  src="http://localhost:8080/exist/rest/db/weather.xq"/>
```

You have now seen how you can bring in instance data dynamically, but this is really just scratching the surface of what is possible. You can also send parameters to your stored XQuery to influence the XML it will produce for your instance. For further information, see "Executing stored queries" on page 335 and "The request Extension Module" on page 209. As an alternative to stored query execution via the REST Server, you could retrieve an instance from a URI provided by a RESTXQ resource function; for further details, see "RESTXQ" on page 353.

It is also possible to dynamically calculate the URI from which to retrieve instance data. Unfortunately, this cannot be done through the `xf:instance` directly, as XPath expressions are not allowed in the `src` attribute. However, this is possible through clever use of an `xf:submission` and event handling to replace instance content, as described at *https://en.wikibooks.org/wiki/XForms/Read_and_write_with_get_and_put#Discussion*.

XForms Submissions

Typically, you will want to store the completed result of your XForm somewhere, either for posterity or for further processing. The responsibility of an `xf:submission` is typically to submit an *instance* from the *model* using a *method* to a *resource*. Almost all XForms implementations support submission by HTTP `GET`, `POST`, and `PUT`, which is a great fit for use with eXist's REST Server or RESTXQ APIs. Given that, you can easily have the result of your completed XForm stored into the database.

Submission to the REST Server

It is relatively trivial to design your XForm to store its result into an XML document in the eXist database, by simply modifying its `xf:submission` to HTTP `PUT` the instance into an XML document in a collection within the eXist database.

For example, the `xf:submission` shown in Example 10-4, when fired, would place the result of the XForm into the document */db/registration/result.xml* within eXist.

Example 10-4. XForms submission to REST Server

```
<xf:submission id="s-save" method="put"
  resource="http://localhost:8080/exist/rest/db/registration/result.xml"
  replace="none">
    <xf:action ev:event="xforms-submit-error">
        <xf:message>Registration failed. Please fill in valid values.</xf:message>
    </xf:action>
    <xf:action ev:event="xforms-submit-done">
        <xf:message>You have been registered successfully.</xf:message>
    </xf:action>
</xf:submission>
```

Some advantages of this approach are:

Automatic collection creation
: As we are doing an HTTP `PUT` to the REST Server, eXist will create any collections that do not yet exist but are required to store the document.

Create or update
: If a document with the same URI does not yet exist in the database, it will be created. However, if a document with the same URI is already present, it will be overwritten with the new instance content.

The major disadvantage of this approach is that we can only create a single document in the database, when it's likely we'll want many users to fill out our form and the results to be stored into the database and/or further processed. Solving this will be discussed next.

Submission via XQuery

You have seen how you may store the result of an XForm directly into eXist via the REST Server without having to know anything more than XForms. However, this approach is quite limited, so we will now look at submission via XQuery to dynamically store and/or post-process the instance content.

By submitting the instance content to a stored XQuery via the REST Server, we have the full power of XQuery at our fingertips to help us decide how to then store the document into the database. Of course, we may also do some post-processing and assert some control over the result of the XForms submission by having the XQuery return an appropriate HTTP response to the submission. Now we will look at storing each instance submission into its own document in the database collection */db/registration*.

Say you create the collection */db/registration*, and then store the XQuery shown in Example 10-5 into the database at */db/registration.xq*.

Example 10-5. Submission via stored query

```
xquery version "1.0";

import module namespace request = "http://exist-db.org/xquery/request";
import module namespace xmldb = "http://exist-db.org/xquery/xmldb";

let $doc-db-uri := xmldb:store ❶
  ("/db/registration", ❷
  (), ❸
  request:get-data() ❹)
return
    <stored> ❺
        <dbUri>{$doc-db-uri}</dbUri>
        <uri>http://{request:get-server-name()}:{request:get-server-port()}
          {request:get-context-path()}/rest{$doc-db-uri}</uri>
    </stored>
```

❶ The XQuery function `xmldb:store` will store a document into a database collection in eXist.

❷ We specify the collection */db/registration* as the first argument to `xmldb:store`, which is the collection in which to store the document.

❸ Note that the second argument to `xmldb:store` is the *empty sequence*; this tells eXist that we do not know the name of the document we wish to store. eXist will create a name for the document on our behalf; a random-number generator is used and the result is encoded into a hexadecimal string for use as the filename.

❹ The function `request:get-data` will retrieve the body of an incoming HTTP `POST` or `PUT` request; in this case, it will be the instance content from the XForm.

❺ We return to the XForm submission a simple XML document, which, although we do not act on it in our form here, could be used for instructing the XForm further.

An XForms *submission declaration* to submit the instance to this XQuery would then look like:

```
<xf:submission id="s-save" method="post"
  resource="http://localhost:8080/exist/rest/db/registration.xq"
  replace="none">
    <xf:action ev:event="xforms-submit-error">
        <xf:message>
          Registration failed. Please fill in valid values
        </xf:message>
    </xf:action>
    <xf:action ev:event="xforms-submit-done">
        <xf:message>You have been registered successfully.</xf:message>
    </xf:action>
</xf:submission>
```

We use the `POST` method here for our `xf:submission` instead of the `PUT` method used in Example 10-4 as the REST Server in eXist does not support stored query execution via HTTP `PUT`.

The disadvantage of the approach in Example 10-5 is that if the same user completes the form twice or hits Submit twice, you will end up with two XML documents in the database collection containing (most likely) the same instance content. There are many different ways to attack this problem, but they all involve being able to identify the user. Two possible approaches are as follows:

- Have the user add a *uniquely identifiable* piece of information to a form field. When the form is submitted, you can check if this information from the instance is already present in the database; if so, you can fail the submission by returning an HTTP `403 Forbidden` error code (e.g., `response:set-set-status-code(403)`).
- Set up authentication and appropriate permissions, and have the user log in via XQuery before allowing her to access the XForm. In this way a session will be created on the server. You can use the username to determine if the user has already created a document in the */db/registration* collection. If she already has a submission in the collection, simply return a `403 Forbidden` response to the form submission.

  In combination with this approach, you could generate the XForm using XQuery (as simply as calling `doc`). If the user already has a submission in the collection, instead of showing her the form, you can display a message or redirect the user to another page.

For further details on calling stored queries via the REST Server, see "Executing stored queries" on page 335. As an alternative to stored query execution via the REST Server, you could instead submit an XForm instance to a URI provided by a RESTXQ resource function, as discussed in "RESTXQ" on page 353.

Submission authentication

So far, all of our submission examples that store or update documents in eXist have ignored the issue of security (or assume that you have manually authenticated). Unfortunately, support in XForms 1.1 for authentication is terribly lacking. You should really be able to do basic HTTP authentication at the very least, but there is no function in XForms to Base64-encode your authentication credentials. There is function support in XForms for creating digests, so even better, you would hope that you could perform HTTP digest authentication. Alas, there is no way to handle the challenge from the server that provides the *nonce* that you need to reuse as part of your digest!

At present there is only one mechanism in XForms that is not eXist-specific and can be reliably used to authenticate with eXist (see Example 10-6). That mechanism involves your passing your username and password in *clear text* as part of the *submission resource* URI. Obviously, sending this information in clear text is not at all ideal! If you are using the betterForm processor because the processing happens on the server side, this information will never leave your server. However, it is still not ideal, so for a betterForm-specific solution, see the next section. If you are using the XSLTForms processor, this information will be sent in clear text, but there is an alternative option covered in "XSLTForms" on page 265.

Example 10-6. Statically coded authentication

```
<xf:submission id="s-save" method="put"
  resource="http://username:password@localhost:8080/exist/rest/db/registration
  /result.xml" replace="none"/>
```

Perhaps slightly better is that through the use of an `xf:resource` element in the `xf:submission`, you could dynamically encode the username and password into the URI from form fields that the user has completed and that are present in an instance. See Example 10-7.

Example 10-7. Constructed authentication from form

```
<xf:submission id="s-save" method="put" replace="none">
  <xf:resource
    value="concat('http://', instance('auth')/Username,
    ':', instance('auth')/Password,
    '@localhost:8080/exist/rest/db/registration/result.xml')"/>
</xf:submission>
```

betterForm

betterForm is an open source (BSD and Apache2 licensed) server-side XForms 1.1 processor written in Java. To say that betterForm only runs on the server side would be unfair; the majority of the processing happens server-side, but the server also generates JavaScript and HTML5 (or XHTML) for the web browser to represent your XForm UI. All UI interaction in the browser and subsequent incremental updating is processed by JavaScript with Ajax calls back to the server.

betterForm comes already bundled with eXist, and thus there is no installation or configuration required to start immediately working with XForms using betterForm and eXist. It really could not be simpler!

How does this work, you ask? The simple explanation is that betterForm acts as a filter between eXist and all HTTP traffic. If betterForm detects that eXist is returning an XML document that contains an XForm, it will transparently intercept this and replace the XForm with an HTML form, CSS, and JavaScript. Likewise, when the form UI is interacted with, or instances need to be submitted or updated, betterForm intercepts these requests to eXist and takes care of processing the state of the XForm before passing the request on to eXist.

By default eXist and betterForm are configured such that any documents stored into the database that are delivered over the URI */exist/apps* can be intercepted and processed by betterForm. Remember that the URI */exist/apps* is mapped onto the collection */db/apps* by the XQuery URL rewriting controller (see "The controller-config.xml Configuration File" on page 206). Therefore, any documents containing XForms stored into the database collection */db/apps* (or a subcollection of it) and requested by a URI starting with */exist/apps* will be processed by betterForm.

Additional Tips for Working with betterForm

Here are some hints and tips for working with betterForm effectively:

- You can change the URI path that betterForm post-processes by adjusting the `XFormsFilter` `url-pattern` in *$EXIST_HOME/webapp/WEB-INF/web.xml*, after which you must restart eXist for the change to take effect. For example:

```
<filter-mapping>
  <filter-name>XFormsFilter</filter-name>
  <url-pattern>/apps/*</url-pattern>
</filter-mapping>
```

- Should you wish to entirely disable betterForm post-processing, you may do so by changing `filter.ignoreResponseBody` to `true` in *$EXIST_HOME/webapp/WEB-INF/betterform-config.xml*, after which you must restart eXist for the change to take effect. For example:

```
<property name="filter.ignoreResponseBody" value="false"/>
```

- If you wish to see what your XForm is doing within betterForm, you can enable the betterForm debugger. This will add an additional toolbar to your rendered XForms page, allowing you to introspect the host document, instances, and events. To enable the debugger, set `betterform.debug-allowed` to `true` in *$EXIST_HOME/webapp/WEB-INF/betterform-config.xml*, after which you must restart eXist for the change to take effect. For example:

  ```
  <property name="betterform.debug-allowed"
    value="true"
    description="if true enables debug bar and event log viewer"/>
  ```

  In addition, if you wish to monitor betterForm on the server and how it processes your XForms host documents, you can find its logfile in *$EXIST_HOME/webapp/WEB-INF/logs/betterform.log*.

- Since version 2.1 of eXist, betterForm has provided an *eXist connector*. This connector may be used when you are running betterForm embedded in eXist. The connector enables betterForm to participate in the current user session of eXist. Instead of using `http://` in the scheme of your URIs for talking to eXist, you can instead use `exist://` and remove the hostname, port, and context from the URIs. For example, if you previously used `http://localhost:8080/exist/rest/db/registration.xq` as the URI for the `src` of your instance, you could now instead use `exist://db/registration.xq`.

There are many excellent betterForm examples at *http://demo.betterform.de/exist/apps/betterform/dashboard.html*. In particular, the *Feature Explorer* is useful for anyone learning XForms, regardless of whether you are using betterForm.

The source code of a small XForm for capturing details of someone using betterForm is provided at *chapters/other-xml-technologies/better-form/test-xform.xhtml* in the *book-code* Git repository (see "Getting the Source Code" on page 15).

To use the example, simply store the *better-form/test-xform.xhtml* document into the */db/apps* collection in eXist. You can then display the form in a web browser by calling the document from eXist's REST Server using a URI like *http://localhost:8080/exist/apps/test-xform.xhtml*. If all goes well, you should see the result of the XForm (after being processed by betterForm) rendered in your web browser (see Figure 10-1).

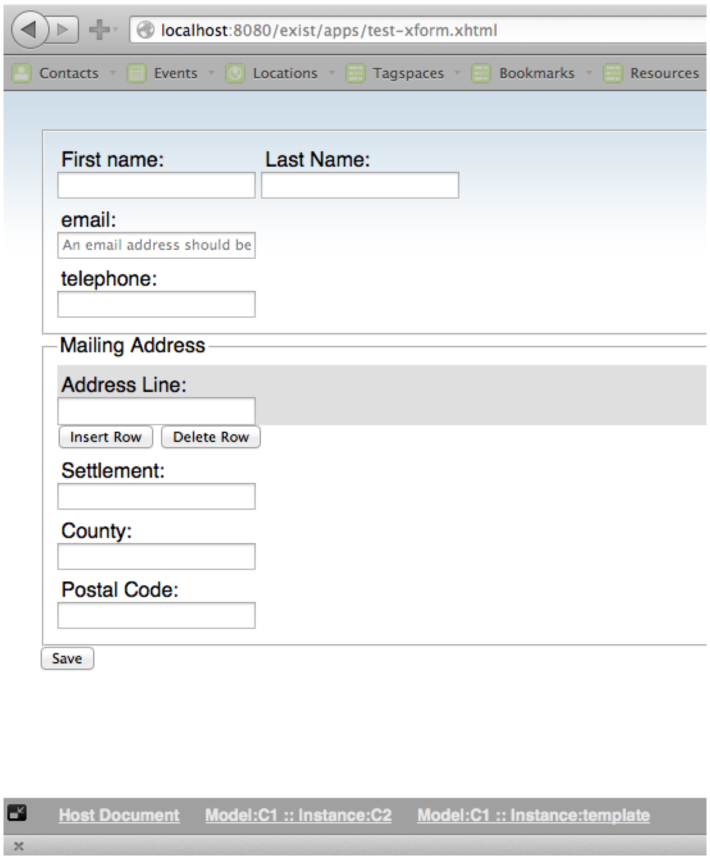


Figure 10-1. Address XForm with betterForm (debug mode)

XSLTForms

XSLTForms is an open source (LGPL v2.1–licensed) client-side XForms 1.1 processor written in XSLT 1.0 and JavaScript. XSLTForms relies on your web browser to execute the XSLT and JavaScript code that will translate your XForm XML into HTML that the browser can render and handle interactions and events for.

While XSLTForms is described as a client-side processor and this is most often how it is deployed, it is entirely possible to process the XSLT part of XSLTForms server-side with eXist. You can do so either by using an XSLT transformation function from XQuery (see "XSLT" on page 238), or by using URL rewriting (see *http://www.exist-db.org/exist/apps/doc/xforms.xml#D1.2.5.3*). As these topics are covered elsewhere, they will not be discussed further here.

To install XSLTForms with eXist, you must install the EXPath package provided for use with eXist via the eXist dashboard. Visit *http://localhost:8080/exist/apps/dashboard/* on your machine, and then install XSLTForms by clicking the Package Manager app and selecting Install for the XSLTForms Files package (see Figure 10-2).

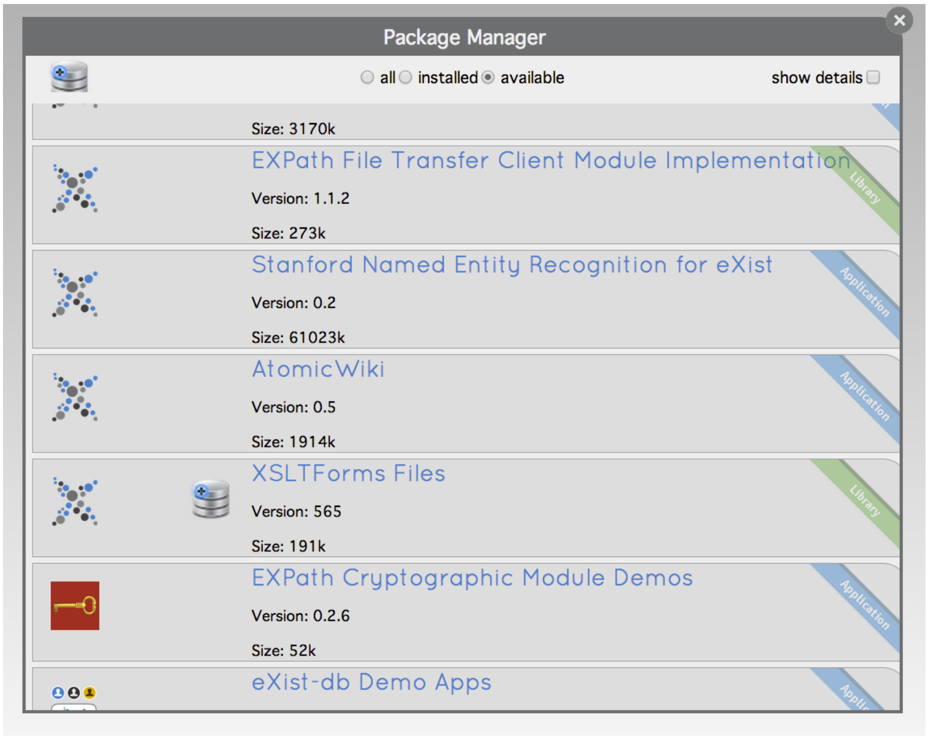
Installing the XSLTForms Files package will download and extract the XSLTForms EXPath package into a new database collection at */db/apps/xsltforms*.

Once the XSLTForms Files package is installed into eXist, you may simply store your XForm host document into the database (typically, if it is an HTML document, you should use the *.xhtml* extension to ensure it is delivered to the web browser with an `application/xml Content-Type`) and add the following processing instruction to the top of it:

```
<?xml-stylesheet
  href="http://localhost:8080/exist/apps/xsltforms/xsltforms.xsl"
  type="text/xsl"?>
```

As opposed to the absolute URI used in the preceding processing instruction, you may use a relative URI. For example, if you had stored your XForm into a subcollection of */db*, you might use the following processing instruction:

```
<?xml-stylesheet href="../apps/xsltforms/xsltforms.xsl" type="text/xsl"?>
```


Package Manager

all installed [✓ available] show details

Size: 3170k

EXPath File Transfer Client Module Implementation
Version: 1.1.2
Size: 273k

Stanford Named Entity Recognition for eXist
Version: 0.2
Size: 61023k

AtomicWiki
Version: 0.5
Size: 1914k

XSLTForms Files
Version: 565
Size: 191k

EXPath Cryptographic Module Demos
Version: 0.2.6
Size: 52k

eXist-db Demo Apps

Figure 10-2. Package Manager: installing the XSLTForms Files package

Additional Tips for Working with XSLTForms

Here are some hints and tips for working with XSLTForms effectively:

- By default, when directly requesting an XForms host document from the REST Server, eXist will attempt to process XML stylesheet processing instructions server-side. You should be able to disable this in *$EXIST_HOME/conf.xml* by changing the `enable-xsl` attribute on the `serializer` config element (in fact, the default in eXist is to disable this), but the setting does not seem to *currently* affect behavior in eXist 2.1. Alternatively, you may manually disable server-side processing on a per-request basis via the REST Server by adding `?_xsl=no` to the URI of your XForm.

  Likewise, should you wish to generate your XForm using an XQuery, you need to instruct the serializer not to expand the processing instruction when it serializes your XQuery by adding the following option to the prolog of the XQuery:

  ```
  declare option exist:serialize
    "method=xhtml media-type=text/xml process-xsl-pi=no";
  ```

- eXist is configured such that betterForm will automatically try to render any XForm content serialized from eXist that was accessed via the */exist/apps* URI of the REST Server. To disable this behavior, see "betterForm" on page 263.
- If you wish to see what your XForm is doing in XSLTForms, you can add this processing instruction below the `xml-stylesheet` processing instruction in your XForm host document:

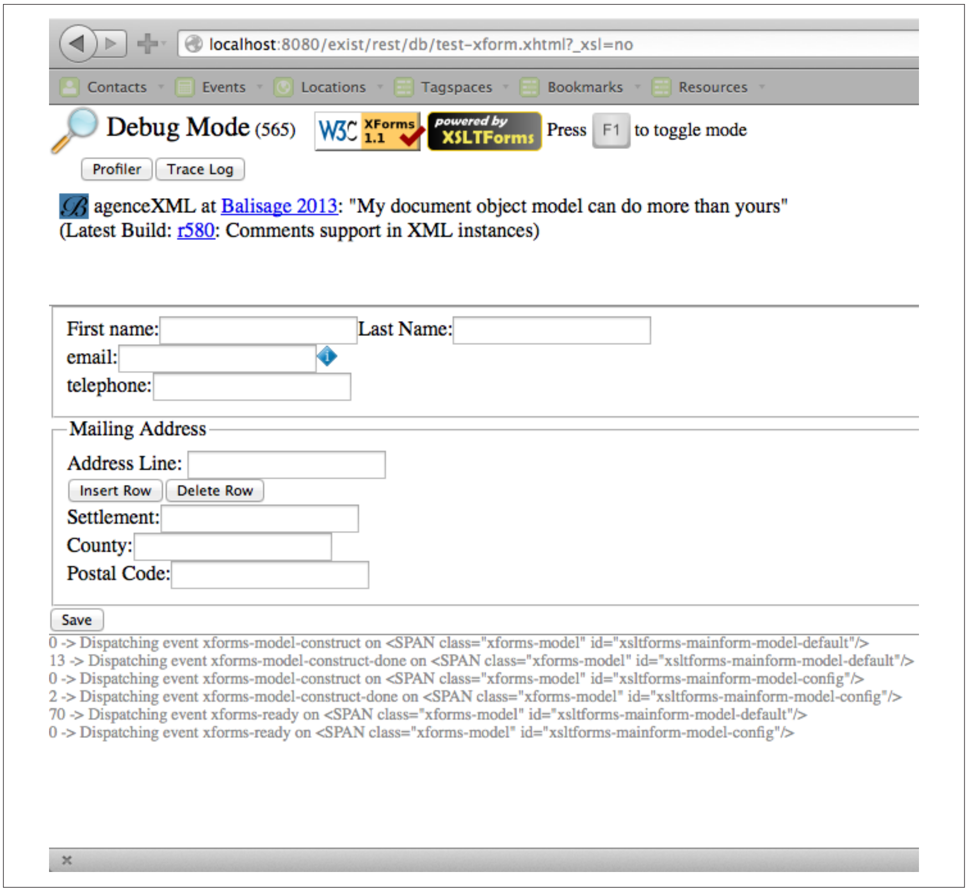
  ```
  <?xsltforms-options debug="yes"?>
  ```

  This will enable the XSLTForms debugger, after which launching the Profiler can be a useful tool for viewing the current state of your model instances (via the built-in Instance Viewer).
- eXist includes an authentication mechanism that is not specific to XForms but will work for XSLTForms served from eXist. The approach is for you to create an XQuery that logs users into eXist using `xmldb:login`. A user must visit this XQuery before being served the XForm host document from eXist. In this way the browser is furnished with an HTTP cookie representing the current user's logged-in session, and any submissions subsequently performed by an XForm within the same session will have the same access rights to the database as that user would. With this mechanism, there is no need to encode a username and password into the resource URI of the `xf:submission`!

Once you have XSLTForms installed and configured, you can simply store your XForms host documents into the database (typically as XHTML documents) and request them via the REST Server.

The source code of a small XForm for capturing details of someone using XSLTForms is provided at *chapters/other-xml-technologies/xslt-forms/test-xform.xhtml* in the *book-code* Git repository (see "Getting the Source Code" on page 15).

To use the example, simply install and configure XSLTForms as described previously and then store the *xslt-forms/test-xform.xhtml* document into the */db* collection. You can then display the form in a web browser by calling the document from eXist's REST Server using a URI like *http://localhost:8080/exist/rest/db/test-xform.xhtml?_xsl=no*. If all goes well, you should see the result of the XForm (after being processed by XSLTForms) rendered in your web browser with the XSLTForms debugger enabled, as shown in Figure 10-3.



agenceXML at Balisage 2013: "My document object model can do more than yours"
(Latest Build: r580: Comments support in XML instances)

First name:	
Last Name:	
email:	
telephone:	

Mailing Address

Address Line:	
Settlement:	
County:	
Postal Code:	

```
0 -> Dispatching event xforms-model-construct on <SPAN class="xforms-model" id="xsltforms-mainform-model-default"/>
13 -> Dispatching event xforms-model-construct-done on <SPAN class="xforms-model" id="xsltforms-mainform-model-default"/>
0 -> Dispatching event xforms-model-construct on <SPAN class="xforms-model" id="xsltforms-mainform-model-config"/>
2 -> Dispatching event xforms-model-construct-done on <SPAN class="xforms-model" id="xsltforms-mainform-model-config"/>
70 -> Dispatching event xforms-ready on <SPAN class="xforms-model" id="xsltforms-mainform-model-default"/>
0 -> Dispatching event xforms-ready on <SPAN class="xforms-model" id="xsltforms-mainform-model-config"/>
```

Figure 10-3. Address XForm with XSLTForms (debug mode)

CHAPTER 11

Basic Indexing

This chapter will take you into the mysterious and sometimes puzzling world of database indexes. As soon as your dataset starts growing and performance starts degrading as a result, indexes become a necessity.

Just for a moment let's imagine that there are no indexes. This would mean that every XPath request must be resolved by brute force. So, for a query like `//line[@author eq "erik"]`, the full document(s) node tree(s) must be traversed to try to find `line` elements with an `author` attribute that matches the value `erik`. You can probably see that on a large dataset this could be an intensive, and ultimately a slow, operation. If you further imagine running many of these queries on demand by your users in parallel, things can only get worse!

Of course, indexes come with a cost of their own: when XML documents are created or updated, the corresponding indexes must be updated too. However, this is generally not a problem. For most (but not all) applications, updating is a much rarer event than querying, and the short time lags created by updating the indexes go unnoticed.

Large databases, XML or otherwise, rarely scale well without indexes. Performance degradation as the dataset grows could be linear, or often worse. Therefore, defining and tuning indexes is well worth the effort, and often a necessity.

Besides the indexes mentioned here and in Chapter 12, there is also an index that supports explicit ordering, known as the *sort index*. Since this works differently to the indexes described here, it is handled separately in `sort`.

Indexing Example

For an introduction to the world of indexing, examine the code that accompanies the book (see "Accompanying Source Code" on page 15)in the folder *chapters/indexing*, or in the */db/apps/exist-book/indexing* collection if you have installed the XAR package. The *data* subcollection contains two XML data files with some old *Encyclopedia Britannica* entries. The contents of these files are exactly the same, with the exception that one is in the `tei` namespace and the other is not.

For a look at the index definitions for these files, open */db/system/config/db/apps/exist-book/indexing/data/collection.xconf*:

```
<collection xmlns="http://exist-db.org/collection-config/1.0">
  <index xmlns:tei="http://www.tei-c.org/ns/1.0">
    <create qname="tei:name" type="xs:string"/>
    <ngram qname="tei:p"/>
    <lucene>
      <text qname="tei:p"/>
    </lucene>
  </index>
</collection>
```

This file defines three indexes on data in the */db/apps/exist-book/indexing/data* collection:

- A *range index* on all `tei:name` elements in this collection. A range index optimizes searches on the content of elements and attributes.
- An *NGram index* on all `tei:p` elements in this collection. An NGram index optimizes searches on substrings within the contents of elements and attributes.
- A *full-text index* on all `tei:p` elements in this collection. Full-text indexes optimize searches on words and phrases within the contents of elements and attributes.

How this works exactly and what index type to use when are handled in this and the next chapter. The effect, however, of these definitions is that the data in the `tei`-namespaced file is indexed, but the (same) data without a namespace is not. This allows us to easily run the same query over the indexed and nonindexed data and compare the results.

The *test-indexes.xq* script does exactly this. It runs a number of queries over the contents of the two files and outputs the results as an XML fragment.

To show the differences between indexed and nonindexed searches more clearly (on modern, fast hardware), all searches are repeated multiple times. If your system is slower or faster than our test system, then it could be that the search is too slow or too fast for you. You can change the number of iterations performed at the top of the script.

Here is the code fragment that performs the test of the range index:

```
{
  let $starttime as xs:time := util:system-time()
  let $result := for $i in 1 to $repeats-range
    return
      for $n in doc($doc-with-indexes)//tei:name[. eq $phrase-range] return $n
  let $endtime as xs:time := util:system-time()
  return
    <Result type="indexed"
               time="{seconds-from-duration($endtime - $starttime)}s"/>
}
{
  let $starttime as xs:time := util:system-time()
  let $result := for $i in 1 to $repeats-range
    return
      for $n in doc($doc-without-indexes)//name[. eq $phrase-range] return $n
  let $endtime as xs:time := util:system-time()
  return
    <Result type="non-indexed"
               time="{seconds-from-duration($endtime - $starttime)}s"/>
}
```

Notice that we use `util:system-time`, not `fn:current-time`, to get the start and end time of a piece of code. The XQuery specification states that the value of `current-time` is deterministic throughout the execution of an XQuery. This means that it will not change during execution. As we want to measure the difference between two times to determine how long the query took, we instead use `util:system-time`, which is nondeterministic and will always return the point in time. It is worth noting that all of the date/time functions within the XPath and XQuery Functions and Operators specification are deterministic.

If you look at the output from one of our systems for this fragment, the differences between the indexed and nonindexed versions are rather dramatic:

```
<Result type="indexed" time="0.011s"/>
<Result type="non-indexed" time="0.782s"/>
```

Performance with an index is a factor of 71 times faster, and the difference will likely only increase as more data is added to the system!

Index Types

Out of the box, eXist has a number of predefined index types available. This section will tell you what they are and what they are used for. This is important, because you have to choose the right tool for the job to get the best results. Configuring and using these indexes is handled in the sections to come.

You can change the available index types (on the Java level), and even add your own. However, that's an advanced subject and not discussed here. Please refer to the online documentation if you want to know more about this.

Structural Index

eXist's structural index keeps track of the tree structure of nodes in all XML documents stored in the database. It indexes all elements and attributes, so that when you do a query like `//title`, it can quickly find all of the appropriate `title` elements. Indexing is done with the *qualified name* (or *QName*) of a node: both the namespace and the local name are used.

The structural index also automatically indexes identifier attributes. These are attributes called `xml:id` or attributes explicitly marked as of type `ID` in an attached DTD.

The structural index is used for resolving nearly all XPath and XQuery expressions. You can't configure or disable it, it is an integral part of eXist.

Range Indexes

Although the structural index allows eXist to find nodes by name quickly, it doesn't do anything with the actual values of the nodes (except `@xml:id`). So, a query like `//author[@name='erik']` will be able to find `author` elements that have a `name` attribute quickly, but after that will still have to use brute force to compare the value of the `name` attribute with `'erik'`. When the database becomes large and the queries more complex, performance will degrade rapidly.

Range indexes come to the rescue here: these indexes work on the *values* of nodes, not their names. The optimizer will use them when you do a `=`, `>`, `<`, etc., comparison in your XPath expression. They are also used for string comparisons within functions like `contains` and even for regular expression lookups with the `matches` function.

Range indexes are also useful for nonstring comparisons. You can define the index to treat the indexed value as, for instance, `xs:double`, `xs:integer`, or `xs:dateTime` (values that cannot be cast as the specified type are ignored when the index is being created). This makes a query like `//article[@price gt 100.0]` with an `xs:double` range index on the `price` attribute very efficient.

Without an index on the price attribute, the syntax for this XPath has to be `//article[xs:double(@price) gt 100.0]`. However, for the index to work, do not add type conversions like `xs:double`. Because of the index, type conversions are implicit. Adding one will actually prevent the index from working.

NGram Indexes

NGram indexes are used to make exact substring searches efficient. For example, if you often do things like searching for `'def'` in `'abcdefghij'`, an NGram index can help make this perform faster. NGram indexes retain whitespace and punctuation and are case-*in*sensitive (index=INDEX=Index). To use an NGram index, you need the functions from the `ngram` extension module. Read more about this in "Using the NGram Indexes" on page 280.

NGram indexes are called "NGram" because values are split into tokens of *n* characters, the so-called *n*-grams (for background information, see the Wikipedia "*n*-gram" article). For eXist *n* is 3 by default, which, in our experience, provides the best compromise between index size and performance. If needed, you can change the value of `n in $EXIST_HOME/conf.xml` (search for n="3").

If all your substring searches are looking for *words* (whitespace-separated), you're better off using a full-text index, as described next. However, some non-European languages don't separate their words with whitespace; this is where the NGram index comes in handy.

Full-Text Indexes

eXist's full-text indexing capabilities allow you to search the text in your documents for words and phrases using a query language with features like wildcards, fuzzy matches, and Boolean expressions. When they are used in combination with eXist's "keywords in context" (KWIC) capabilities, results can be displayed in an attractive manner, highlighting the found words within a fragment of surrounding text.

Full-text indexes in eXist are implemented via Lucene, a fast, efficient, and customizable full-text indexer.

Since full-text indexing is a detailed subject, it is handled separately in Chapter 12.

Configuring Indexes

Configurable indexes (i.e., all but the structural index) are defined at the collection level. This means that specific indexes can be defined for all XML documents in a certain collection (and any subcollections).

To define indexes for a specific collection (and its subcollections), you need to do the following:

1. The database has a system collection */db/system/config*. Under here, you create a collection path that reflects the path from the database root (*/db*) to the collection that holds the data you wish to index. For instance, if you wanted to create indexes for */db/myapp/data*, you would create the collection */db/system/config/db/myapp/data*.
2. Create an XML file there called *collection.xconf* with the following basic contents:

   ```
   <collection xmlns="http://exist-db.org/collection-config/1.0">
     <index>
       <!-- Index definitions here -->
     </index>
   </collection>
   ```

   All elements in the *collection.xconf* file must be in the `http://exist-db.org/collection-config/1.0` namespace.
3. Add the index definitions as children of the `index` element (details covered shortly, and for full text in Chapter 12).
4. If there's already data in the collection, don't forget to reindex! See "Maintaining Indexes" on page 278 for details.

Configuring Range Indexes

You may define range indexes by adding `create` elements to the index configuration in *collection.xconf*. For instance:

```
<collection xmlns="http://exist-db.org/collection-config/1.0">
  <index xmlns:ns1="http://mynamespace">
    <create qname="ns1:article" type="xs:string"/>
    <create qname="@price" type="xs:double"/>
  </index>
</collection>
```

This creates a string-type range index on every `article` element (in the `http://mynamespace` namespace) and a double-precision floating-point range index on every `price` attribute, for every document in the collection (and subcollections).

The `create` element must be a direct child of the `index` element. It takes the form:

```
<create qname = xs:QName
        type = xs:QName />
```

where:

- `qname` holds the qualified name (local name with an optional namespace prefix) of the element or attribute to index. An attribute must be prefixed with an at sign (`@`).
- `type` contains the data type of the element/attribute to index, expressed as an XML schema data type. The supported types are `xs:string` , `xs:integer`, `xs:decimal`, `xs:boolean`, `xs:dateTime`, and all their subtypes.

If the value of the `type` attribute is invalid, the index configuration is silently ignored! However, a warning indicating the problem is written to the *$EXIST_HOME/webapp/WEB-INF/logs/exist.log* logfile.

Indexing Specific Nodes

There is a second format for defining range indexes: using a `path` instead of a `qname` attribute (e.g., `<create path="//ns1:article/@price" type="xs:double"/>`). The `path` attribute can contain a limited form of XPath expression to specify the nodes to index; only element and attribute (with `@`) names, the `//` and `/` axes, and the wildcard `*` operator are allowed. Predicates are not allowed.

At first sight, this looks like it's leading to more efficient indexes as you provide the indexer with a more targeted set of elements/attributes to index, which should in turn lead to smaller and therefore faster indexes.

Unfortunately, these kinds of indexes make life extremely hard for the query optimizer. An index like this is only valid in certain contexts, and since optimization is done at compile time, when the context is frequently not yet known, most optimization techniques cannot be applied.

Therefore, our strong advice is to stick to range indexes based on qualified names and not use context-dependent indexing. In most cases, the indexes will be larger but the queries will still be faster!

Range indexing is done on the text values of the defined nodes. For example, assume you have a range index defined on `title` elements and the XML contains:

```
<title>Books written by <author>Joe Dumb</author></title>
```

The `title` range index will be on the concatenated text nodes of `title`, `"Books written by Joe Dumb"`.

Range indexes are on the defined node only, though, not on their children. This means that in the preceding example there is no index on the `author` element.

Expressions like `//title[author='Joe Dumb']` are evaluated without using any index. Luckily, nothing stops you from defining an index on both `title` and `author`.

Configuring NGram Indexes

You can define NGram indexes by adding `ngram` elements to the index configuration in *collection.xconf*. For instance:

```
<collection xmlns="http://exist-db.org/collection-config/1.0">
  <index xmlns:ns1="http://mynamespace">
    <ngram qname="ns1:article"/>
    <ngram qname="@price"/>
  </index>
</collection>
```

This creates an NGram index on every `article` element (in the `http://mynamespace` namespace) and on every `price` attribute, for every document in the collection (and subcollections).

The `ngram` element must be a direct child of the `index` element. It takes the following form:

```
<ngram qname = QName />
```

where `qname` holds the qualified name (local name with an optional namespace prefix) of the element or attribute to index. An attribute must be prefixed with an `@`.

Maintaining Indexes

Once they are defined, eXist automatically keeps all indexes up to date. Adding or updating documents will automagically update all relevant indexes.

The only time you explicitly have to reindex is when you add or change an index definition for an existing collection. Luckily, this is easy. There are several ways to do this:

From the eXist client
: Start eXist's Java Admin Client, select the right collection, and choose the menu command File→"Reindex collection."

From the dashboard's collection browser
: Open the dashboard, start the collection browser, select the right collection, and click the toolbar command "Reindex collection."

From XQuery code
: Use the extension function `xmldb:reindex`, passing it the URI of the collection to reindex. Upon completion, it will return `true` or `false`.

Using Indexes

Once they are defined, eXist tries to use the indexes as efficiently as possible, often by silently optimizing XPath expressions. You can also use them by explicitly calling certain extension functions. This section will tell you how to get the most out of your indexes.

Using the Structural Index

The structural index is a core part of eXist and cannot be avoided, even if you want to. However, the way it is implemented, as an index of qualified names, leads to some surprising effects. Being aware of this can help you write more efficient code.

When eXist evaluates a query like `//title/author`, it performs two lookups in the structural index: all `title` elements *and* all `author` elements (in the full database!). It then performs a structural join between these sets to determine which `author` elements are children of `title` elements. Because of the way internal node identifiers are built up (see "Dynamic Level Numbering of Nodes" on page 80), this is extremely efficient.

As a consequence, some of the common wisdom about XPath may not hold true for eXist. For instance, more specific queries like `/a/b/c/d/e` are often presented in textbooks as being more efficient than `//e`, but since every step in an XQuery expression would cause eXist to perform a join, the fewer steps the better. In eXist, `//e` or `/a//e` is more efficient than `/a/b/c/d/e`.

The structural index is also used for looking up identifiers with the `id` function (finding nodes with a matching `xml:id` attribute or with a matching attribute that is explicitly marked as of type `ID` in an attached DTD).

Using the Range Indexes

Range indexes are used automatically to optimize queries. For this to work, the following conditions must be met:

- The data type of the range index must match the data type used in the query.

  For instance, assume you have a range index of data type `xs:integer` defined on attribute `customerid`. A query `//Customer[@customerid eq '3456']` will not use this index because your query uses string comparison. To make use of the index, you need to rewrite this as `//Customer[@customerid eq 3456]`.

When you have an index of the `xs:double` data type, make sure your queries actually use doubles. Don't write `//Article[@price gt 100]`, but `//Article[@price gt 100.0]`.

- The query must not depend on the current context item.

  For instance, a range index on `id` attributes will not be used in queries like `//Article[@id eq ../@refid]`.

Indexes of the `xs:string` type are also used for queries that include the XPath functions `contains`, `starts-with`, and `ends-with`. However, substring searches like this are not very efficient with a range index. If you do this frequently, use an NGram index instead (and don't forget to use the special `ngram` extension functions discussed in the next section to exploit those indexes).

When you're using range indexes, if you perform a query over multiple collections and not all collections are indexed, or they are in different indexes, eXist takes the first available index and only uses this. Matches outside this index are ignored.

For instance, assume you have a range index defined on collection */db/myapp/a* but not on */db/myapp/b*. Performing a query over `collection('/db/myapp')` will return matches from */db/myapp/a* only!

Using the NGram Indexes

NGram indexes are not applied automatically in general comparisons; instead, you need to use the functions from the `ngram` extension module to exploit them.

For instance, assume there is an NGram index on `Text` elements. An NGram query for `'eXist indexes'` might look like this:

```
//Text[ngram:contains(., 'eXist indexes')]
```

Remember that NGram indexes are case-insensitive. This means that all `ngram` extension functions are case-insensitive too, in contrast to their XPath counterparts!

The `ngram` extension module includes the following functions:

`ngram:contains`, `ngram:ends-with`, `ngram:starts-with`
: These work the same way as their XPath counterparts (albeit case-insensitive).

`ngram:wildcard-contains`
: This allows searching for substrings using a limited regular expression syntax. Please refer to the function documentation for details.

General Optimization Tips

To round off this section, here are some general tips to help you make the most of eXist's optimization techniques and indexes:

Prefer short paths
: Because of the way the structural index works, queries like `/a/b/c/d` are slower than `//d`.

Prefer XPath filters over FLWOR `where` *clauses*
: Don't use `for $n in //e where @id eq 1`. Rewrite this as `for $n in //e[@id eq 1]`. eXist is much better at rewriting and optimizing predicates than `where` clauses.

Reduce the search space as early as possible
: If you have an XPath query with multiple predicates, like `//Text[contains(., 'eXist')][@id eq 1]`, you should put the most restrictive one first. So in this case (assuming identifiers are unique), `//Text[@id eq 1][contains(., 'eXist')]` will most likely perform better.

Use multiple XPath predicates instead of the `and` *operator*
: Writing `//Text[@id eq 1][contains(., 'eXist')]` is better than `//Text[@id eq 1 and contains(., 'eXist')]`. The XQuery optimizer will try to do this rewriting trick for you, but helping it by explicitly using multiple predicates won't hurt.

Use a union operator instead of an `or` *operator*
: An expression like `//Text[@id eq 1 or contains(., 'eXist')]` performs better when rewritten as `//Text[@id eq 1] | //Text[contains(., 'eXist')]`.

In all cases, of course, watch out for overdoing it. It's no use rewriting simple queries that are already fast and/or work on small datasets. Also, try to remember the old software engineering gem: maintaining code is usually more expensive than creating it. Manually rewriting a query to squeeze out every last millisecond of performance can easily obfuscate its meaning.

Debugging Indexes

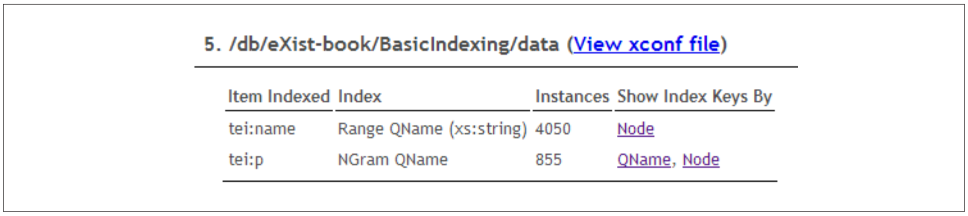
Applying indexes can feel a bit unnerving: you've defined them and all seems to be working, but is this really the case? Are your queries as efficient as you would like them to be? This can be a hard thing to test, because during development you may not have enough data at hand to really see if an index makes any difference at all. To help you with this, eXist has several ways to ascertain that your indexes are correctly defined and applied.

When operations designed to exploit indexes don't seem to be working as you would expect them to, you can of course always look in the eXist logfiles for any messages regarding the indexes.

Checking Index Definitions

An index definition is set up in the appropriate *collection.xconf* file (see "Configuring Indexes" on page 275). To find out if the index is correctly defined (and you didn't, for instance, make any syntactical mistakes), open the dashboard's Admin Web Application and select the Browse - Indexes page. There you'll find an overview of all the defined indexes per collection. Find your collection in this list and check if the intended indexes are shown.

For instance, the indexes in the collection for our indexing example (see "Indexing Example" on page 272) look like Figure 11-1.



5. /db/eXist-book/BasicIndexing/data (View xconf file)

Item Indexed	Index	Instances	Show Index Keys By
tei:name	Range QName (xs:string)	4050	Node
tei:p	NGram QName	855	QName, Node

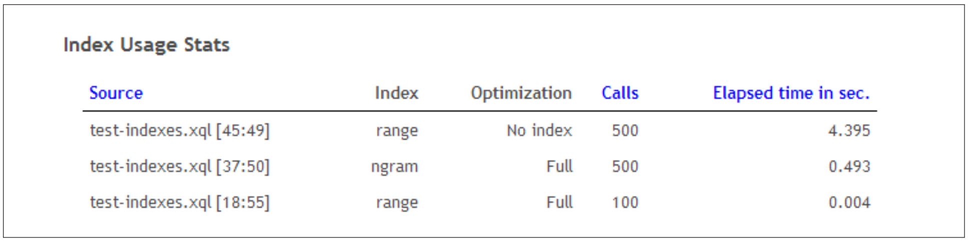
Figure 11-1. Viewing the index definitions in the Admin Web Application

The links underneath Show Index Keys By provide you with an overview of all the keys in the index. If you want a fast but database-wide list of index keys for a given QName, choose "QName." If you want a collection-specific list of index keys, choose "Node."

You might want to check the number of instances in the index. If this is less than you expect, did you perhaps forget to reindex the collection after defining the index?

Checking Index Usage

You can check whether an index is used in query optimizations by using the Tooling – Query Profiling page of the Admin Web Application. Click Enable Tracing and then run your queries that you expect to benefit from the defined indexes. Finally, come back to the admin interface, click Disable Tracing, and go to the Indexes tab. You should see something like Figure 11-2.



Index Usage Stats

Source	Index	Optimization	Calls	Elapsed time in sec.
test-indexes.xql [45:49]	range	No index	500	4.395
test-indexes.xql [37:50]	ngram	Full	500	0.493
test-indexes.xql [18:55]	range	Full	100	0.004

Figure 11-2. Index usage output

The most interesting entries here are probably the ones marked "No index." Double-check these to make sure this value is indeed what you expect, and if not, try to fix them using the information in "Using Indexes" on page 279.

There is an option on the Query Profiling page called "Write additional info to log." If you select this, additional tracing information regarding indexes is written to *$EXIST_HOME/webapp/WEB-INF/logs/profile.log*.

Tracing the Optimizer

If you need to investigate the details of what the optimizer is doing, you can enable tracing log output. This will give you a detailed list of all the (normally invisible) decisions and optimizations made.

To enable tracing, change eXist's root logging level to `trace` by opening *$EXIST_HOME/log4j.xml* and searching for the `root` element (probably at the bottom of the file). You should see something like this:

```
<root>
    <priority value="info"/>
    <appender-ref ref="exist.core"/>
</root>
```

Change the priority to `<priority value="trace"/>`, restart eXist, and rerun the scripts you want to examine. The *$EXIST_HOME/webapp/WEB-INF/logs/exist.log* file should now contain lots of detailed information regarding your queries and their optimizations.

Don't forget to reset the priority to `<priority value="info"/>` (and restart) after you're done; otherwise, the logfile will quickly fill up your disk!

CHAPTER 12

Text Indexing and Lookup

Besides the "basic" indexing capabilities, as explained in Chapter 11, eXist also supports full-text indexes based on the Apache Lucene text search-engine library. Lucene allows eXist to offer search capabilities like looking for words near each other or words like other words, using Boolean text comparison operators, and more. Full-text indexes allow you to do much more with your content than you can do using straight XPath expressions.

If your application needs to support searches based on human input, such as searching documentation or the like, full-text indexes can really help. But things get even better: on top of the full-text index searches, eXist offers *keywords in context* (KWIC) functionality. This makes it extremely easy to display the results of your searches in context, showing the search results within the surrounding text. We'll examine this further in "Using Keywords in Context" on page 297.

Full-Text Index and KWIC Example

The examples for this book include a simple full-text search example. This example searches, using the full-text index, over some ancient *Encyclopedia Britannica* entries. Important components of the example are:

- A full-text index on `tei:p` elements, defined in */db/system/config/db/apps/exist-book/indexing/data/collection.xconf*:

```
<collection xmlns="http://exist-db.org/collection-config/1.0">
  <index xmlns:tei="http://www.tei-c.org/ns/1.0">

    <!-- other indexes -->

    <lucene>
      <text qname="tei:p"/>
```

```
        </lucene>
      </index>
    </collection>
```

- An extremely simple HTML form that allows you to enter a search expression, in */db/apps/exist-book/indexing/search-demo.xq*
- A script that uses this expression to perform a search, in */db/apps/exist-book/indexing/search-demo-result.xq*

If you look at the code in *search-demo-result.xq* that actually performs the search and displays the results, there is surprisingly little there:

```
{
  for $hit in doc($doc-with-indexes)//tei:p[ft:query(., $search-expression)] ❶
  let $score as xs:float := ft:score($hit) ❷
  order by $score descending
  return (
       <p>Score: {$score}:</p>,
       kwic:summarize($hit, <config width="40"/>) ❸
  )
}
```

❶ First we do a full-text query using the `ft:query` function on `tei:p` elements. This works because we have a Lucene index defined on these elements.

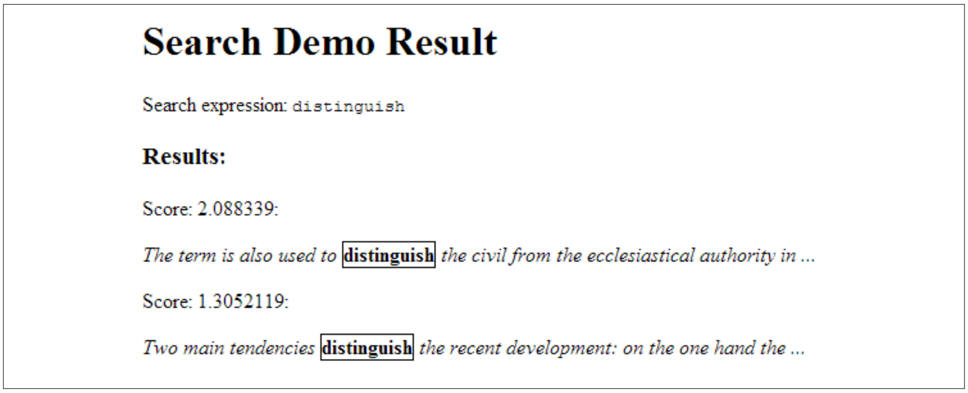
❷ Then we get the score for every search result using `ft:score`. This returns a floating-point number. The higher the number, the more relevant Lucene thinks the match is. We order the results by score, resulting in the most relevant first.

❸ The `kwic:summarize` function has the ability to summarize the search results with a bit of text before and after the actual match; the second parameter specifies that this must be 40 characters. It outputs an HTML fragment with `span` elements with different CSS classes for the trailing part, the match, and the leading part of the output. You can use this to create pretty layouts for the search results (as attempted in the example).

If you run the example and search on, for instance, `distinguish`, the results look like Figure 12-1.

Configuring Full-Text Indexes

Configuring a full-text index is done in the same *collection.xconf* document we used for the other indexes. For more information on where to locate such a document and its basic syntax, please refer to "Configuring Indexes" on page 275.



Search Demo Result

Search expression: distinguish

Results:

Score: 2.088339:

The term is also used to **distinguish** *the civil from the ecclesiastical authority in ...*

Score: 1.3052119:

Two main tendencies **distinguish** *the recent development: on the one hand the ...*

Figure 12-1. Example output for a search on "distinguish"

The full-text index definition is a child of the `index` element and, as is true of everything in a *collection.xconf* file, it is in the `http://exist-db.org/collection-config/1.0` namespace. It has the following structure:

```
<lucene>
  analyzer*
  text+
  ( inline | ignore )*
</lucene>
```

- The optional `analyzer` element allows you to change the analyzer class(es) Lucene uses to analyze the text and/or pass it parameters. This is an advanced topic, explained in "Defining and Configuring the Lucene Analyzer" on page 298.
- The `text` element defines which elements/attributes Lucene creates an index for. See more about this in "Choosing the correct context" on page 288.
- The `inline` and `ignore` elements are important when you're indexing mixed content. They can be defined either globally (as a child of `lucene`) or for a specific `text` element. Read more about this in "Handling Mixed Content" on page 290.

Configuring the Search Context

The `lucene/text` element defines the context for the full-text index. This is usually an element, but could just as well be an attribute. It has the following structure:

```
<text qname = string | match = string
      boost? = float
      analyzer? = NCName >
  ( inline | ignore )*
</text>
```

- qname is the qualified name of the element or attribute (if you start it with an @ character) for which you want the text to be indexed. It can have a namespace prefix (which must, of course, be defined).

  Examples of qname usage are qname="tei:p", qname="mytextelement", and qname="@title".

- With match you can define the search context using a limited XPath-like expression. Only the / and // operators are allowed, plus the wildcard * to match an arbitrary element.

  For instance, match="//tei:div/*" will put a full-text index on all direct child elements of tei:div.

You may use either a qname or a match attribute, but not both:

- The boost attribute gives you the ability to influence the scoring of matches during indexing. It multiplies the default search score with this floating-point value. There's more about search scores in "Scoring Searches" on page 296.

- analyzer allows you to change the Lucene analyzer class for this index. This is explained in "Defining and Configuring the Lucene Analyzer" on page 298.

Additionally, you can define how inline elements must be treated for this particular index using nested inline and/or ignore elements. This is explained in "Handling Mixed Content" on page 290.

Choosing the correct context

Defining the correct context for your full-text index is critical. For example, take the following XML fragment:

```
<Text>
  <Heading>eXist index configuration</Heading>
  <Content>eXist index configuration is done in
    the collection.xconf file</Content>
</Text>
```

Assume we've indexed this with the following index configuration in *collection.xconf*:

```
<lucene>
  <text qname="Text"/>
</lucene>
```

Passed to the indexer is the text value of the nodes identified by the qname attribute. So, in this example the indexer will see "eXist index configuration eXist index configuration is done in the collection.xconf file". This is linked to the Text element only; Lucene preserves no knowledge about the child elements of Text.

If you use this configuration, the following query will return the expected Text elements:

```
//Text[ft:query(., 'index')]
```

However, the following query will return nothing (i.e., an empty sequence) because Lucene has no index on Heading elements:

```
//Text[ft:query(Heading, 'index')]
```

Searching only within the contents of headings is often desirable, so you may in fact want this to work. Luckily, nothing stops you from defining two overlapping indexes:

```
<lucene>
  <text qname="Text"/>
  <text qname="Heading"/>
</lucene>
```

Here is another useful example. Assume we've marked up filenames separately, as in:

```
<Text>
  <Heading>eXist index configuration</Heading>
  <Content>eXist index configuration is done in the
    <Filename>collection.xconf</Filename> file</Content>
</Text>
```

To give the user the option to search within the full text *or* within either the filenames *only*, we define two indexes:

```
<lucene>
  <text qname="Text"/>
  <text qname="Filename"/>
</lucene>
```

Now the following two queries will return the same Text element, even though the search context for the second one is much narrower:

```
//Text[ft:query(., 'collection.xconf')]
//Text[ft:query(Filename, 'collection.xconf')]
```

Search context and performance

Full-text indexing is an expensive operation and can have a huge impact on the performance of storing and updating documents, so use it wisely.

It's important not to define full-text indexes too broadly. For instance, a classical mistake is defining your full-text indexes as:

```
<text match="//*" />
```

This will create an index on all elements anywhere in your document. That may sound simple and attractive, but it will cost you dearly in terms of performance. Remember that the index is created over the text contents of a node, so an index on the root element will index all text in the document. The same is true for all other elements, so every piece of text is indexed multiple times. Depending on how deeply the text is nested in the document, this may be slow and create a huge number of index files.

So, the best strategy for full-text indexes is to define them as narrowly as you can. And be careful using wildcards, because they can quickly get out of hand!

Handling Mixed Content

You can decide how to handle mixed content by using the `inline` and `ignore` elements. These elements can appear globally (as children of the `lucene` element) or per index (as children of the `text` element). `inline` also has an effect on how Lucene treats whitespace. They have the following format:

```
<inline qname = string />
<ignore qname = string />
```

`qname` holds the qualified name (with an optional namespace prefix) of the inline element.

Inline content and whitespace

By default, Lucene treats inline elements as token separators, which may or may not be what you want. For instance, assume we have an XML fragment like:

```
<p>This is <b>un</b>clear.</p>
```

Because of the `b` inline element, Lucene will see this as `"This is un clear."` (notice the space between `un` and `clear`)—probably not what you intended! To address this, use an index definition like:

```
<lucene>
  <text qname="p">
    <inline qname="b"/>
  </text>
</lucene>
```

Or, if the `b` element is always an inline element in all other elements of the collections documents:

```
<lucene>
  <text qname="p"/>

  <!-- other text indexes -->

  <inline qname="b"/>
</lucene>
```

Ignoring inline content

You can tell eXist to completely ignore inline content by using the `ignore` element. This is useful when, for instance, your content contains editorial notes like:

```
<p>Columbus discovered Finland in 1492
  <note>I don't think the year is correct, could someone check this?</note></p>
```

Ignoring the `note` elements within the *p* elements can be done with:

```
<lucene>
  <text qname="p">
    <ignore qname="note"/>
  </text>
</lucene>
```

Or, when `note` should be ignored in all documents in the collection:

```
<lucene>
  <text qname="p"/>
  <!-- other text indexes -->
  <ignore qname="note"/>
</lucene>
```

An `ignore` element only ignores *descendants* of the indexed element. This means that a seemingly contradictory index definition like this one is perfectly valid:

```
<lucene>
  <text qname="p"/>
  <text qname="note"/>
  <ignore qname="note"/>
</lucene>
```

With this definition, the following query would return nothing:

```
//p[ft:query(., 'check this')]
```

But an editor searching on notes within paragraphs would get a result by using:

```
//p[ft:query(note, 'check this')]
```

Maintaining the Full-Text Index

Basic maintenance of the full-text index is the same as for the other indexes (see "Maintaining Indexes" on page 278): once defined, eXist maintains them automatically for the most part. But when you create a new one or change a configuration, you have to reindex manually.

In previous versions of eXist it was necessary to call `ft:optimize` now and then for optimal performance. This is no longer the case.

Searching with the Full-Text Index

Using the full-text index to search for words and phrases is done through the extension functions in the `lucene` extension module (see `ft`). The default namespace prefix for this module is `ft`.

Basic Search Operations

The basic function for using full-text indexes is `ft:query`. We've already seen some examples of its use in "Full-Text Index and KWIC Example" on page 285. Its full definition is:

```
ft:query($nodes, $query, [$options])
```

where:

`$nodes`
: Contains the node set to search.

`$query`
: Contains the search query. If this is a string, it is assumed to be in Lucene's native query syntax (described in the next section). For more complex queries, you may provide an XML fragment as described in "The full-text query XML specification" on page 293.

`$options`
: An optional parameter that contains additional query options. See "Additional search parameters" on page 295.

Lucene's native query syntax

Lucene has a native query syntax for defining full-text searches. Its full definition can be found at *http://lucene.apache.org/core/3_6_1/queryparsersyntax.html*. Here are some examples:

- `exist database` searches for text with the terms `exist` and/or `database`. This is equivalent to writing `exist OR database`.
- You can use wildcards like `data*` for multiple unknown characters, or `database?` for a single unknown character.
- If you want to search on a phrase (multiple words), use quotes: `"exist database"`.
- You can also do a proximity searche. `"exist database"~10` means that the words "eXist" and "database" must occur within 10 words of each other.

- For a fuzzy search (words like the search term), add a tilde character at the end, as in `database~`.
- Add a `+` in front of words and phrases that must occur. `+exist database` means the text *must* include the word `exist` but *may* include the word `database`.
- Add a `-` in front of words and phrases that should *not* occur in the text.
- Boolean operators (`AND`, `OR`, and `NOT`) are supported.
- You can group expressions using parentheses.

The full-text query XML specification

The `ft:query` function also accepts an XML fragment that allows you to build a query using Lucene's internal API indirectly. The XML is transformed into an internal representation used by Lucene and then executed. This fragment takes the following form:

```
<query>
 ( term | wildcard | regex | phrase | near | bool )+
</query>
```

All subelements accept an optional `boost` attribute of type `xs:float` to specify a boost value for the score (see "Scoring Searches" on page 296). The `query` element can contain the following subelements:

- The `term` element defines a single term to search for. The following example searches for text containing `exist` and/or `database`:

  ```
  <query>
    <term>exist</term>
    <term>database</term>
  </query>
  ```

- The `wildcard` element is the same as the `term` element but can contain a wildcard `*` or `?` operator. To search for all text with words starting with `data`, use:

  ```
  <query>
    <wildcard>data*</wildcard>
  </query>
  ```

- The `regex` element contains a regular expression used for the search.
- The `phrase` element searches for a group of terms *in the correct order*. It can contain text (which is tokenized into terms), or a number of `term` child elements. The following two examples are equivalent:

  ```
  <query>
    <phrase>exist database</phrase>
  </query>
  ```

```
<query>
  <phrase>
    <term>exist</term>
    <term>database</term>
  </phrase>
</query>
```

- With the `near` element, you can build even more specific phrase queries. Its syntax is:

  ```
  <near slop? = integer
        ordered? = "yes" | "no" >
    #PCDATA | ( term | first | near )+
  </near>
  ```

  — The optional `slop` attribute allows you to define the "slop" for the matching. Slop is the maximum number of other words between the words searched upon.

  — The optional `ordered` attribute defines whether or not the terms must be in the defined order. The default is `"yes"`.

  — If the `near` element contains character data, this is tokenized. The effect is the same as using the `phrase` element with character data.

  — Instead of tokenized character data, you can use nested `term` elements.

  — The `first` element allows you to search against the start of the text. It has an optional `end` attribute to specify the maximum distance (in words) allowed from the start of the text:

  ```
  <first end? = integer >
    #PCDATA | ( term | near )+
  </first>
  ```

  — To allow even more complex search expressions, you can nest `near` elements within one another, or within `first` elements.

  For instance, the following expression will search for nodes with the word `exist` somewhere in the first 4 words of the text and the word `database` within 10 other words from this:

  ```
  <query>
    <near slop="10">
      <first end="4">exist</first>
      <term>database</term>
  </query>
  ```

- The `bool` element allows you to combine the other elements into a Boolean expression. For this, all elements accept an `occur = "must" | "should" | "not"` attribute:

— "`must`" means that this part of the query must be matched.

— "`should`" means that this part of the query *should* be matched, but doesn't necessarily *need* to be.

— "`not`" means that this part of the query must not be matched.

For instance, searching for text that contains the term `exist` but not the term `database` can be expressed by:

```
<query>
  <bool>
    <term occur="must">exist</term>
    <term occur="not">database</term>
  </bool>
</query>
```

Additional search parameters

The optional third parameter to `ft:query` contains an XML fragment that sets a number of miscellaneous parameters for the search operation (all elements are optional):

```
<options>
  <default-operator> or | and </default-operator>?
  <phrase-slop> integer </phrase-slop>?
  <leading-wildcard> yes | no </leading-wildcard>?
  <filter-rewrite> yes | no </filter-rewrite>?
</options>
```

- The `default-operator` element sets the default operator with which multiple terms are combined. The default is `or`.
- The `phrase-slop` element sets the maximum distance (measured in words) between terms within phrases. The default is `0`.
- The `leading-wildcard` element sets whether the wildcard characters `*` and `?` are allowed as the *first* character of a wildcard expression. The default is `no`.
- The `filter-rewrite` element determines how terms are expanded for wildcard or regular expression searches. If set to `yes`, Lucene will use a filter to preprocess matching terms. If set to `no`, all matching terms will be added to a single Boolean query, which is then executed. This may generate a "too many clauses" exception when applied to large datasets. The default is `yes`.

Scoring Searches

Lucene tries to attach a relevance score to the search results. This is always a positive floating-point number. The higher the number, the more relevant Lucene thinks the result is. You can retrieve the score for a search result by calling the `ft:score` function. Here is an example:

```
for $hit in doc('/db/myapp/doc.xml')//p[ft:query(., 'exist database')]
let $score as xs:float := ft:score($hit)
order by $score descending
return
      (: results code here :)
```

How exactly Lucene computes these scores is a complex topic in its own right; you can read more about the specifics of Lucene's approach here: *http://lucene.apache.org/core/3_6_1/scoring.html*. In many cases, the specifics are not important; it is enough to trust that the Lucene score is a good approximation of what we, mere humans, consider relevant.

Locating Matches

When you perform a full-text search like `//p[ft:query(. 'database')]`, the results you get are the matching `p` elements. Some applications, however, need to know where in the text of the resulting elements the actual matches were. For example, if you offer a documentation search, it would be nice to show in the results which pieces of text matched the query.

Although used mostly for full-text search results, locating matches also works for NGram search results.

To enable this, eXist not only returns the results of the query, but also invisibly remembers where the matches were. Nothing happens if you don't use this information, but if you need it, it's there.

As an example, let's assume we've done a full-text query on the word `database` that resulted in a single `p` element:

```
<p>eXist is a native XML database.</p>
```

To find out where the matches were, you can call the extension function `util:expand` on the search result. This will wrap the matches in `exist:match` elements (the `exist` namespace prefix is bound to `http://exist.sourceforge.net/NS/exist`). A call to `util:expand` on this search result would therefore return:

```
<p>eXist is a native XML <exist:match
     xmlns:exist="http://exist.sourceforge.net/NS/exist">
     database</exist:match>.</p>
```

By default, `util:expand` will also expand any XIncludes (`xi:include` elements; see "XInclude" on page 243) in the search result. If you don't want XInclude expansion, you can specify an optional second argument to the function, which accepts serialization parameters (as defined in "Serialization Options" on page 119) that you can use to control this.

Using Keywords in Context

As we saw in the previous section, eXist remembers where the matches were for full-text (and NGram) queries. This allows you to use a feature called "keywords in context," or KWIC, that can show these matches to the user, surrounded by limited parts of the text. If you followed the example explained in "Full-Text Index and KWIC Example" on page 285, you've seen this in action already.

You can generate KWIC output using the `kwic` extension module. This is an XQuery module, and thus (as fully explained in Chapter 7) you'll have to import it explicitly in your query's prolog:

```
import module namespace kwic="http://exist-db.org/xquery/kwic";
```

If you look at the documentation for the `kwic` module, you'll see lots of functions; most of these, however, are internal.

The easiest way to use the `kwic` module is by calling `kwic:summarize` on a search result. This will return the matches, surrounded with customizable chunks of text, in HTML, ready for display. To find out where these matches are, it uses the match locating functionality as explained in "Locating Matches" on page 296. We've already seen this in action, in the example at the beginning of this chapter.

The full definition of the `kwic:summarize` function is:

```
kwic:summarize($search-result, $options)
```

The `$options` parameter accepts a small XML fragment that allows you to customize the function's behavior:

```
<config width = integer
        table? = "yes" | "no"
        link? = string />
```

- `width` (mandatory) tells KWIC how much text (expressed in characters) to keep before and after the match.
- Omitting `table` or setting it to `"no"` causes the output to be wrapped in a `p` element:

```
<p>
  <span class="previous">... text before the </span>
  <span class="hi">match</span>
  <span class="following"> and after the match...</span>
</p>
```

Setting `table` to `"yes"` causes the output to be returned in an HTML table row format:

```
<tr>
  <td class="previous">... text before the </td>
  <td class="hi">match</td>
  <td class="following"> and after the match...</td>
</tr>
```

- If you specify `link`, the match will be enclosed in an `a` element with the value of this attribute as its target. For example, specifying `link="otherpage"` will change the output for the match to:

```
<span class="hi"><a href="otherpage">match</a></span>
```

Defining and Configuring the Lucene Analyzer

Lucene allows its users to specify how text is analyzed. Analyzers are Java classes, with each one defining a different way of tokenizing and/or filtering text. There are several prebaked analyzers available. If you're indexing a language other than English, it might be worthwhile to change the analyzer to one especially tailored for your language. Other reasons might include changing the list of *stopwords* (words ignored by the analyzer).

A list of available analyzers can be found in the Lucene JavaDocs the list of direct subclasses here tells you which analyzers are available.

By default, eXist uses the standard analyzer `org.apache.lucene.analysis.standard.StandardAnalyzer`. Although called "standard," it is actually an English analyzer (and contains a list of the most-often-used English stopwords).

You can define and configure a different Lucene analyzer in the Lucene definition of the *collection.xconf* document, as explained fully in "Defining and Configuring the Lucene Analyzer" on page 298. The `analyzer` element defines the Lucene analyzer to use:

```
<analyzer class = string ❶
          id? = NCName > ❷
  param*
</analyzer>
```

❶ `class` holds the name of the Java class to use for tokenizing and filtering the text; for instance, `"org.apache.lucene.analysis.WhitespaceAnalyzer"`.

❷ `id` defines the identifier for this analyzer. This is for referencing the analyzer (in `text` elements using the `analyzer` attribute). If you don't specify an `id`, this changes the default analyzer.

An analyzer definition can contain parameters to pass to the analyzer using `param` elements. These parameters are passed to the constructor of the analyzer class:

```
<param name = string
       type? = string
       value? = string >
  value*
</param>
```

- `name` is the name of the parameter.
- `type` is the (Java) type of the parameter. Several types are currently supported:

`java.lang.String`
: A string that may be either a literal value, the name of a class, or the fully qualified name of an enumeration value, depending on the parameter context.

`java.io.File`
: A path to a file on the filesystem; it must be in the appropriate Java path syntax for the operating system in use.

`java.util.Set`
: Assumed to be a set of `java.lang.String`. When this is used, we can provide multiple values; for example:

```
<param name="stopwords" type="java.util.Set">
  <value>and</value>
  <value>or</value>
  <value>the</value>
  <value>a</value>
  <value>an</value>
  <value>this</value>
  <value>there</value>
</param>
```

`java.lang.Integer` *(or* `int`*)*
: An integer.

`java.lang.Boolean` *(or* `boolean`*)*
: A Boolean.

`java.lang.reflect.Field`
: Used to reference a static field from another class. For example, if there were a static field named `STOPWORDS` in the class `org.something.text.Common`:

```
<param
  name="stopwords"
  type="java.lang.reflect.Field"
  value="org.something.text.CommonStopWords"/>
```

When no type is specified, the default is assumed to be `java.lang.String`.

- `value` contains the value of the parameter, using either the `<param name="a" value="b"/>` or the `<value>a</value>` form (when `type` is a `java.util.Set`).

If the parameters need more than one value, use embedded `value` elements instead of the `value` attribute (not both).

A simple example of changing the analyzer would be to tell Lucene that the text we're going to index is in Dutch:

```
<lucene>
  <analyzer class="org.apache.lucene.analysis.nl.DutchAnalyzer"/>
  <text qname="p"/>
</lucene>
```

For a more advanced example of defining analyzers and passing parameters, we use the ability of the standard analyzer to define a set of stopwords (as mentioned, these are words to be ignored, like *the*, *a*, *an*, etc.). The following example changes the default analyzer and passes it a set of stopwords in a text file:

```
<lucene>
  <analyzer class="org.apache.lucene.analysis.standard.StandardAnalyzer">
    <param
      name="stopwords"
      type="java.io.File"
      value="/usr/local/exist/webapp/WEB-INF/data/stopwords.txt"/>
  </analyzer>
  <text qname="p"/>
</lucene>
```

Now assume you need some other element indexed also, but with a much more limited set of stopwords. This could be accomplished by:

```
<lucene>
  <analyzer class="org.apache.lucene.analysis.standard.StandardAnalyzer">
    <param name="stopwords" type="java.io.File"
      value="/usr/local/exist/webapp/WEB-INF/data/stopwords.txt"/>
  </analyzer>
  <analyzer id="a2"
    class="org.apache.lucene.analysis.standard.StandardAnalyzer">
    <param
        name="stopwords"
```

```
          type="java.io.Set">
        <value>the</value>
        <value>a</value>
        <value>an</value>
      </param>
    </analyzer>
    <text qname="p"/>
    <text qname="h1" analyzer="a2"/>
  </lucene>
```

Now the h1 element is indexed with the stopwords *the*, *a*, and *an* only.

Manual Full-Text Indexing

There is yet another way to use the Lucene full-text indexer inside eXist. You can manually (through your own XQuery code) create an index associated with a resource in the database. You can then use this index to query the contents of this resource. Interestingly enough, the resource does not have to be an XML document, so, in conjunction with the contentextraction extension module (see contentextraction), you can create indexes to search binary content!

Here is how it works:

1. For some resource in your database (XML or otherwise), extract (or create) the text fragments you want to index. For instance, assume we have an XHTML document for which we want to index all the p and h3 elements. We also want to be able to search the p and h3 elements separately.
2. Create an XML fragment with root element doc in which you list all these text fragments and add them to so-called *fields*. A field can be seen as a subindex on a document, so in our case we create two fields: one for the h3 elements, called headers, and one for the p elements, called paras. Here is the code that does this:

```
declare namespace xhtml="http://www.w3.org/1999/xhtml";

let $resource := '/db/path/to/your/xhtml/document'
let $index-def :=
  <doc>
  {
    for $header in doc($resource)//xhtml:h3
    return
      <field name="headers" store="yes">{ string($header) }</field>
  }
  {
    for $para in doc($resource)//xhtml:p
    return
      <field name="paras" store="yes">{ string($para) }</field>
  }
  </doc>
```

3. Call the `ft:index` function to create the index for this specific resource:

   ```
   ft:index($resource, $index-def)
   ```

   Now Lucene creates an index with two subindexes (fields). It indexes all the text fragments passed in the `doc/field` elements and stores this information, *together with the indexed text* (because the `store` attribute is set to `"yes"`).

   If you don't store the text (by setting the `store` attribute to `"no"`), the text is indexed but cannot be retrieved. The only thing you can do with an index without stored text is find out whether or not a certain phrase is present; you can't get its context.

4. Using such an index is done via the extension function `ft:search`. The search expression passed must contain the field name as a prefix. So, for instance, to search the paragraphs for the word *eXist*, you would do something like this:

   ```
   ft:search($resource, 'paras:eXist')
   ```

   Which would return an XML fragment like:

   ```
   <Indexing file="/db/path/to/your/xhtml/document">
     <results>
       <search uri="/db/path/to/your/xhtml/document" score="0.5260675">
         <field name="paras">Please use
           <exist:match xmlns:exist="http://exist.sourceforge.net/NS/exist">
             eXist</exist:match>
           for storing your information. You know why!
         </field>
       </search>
     </results>
   </Indexing>
   ```

   Information about the meaning of the `score` attribute can be found in "Scoring Searches" on page 296.

An interesting use case for this manual index creation is that of indexing binary content. You do so by first extracting the content from the binary resource using the `contentextraction` extension module (see `contentextraction`), then creating an index for it as just described. The book's sample code contains a short example of how to do this in the *chapters/indexing/index-binary.xq* file (or in the */db/apps/exist-book/chapters/indexing/index-binary.xq* file if you have installed the XAR package). There is also an interesting article about content extraction and binary resource indexing available on the eXist wiki.

CHAPTER 13
Integration

eXist provides many APIs, each of which allow you to integrate or interact with eXist in a different manner. Multiple APIs are provided in the hope that at least one of them is already supported fin the system or application that the developer or user wishes to integrate with eXist.

eXist provides two classes of API:

Local APIs
: These are intended for when a developer wishes to embed eXist as a library within his own application running on the JVM.

Remote APIs
: These are intended for when eXist is run as a server and a user or application wishes to make requests to eXist. All of the remote APIs are developed as layers atop HTTP. There is nothing to stop you from using a remote API from the same machine that eXist is running on.

We have found that the majority of eXist users are interested in the remote class of API, as they wish to use eXist as a database server and/or a web application server, so we will focus on the remote APIs first.

Choosing an API

As you are about to see, eXist offers many options for integration with existing systems and programming languages. Choosing the right one can be confusing, so we have produced the flowchart in Figure 13-1 to help you with your decision.

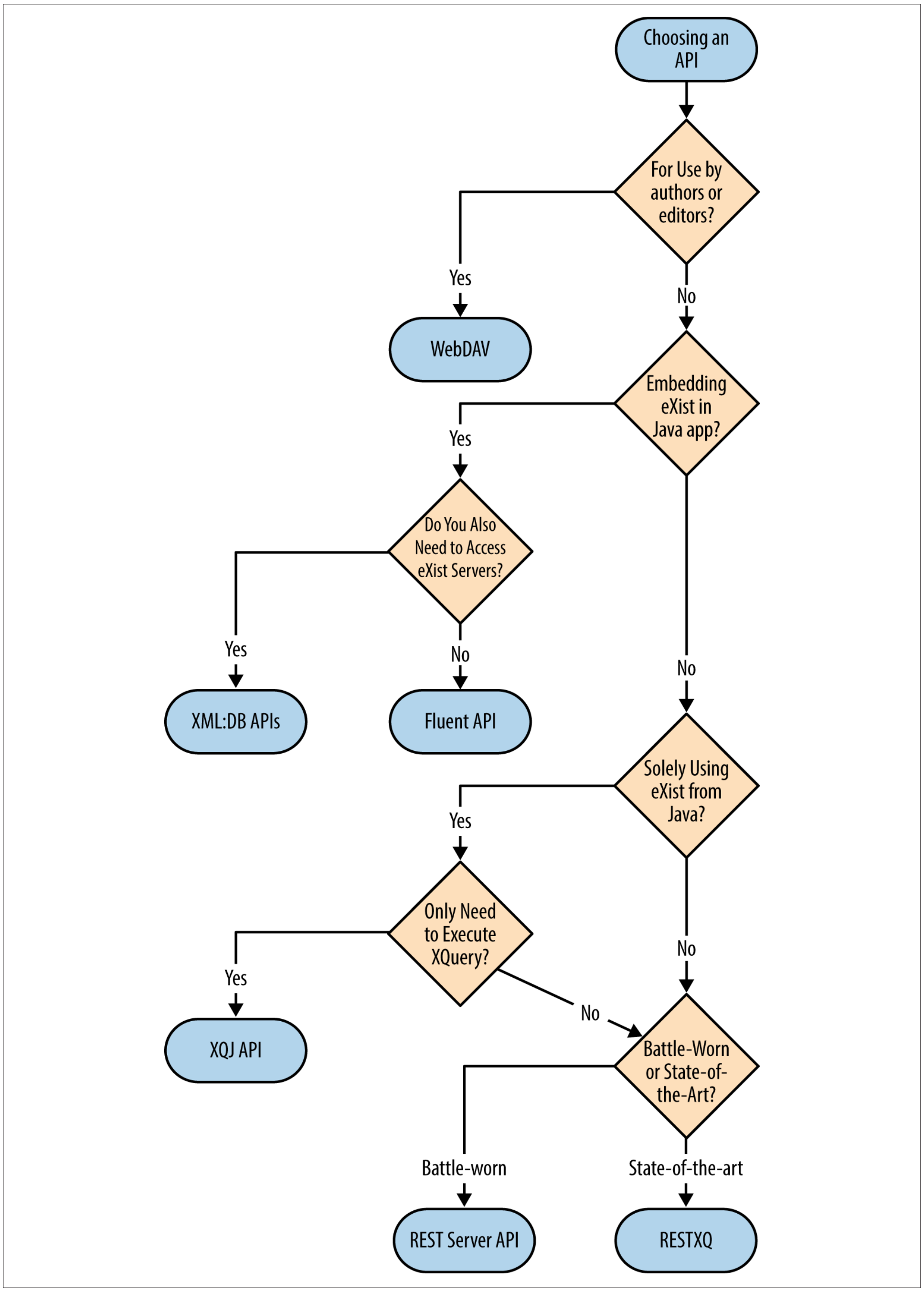

Choosing an API
For Use by authors or editors?
Yes
WebDAV
No
Embedding eXist in Java app?
Yes
Do You Also Need to Access eXist Servers?
Yes
XML:DB APIs
No
Fluent API
No
Solely Using eXist from Java?
Yes
Only Need to Execute XQuery?
Yes
XQJ API
No
No
Battle-Worn or State-of-the-Art?
Battle-worn
REST Server API
State-of-the-art
RESTXQ

Figure 13-1. Flowchart to help you choose an API for integration

The last choice in this flowchart—*battle-worn* or *state-of-the-art*?—may need some clarification. By way of explanation, the REST Server API is stable and has been around for some years, with many organizations frequently using it. So, if you have a short-term project in mind that needs to be delivered immediately on a solid technology base, this is probably the correct choice for you. Conversely, the RESTXQ framework is relatively new and easier to use, but while there are several organizations already using it, it is still very much under development. Many believe that RESTXQ will eventually replace the REST Server API, as it offers a superset of that functionality.

Remote APIs

There are many remote APIs available for eXist, and in addition it is possible to develop your own RESTful HTTP APIs using XQuery with either the REST Server API (see "REST Server API" on page 319) or RESTXQ (see "RESTXQ" on page 353).

Which API you should use depends on many factors, but if your concern is users manipulating documents we would recommend the WebDAV API (see the next section) for its simplicity and ease of use. Likewise, if you want to quickly build a simple REST API, RESTXQ (see "RESTXQ" on page 353) could be a good candidate. If you are serious about building a stable bridge with eXist, you should study each option available to you in this chapter before making a decision, as each has its advantages and disadvantages.

WebDAV

Web Distributed Authoring and Versioning (WebDAV) is an IETF standard (RFC 4918) that focuses predominantly on the distributed authoring of documents. The name can be somewhat confusing, because while versioning was initially a consideration, it was perceived as too complicated and shelved. Versioning was later added as an extension to WebDAV in IETF standard RFC 3253. However, versioning with WebDAV does not seem to have been widely adopted and is not yet supported in eXist.

While eXist has had WebDAV support for several years, its interoperability with some WebDAV clients was less than perfect. eXist 2.0 added a complete rewrite of the WebDAV server based on the excellent Milton Java WebDAV Server Library. Milton does a great job of ensuring compatibility with almost all WebDAV clients. For a list of compatible WebDAV clients, see *http://milton.io/guide/m2/docs/compat.html*.

WebDAV is most useful for those who wish to work at the document level (for example, content authors). It is very simple to create and edit documents, and also to manage them by organizing them into folders (collections in eXist) or removing old documents. Many operating systems and other tools have WebDAV support built in, so making use of WebDAV in eXist will come naturally to many—the client is the same as the file manager on your computer (e.g., Microsoft Windows Explorer, Mac OS X Finder, and Gnome Nautilus on Linux).

Connecting to eXist using WebDAV with the oXygen XML Editor is covered in "Connecting with oXygen Using WebDAV" on page 376.

The base URI of the WebDAV Server in eXist on a default installation is *http://localhost:8080/exist/webdav/db/*, or for secure access, *https://localhost:8443/exist/webdav/db/*.

Using WebDAV from Microsoft Windows

Microsoft Windows has had WebDAV support built in since Windows 98 was released. Here we will show you how to use WebDAV from Windows 7 with eXist 2.1.

Windows 7 has some mandatory security restrictions around WebDAV access. This means that Windows 7 will not work with eXist by default, as basic authentication is disabled. However, you can re-enable basic authentication for WebDAV in Windows 7 by modifying a registry setting; you do so from the command prompt (you must have *Administrator* rights) by executing the following:

```
reg add HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet
\Services\WebClient\Parameters /v BasicAuthLevel
/t REG_DWORD /d 2
```

When prompted to overwrite the existing value, type **yes** and press Return. You may have to restart your PC for the changes to take effect.

Further details can be found in the Microsoft Knowledge Base article 841215.

Follow these steps to map a drive to eXist WebDAV from Windows Explorer:

1. First open Windows Explorer, and then press the Alt key on the keyboard once to reveal the menu. Choose Tools→"Map network drive" as shown in Figure 13-2.

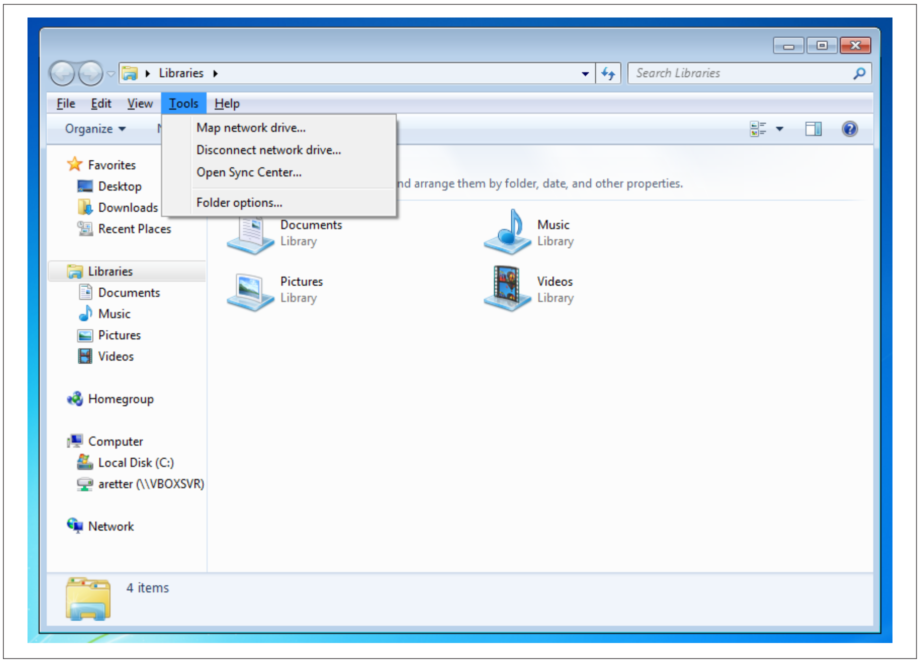
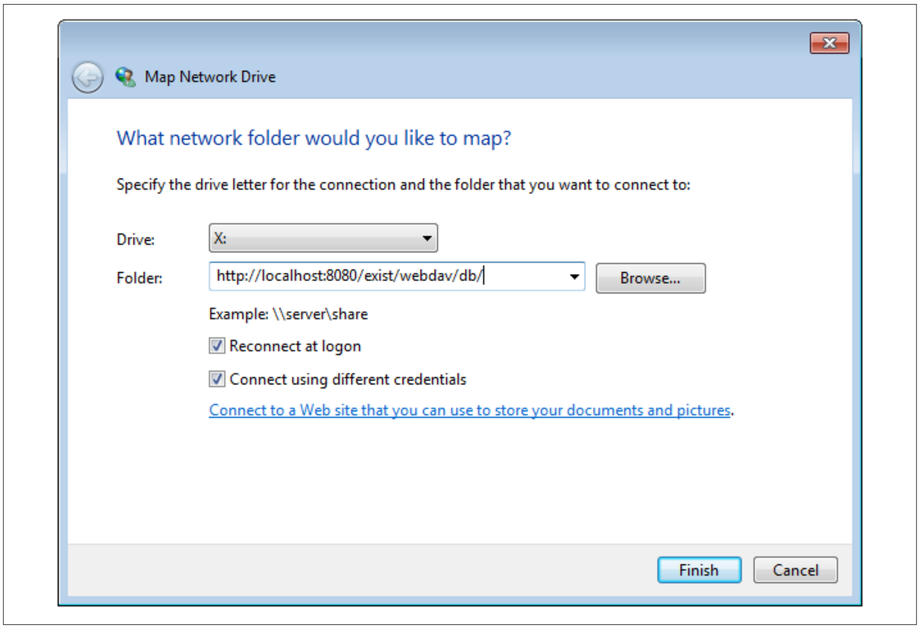


Figure 13-2. Windows Explorer: select the "Map network drive" menu item

2. Next, you need to choose a Windows drive letter to map to the eXist database. You then also need to add in the URI of the eXist server and its WebDAV API. Typically, this takes the form *http://<myserver>:8080/exist/webdav/db/*. If you are running eXist on your own Windows PC, you can replace *<myserver>* with *localhost*. Ensure that the checkbox "Connect using different credentials" is checked, and if you wish the connection to be available after restarting or shutting down your PC, ensure that the "Reconnect at logon" checkbox is checked. Finally, click Finish. See Figure 13-3.


Map Network Drive

What network folder would you like to map?

Specify the drive letter for the connection and the folder that you want to connect to:

Drive: X:

Folder: http://localhost:8080/exist/webdav/db/

Example: \\server\share

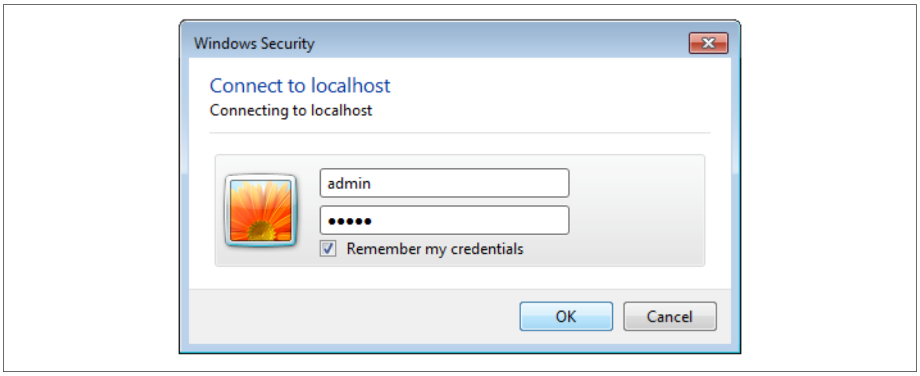
☑ Reconnect at logon

☑ Connect using different credentials

Connect to a Web site that you can use to store your documents and pictures.

Figure 13-3. Windows Explorer Map Network Drive dialog

3. Finally, you need to provide the username and password of your eXist user account. If you have just set up eXist or will be the only user, you can use the default built-in `admin` user and the password that you set during the installation of eXist. See Figure 13-4.


Windows Security

Connect to localhost

Connecting to localhost

admin

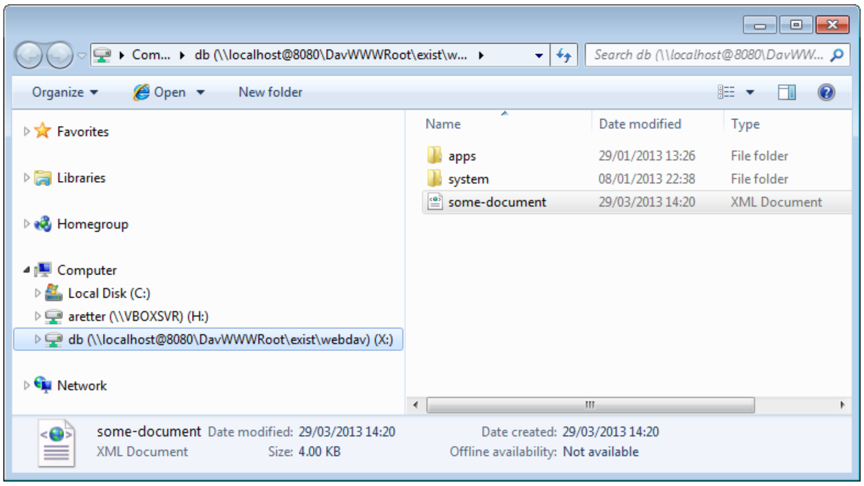
●●●●●

☑ Remember my credentials

Figure 13-4. Connect using your eXist WebDAV username and password

4. You can now access the eXist database via the Windows drive letter that you chose in step 2. As far as Windows is concerned, eXist is just another filesystem, and you can use any Windows application to read and write documents in eXist.

You can also create/move/delete collections in this manner, as they appear to Windows Explorer as regular folders. See Figure 13-5.


Com... ▸ db (\\localhost@8080\DavWWWRoot\exist\w... ▸
Search db (\\localhost@8080\DavWW...
Organize
Open
New folder
Favorites
Libraries
Homegroup
Computer
Local Disk (C:)
aretter (\\VBOXSVR) (H:)
db (\\localhost@8080\DavWWWRoot\exist\webdav) (X:)
Network

Name	Date modified	Type
apps	29/01/2013 13:26	File folder
system	08/01/2013 22:38	File folder
some-document	29/03/2013 14:20	XML Document

some-document Date modified: 29/03/2013 14:20 Date created: 29/03/2013 14:20
XML Document Size: 4.00 KB Offline availability: Not available

Figure 13-5. Windows Explorer with WebDAV connection to eXist

Using WebDAV from Mac OS X

Apple's Mac OS X has always had WebDAV support built in. Here we will show you how to use WebDAV from Mac OS X (in our example, we used OS X Mountain Lion [10.8.3]) with eXist 2.1.

Follow these steps to mount eXist WebDAV from the Finder:

1. Open the Finder, and, from the Go menu, select the menu item "Connect to Server" (or press Command-K). See Figure 13-6.

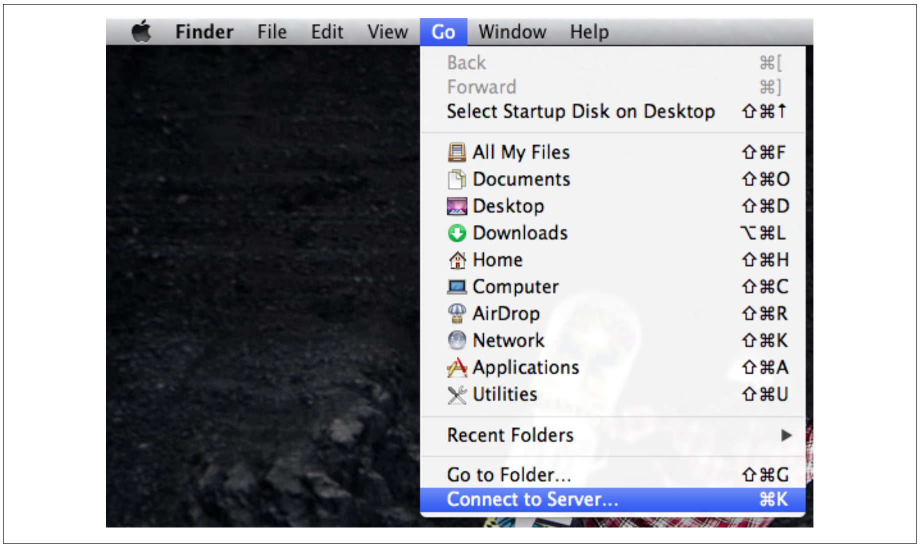
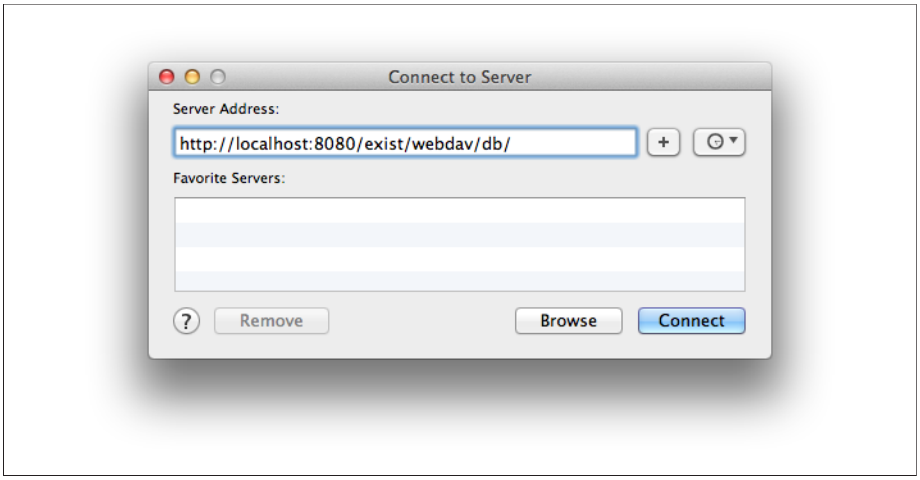


Figure 13-6. Mac OS X Finder: select the "Connect to Server" menu item

2. Next, you need to enter the URI of the eXist server and its WebDAV API. Typically this takes the form *http://<myserver>:8080/exist/webdav/db/*. If you are running eXist on your own Mac, you can replace *<myserver>* with *localhost*. Finally, click Connect. See Figure 13-7.

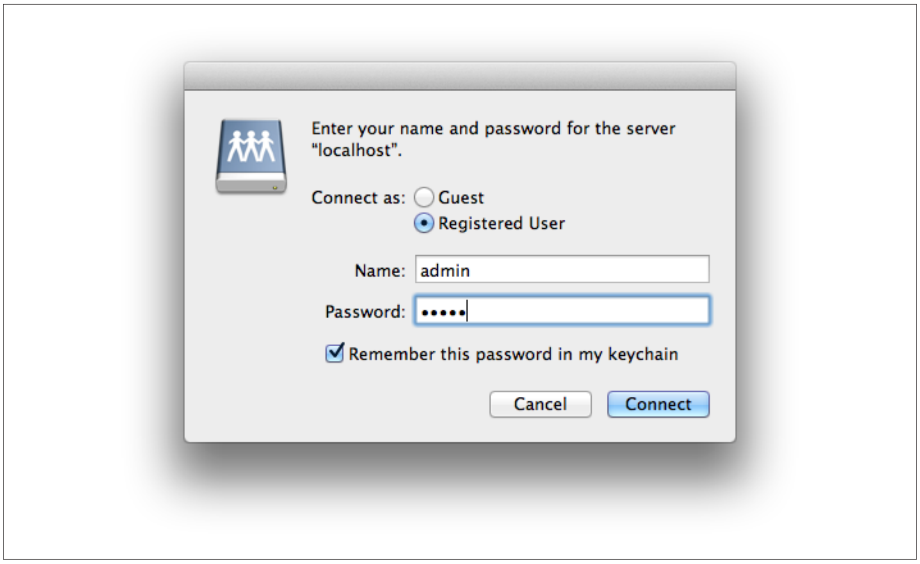


http://localhost:8080/exist/webdav/db/

Figure 13-7. Mac OS X Finder "Connect to Server" dialog

3. Finally, you need to provide the username and password of your eXist user account. If you have just set up eXist or will be the only user, you can use the

default built-in `admin` user and the password that you set during the installation of eXist. See Figure 13-8.


Enter your name and password for the server "localhost".

Connect as: Guest [✓ Registered User]

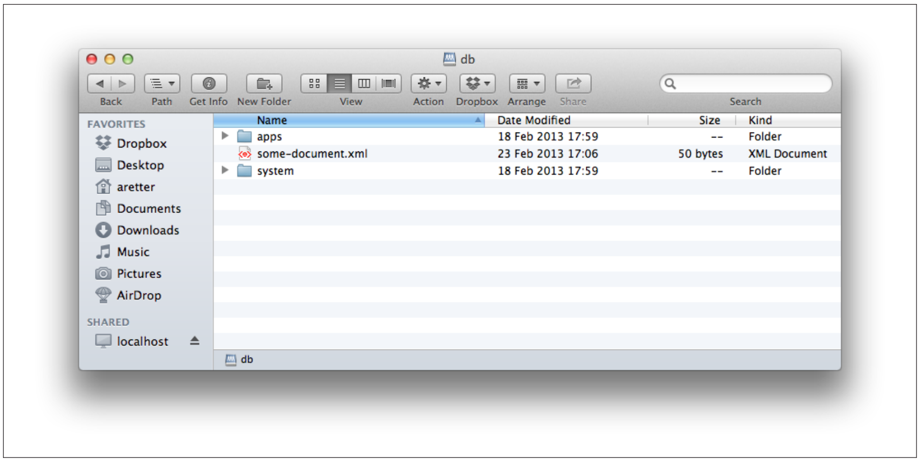
Name: admin

Password: •••••

[✓] Remember this password in my keychain

Figure 13-8. Connect using your eXist WebDAV username and password

4. You can now access the eXist database from the Finder and your other Mac applications just as if it were a networked filesystem. It appears in the Finder panel as *localhost* (or the name of your server) under the Shared items. You can now use any Mac application to read and write documents in eXist. You can also create/move/delete collections in this manner, as they appear to the Finder as regular folders. See Figure 13-9.



Name	Date Modified	Size	Kind
apps	18 Feb 2013 17:59	--	Folder
some-document.xml	23 Feb 2013 17:06	50 bytes	XML Document
system	18 Feb 2013 17:59	--	Folder

Figure 13-9. Finder with WebDAV connection to eXist

Using WebDAV from Linux

There are many different distributions of Linux available, some of which have GUI desktop environments and some of which do not. As covering them all would probably take a book in itself, we will cover just two approaches here that are suitable for a large proportion of users.

Using WebDAV from GNOME Nautilus. If you are using a GNOME 2– or GNOME 3–based Linux desktop environment such as CentOS, RHEL, Linux Mint, or Ubuntu, then you most likely have Nautilus or a derivative of it available to you. Nautilus, like Windows Explorer and the Mac OS X Finder, provides an easy mechanism for mounting WebDAV folders.

Follow these steps to mount eXist WebDAV from Nautilus:

1. First, locate the "Connect to Server" menu item in Nautilus (under the Places menu, as shown in Figure 13-10) and click it. In CentOS and RHEL (for our examples, we used CentOS 6.5), there is also a shortcut from the desktop menu.

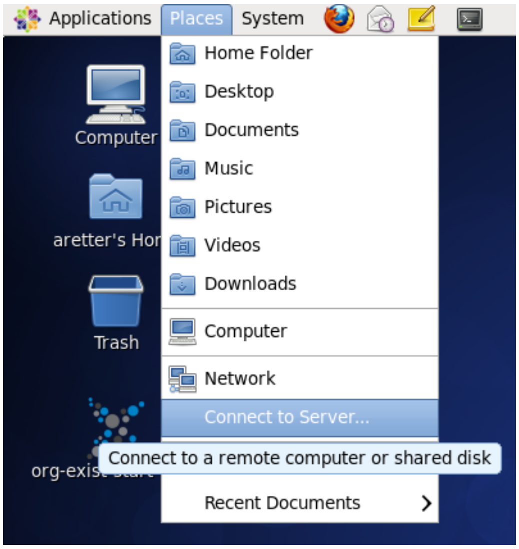
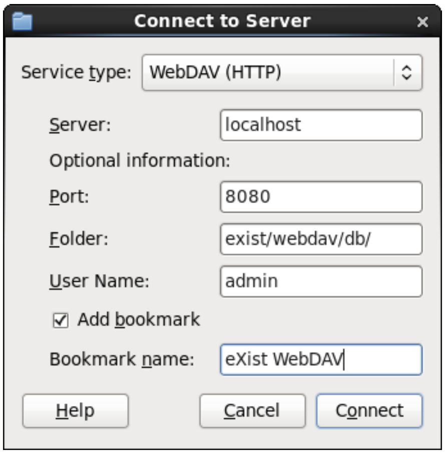


Figure 13-10. CentOS 6: select the "Connect to Server" menu item

2. Next, you need to enter the URI of the eXist server and its WebDAV API. Typically, this takes the form *http://<myserver>:8080/exist/webdav/db/*. If you are running eXist on your Linux PC, you can replace *<myserver>* with *localhost*. You will also need to provide a username. If you have just set up eXist or will be the only user, you can use the default built-in `admin` user. Finally, click Connect (see Figure 13-11).


Connect to Server

Service type: WebDAV (HTTP)

Server: localhost

Optional information:

Port: 8080

Folder: exist/webdav/db/

User Name: admin

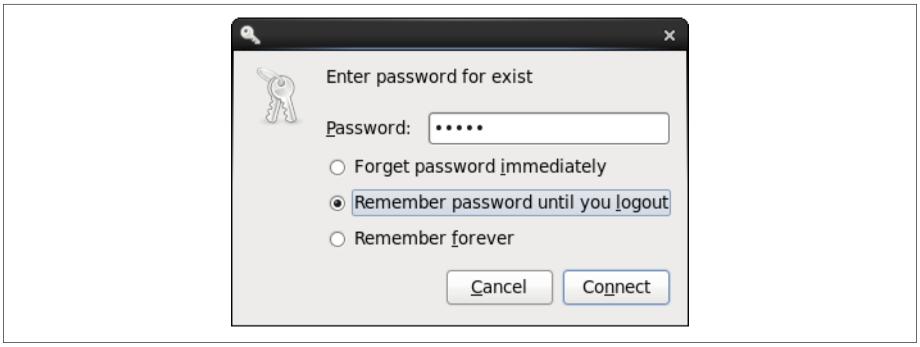
☑ Add bookmark

Bookmark name: eXist WebDAV

Figure 13-11. CentOS 6 "Connect to Server" dialog

You can also add a bookmark for the WebDAV connection if you wish; this will enable to you to reconnect easily in the future from your bookmarks.

3. Finally, you need to provide the password of your eXist user account. If you are using the built-in `admin` user and have just set up eXist, the password will be the same as the one that you set during the installation of eXist. See Figure 13-12.



Enter password for exist

Password: •••••

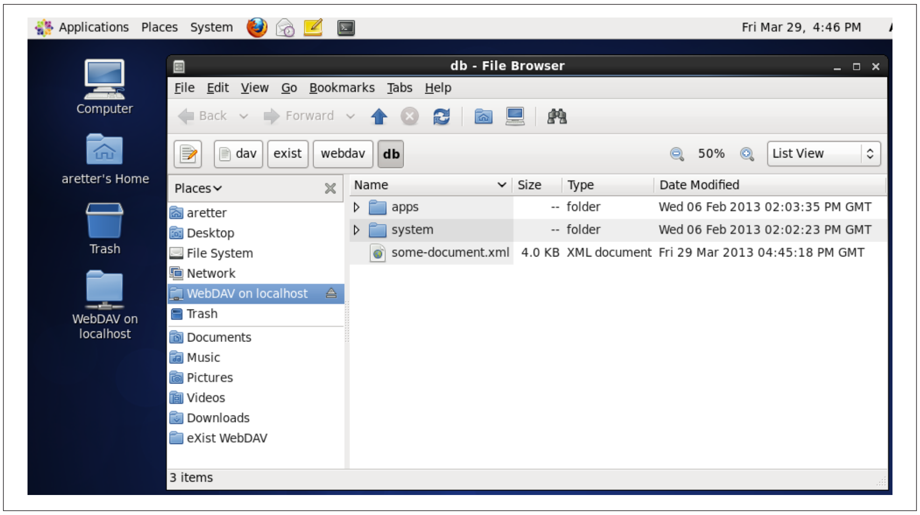
( ) Forget password immediately

(●) Remember password until you logout

( ) Remember forever

Figure 13-12. Connect using your eXist WebDAV password

4. You can now access the eXist database from Nautilus and your other GNOME applications just as if it were a networked filesystem. It appears in the Nautilus panel as WebDAV on *localhost* (or the name of your server) under the Places items. If you have a Places menu on your desktop, it will also appear there. You can now use any GNOME application to read and write documents in eXist. You can also create/move/delete collections in this manner, as they appear to Nautilus as regular folders. See Figure 13-13.



Name	Size	Type	Date Modified
apps	--	folder	Wed 06 Feb 2013 02:03:35 PM GMT
system	--	folder	Wed 06 Feb 2013 02:02:23 PM GMT
some-document.xml	4.0 KB	XML document	Fri 29 Mar 2013 04:45:18 PM GMT

Figure 13-13. CentOS 6: Nautilus with WebDAV connection to eXist

Using WebDAV with FUSE. If you do not have a GNOME desktop environment or want to be able to use eXist via WebDAV from non-GNOME applications, then another option is to use FUSE and davfs2 together. FUSE is typically installed already in most modern Linux distributions.

You can install davfs2 in Debian-based distributions (e.g., Ubuntu and Mint) by running the following from a terminal:

```
sudo apt-get install davfs2
```

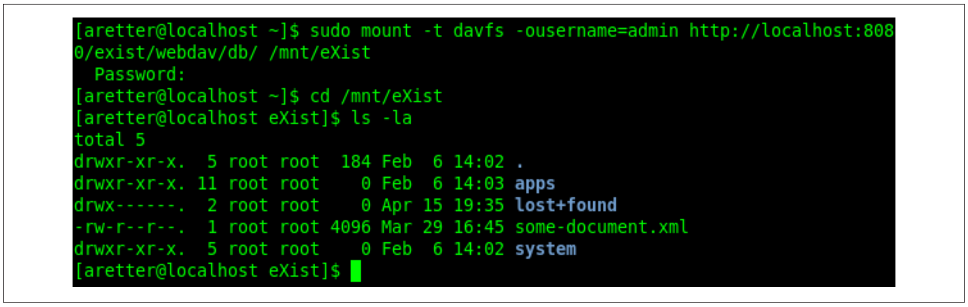
Likewise, you can install davfs2 in distributions with RPM package managers (e.g., RHEL, CentOS, SLES and openSUSE) via RPMForge by running the following from a terminal:

```
sudo rpm -Uhv
  http://pkgs.repoforge.org/rpmforge-release/
  rpmforge-release-0.5.3-1.el6.rf.x86_64.rpm
sudo yum install fuse-davfs2
```

The RPMForge release used here is for RHEL/CentOS 6 x64; you can find details of the correct RPMForge for your distribution at *http://repoforge.org/use/*.

Once you have davfs2 installed, you can mount the eXist WebDAV folder (as shown in Figure 13-14):

```
sudo mount -t davfs -ousername=admin
  http://localhost:8080/exist/webdav/db/ /mnt/eXist
```



```
[aretter@localhost ~]$ sudo mount -t davfs -ousername=admin http://localhost:808
0/exist/webdav/db/ /mnt/eXist
  Password:
[aretter@localhost ~]$ cd /mnt/eXist
[aretter@localhost eXist]$ ls -la
total 5
drwxr-xr-x.  5 root root  184 Feb  6 14:02 .
drwxr-xr-x. 11 root root    0 Feb  6 14:03 apps
drwx------.  2 root root    0 Apr 15 19:35 lost+found
-rw-r--r--.  1 root root 4096 Mar 29 16:45 some-document.xml
drwxr-xr-x.  5 root root    0 Feb  6 14:02 system
[aretter@localhost eXist]$
```

Figure 13-14. Linux davfs2 FUSE mount to eXist WebDAV

If the folder */mnt/eXist* does not exist on your system, you need to either create it or choose another empty folder to which you have access to act as the mount point.

You can now access the eXist database from any Linux application just as if it were a networked filesystem. You can use any Linux application to read and write documents in eXist. You can also create/move/delete collections in the same way as you would any other folder, as they appear to Linux as regular folders on a filesystem.

This is a very simple example, and you should be aware that davfs2 maintains a cache of file changes that is periodically flushed. In particular, the cache is flushed when the filesystem is unmounted, so you should aim to unmount the filesystem before shutting down the eXist server. davfs2 has many configuration options, so it's a good idea to consult the manpage (`man davfs2.conf`) if you plan on making serious use of this tool.

Using WebDAV from Java

There are many ways in which you could connect to eXist using WebDAV from Java, but unless you really want to spend all your time building a WebDAV client it is perhaps more pragmatic to use an existing library to assist you. There are several available libraries for Java that offer WebDAV client features, but here we'll look briefly at using the Milton client library to talk to eXist from Java. At the time of writing the latest version of Milton was version 2.4.2.5, and the version of Java used was 1.6.

There are just three main objects that you need to understand in the Milton client library to make WebDAV requests to eXist: `Hosts`, `Resources`, and `Paths`.

`Host`
: In Milton, the `Host` object holds all of the details needed to make connections to the WebDAV server. To access eXist, at minimum you will need to provide:

  `Server`
  : The hostname or IP address of the eXist server that you wish to connect to. If you are running your WebDAV client on the same machine as eXist, then you may use either `localhost` or `127.0.0.1`.

  `Port`
  : The TCP port that the eXist server you wish to connect to is listening on. If you have not reconfigured this setting in eXist, it will be `8080` by default.

  `RootPath`
  : The path on the eXist server to the WebDAV server endpoint. If you have not reconfigured this setting in eXist, it will be `exist/webdav/db` by default.

  `Username`
  : The username of a valid user account in eXist that you wish to connect to eXist as. If you have a newly installed eXist, you may use the `admin` account.

  `Password`
  : The password that accompanies the aforementioned username. If you are using the `admin` account of a newly installed eXist, the password will be whatever you defined during the installation, or otherwise the empty string.

  Milton provides a convenient `HostBuilder` class to help you construct your host (see Example 13-1).

  Example 13-1. Constructing a suitable Milton Host object for eXist

```
HostBuilder builder = new HostBuilder();
builder.setServer("localhost");
builder.setPort(8080);
builder.setRootPath("exist/webdav/db");
builder.setUser("admin");
builder.setPassword("my-admin-password ");

Host host = builder.buildHost();
```

`Resource`
: In Milton, the `Resource` object represents a resource on the WebDAV server. In Milton terms, this is one of the following:

  `Folder`
  : A *folder* resource in Milton is equivalent to a collection in eXist.

`File`
: A *file* resource in Milton is equivalent to a resource in eXist. Milton does not differentiate between XML files and binary files in eXist; to Milton they are all just files.

From a host we can retrieve resources, and we can use those resources to find subresources (see Example 13-2).

Example 13-2. Retrieving a resource from eXist using Milton

```
final Resource resource = host.child("my-collection");
if(resource != null) {
  if(resource instanceof Folder) {
    //resource is a Folder, i.e. collection in eXist
    final Folder folder = ((Folder)resource);

    //TODO you do something with the Folder here

  } else if(resource instanceof File) {
    //resource is a File, i.e. resource in eXist
    final File file = ((File)resource);

    //TODO you do something with the File here
  }
}
```

`Path`
: In Milton, the `Path` object encapsulates a path to a resource. The path may be either absolute from the root path, or relative to an existing resource.

We can construct paths that are relative to other paths and then execute operations relating to those paths (see Example 13-3).

Example 13-3. Creating the collection /db/my-collection/apples/pears in eXist with Milton

```
Path rootPath = host.path();
Path pathPears = Path.path(rootPath, "my-collection/apples/pears");
Folder pears = host.getOrCreateFolder(pathPears, true);
```

Examples. The source code of two small complete examples of using Milton from Java to store a file and retrieve a file is included in the folder *chapters/integration/webdav-client* of the *book-code* Git repository (see "Getting the Source Code" on page 15).

To compile the examples, enter the *webdav-client* folder and run `mvn package`.

You can then execute the StoreApp example like so:

```
java -jar webdav-client-store/target/webdav-client-store-1.0-example.jar
```

This shows the available arguments for using the StoreApp.

A complete example of using the application might look like the following, which would upload the file */tmp/large.xml* to the collection */db/my-new-collection* in eXist:

```
java -jar webdav-client-store/target/webdav-client-store-1.0-example.jar
  localhost 8080 /tmp/large.xml /db/my-new-collection admin
```

You can execute the RetrieveApp example like so:

```
java -jar webdav-client-retrieve/target/webdav-client-store-1.0-example.jar
```

This shows the available arguments for using the RetrieveApp.

A complete example of using the application might look like the following, which would download the resource */db/some-document.xml* to the file in the current directory named *some-document.xml*:

```
java -jar webdav-client-retrieve/target/webdav-client-retrieve-1.0-example.jar
  localhost 8080 /db/some-document.xml admin > some-document.xml
```

REST Server API

The REST Server in eXist offers a REST-like API that enables you to both manipulate the contents of the database and also send queries to be executed against the contents of the database. This section looks at the REST Server API in detail and also provides information for programmers who may like to integrate with eXist. If you are looking to get started with the REST Server, then you should first read "Querying the Database Using REST" on page 94.

In addition, and perhaps more interestingly, the REST Server API allows you to prestore XQuery (and XProc) resources into the database and then execute them by calling them by URI. The entire HTTP request and response are made available to your XQuery, enabling you to determine processing dynamically in your XQuery based on parameters of the HTTP request and create your own HTTP response. This mechanism allows you to build complete and versatile web applications in XQuery; see "Executing stored queries" on page 335 and Chapter 9 for further details. For a complete illustration of the operations provided by the REST Server, see Appendix B.

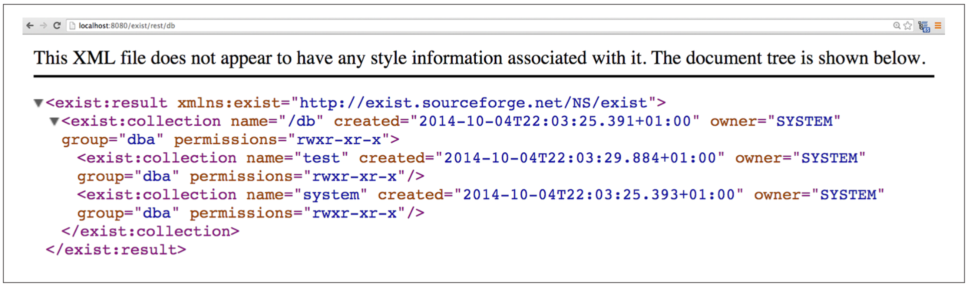
There are many tools, programming languages, and libraries that allow you to interact with a REST API (including web browsers, to a limited extent), but in these examples we will show you how to use cURL. We'll also provide some simple examples in Java in "Using the REST Server API from Java" on page 339.

Retrieving collections and documents

The base URI of the REST Server in eXist on a default installation is *http://localhost:8080/exist/rest/db*.

The */db* on the end of the URI indicates the root collection in the database. When the REST Server receives an HTTP `GET` request (e.g., a typical request from a web browser such as Firefox or Chrome) for a collection URI in the database, it will by default produce a listing of the resources and subcollections in that collection in XML, as seen in Figure 13-15.

You can disable the collection listing provided by the REST Server; see "Disabling direct access to the REST Server" on page 180.


localhost:8080/exist/rest/db

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<exist:result xmlns:exist="http://exist.sourceforge.net/NS/exist">
  ▼<exist:collection name="/db" created="2014-10-04T22:03:25.391+01:00" owner="SYSTEM"
   group="dba" permissions="rwxr-xr-x">
     <exist:collection name="test" created="2014-10-04T22:03:29.884+01:00" owner="SYSTEM"
     group="dba" permissions="rwxr-xr-x"/>
     <exist:collection name="system" created="2014-10-04T22:03:25.393+01:00" owner="SYSTEM"
     group="dba" permissions="rwxr-xr-x"/>
   </exist:collection>
 </exist:result>
```

Figure 13-15. Browsing the REST Server API with the Chrome web browser

When accessing a collection from the REST Server, rather than listing the collection contents, you can present a document (or anything, really) instead by executing an XQuery known as a *default document*. You can enable default documents by adding mappings in *$EXIST_HOME/descriptor.xml* and then restarting eXist.

For example, if you wanted to remove the default collection listing response provided by the REST Server when accessing the collection */db/products*, you could add the following mapping to the maps section of *$EXIST_HOME/descriptor.xml*:

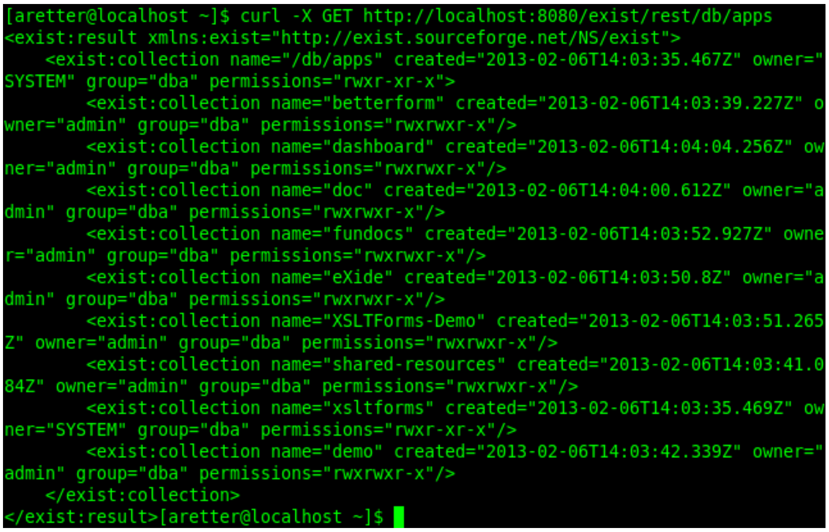
```
<map path="/db/products" view="/db/products/default.xq"/>
```

then, instead of the collection listing, the result of executing the XQuery */db/products/default.xq* would be returned.

Of course, manipulating the *descriptor.xml* configuration file is not the only solution available, but it is simple. If you require something more complex, you can make use of the full power of eXist's XQuery URL rewriting (see "URL Mapping Using URL Rewriting" on page 194).

You can use cURL to make a `GET` request to eXist, by specifying `-X GET` to the `curl` command and then the URI. For example, the following cURL command would return a listing of the */db/apps* collection in the database, as shown in Figure 13-16:

```
curl -X GET http://localhost:8080/exist/rest/db/apps
```



```
[aretter@localhost ~]$ curl -X GET http://localhost:8080/exist/rest/db/apps
<exist:result xmlns:exist="http://exist.sourceforge.net/NS/exist">
    <exist:collection name="/db/apps" created="2013-02-06T14:03:35.467Z" owner="
SYSTEM" group="dba" permissions="rwxr-xr-x">
        <exist:collection name="betterform" created="2013-02-06T14:03:39.227Z" o
wner="admin" group="dba" permissions="rwxrwxr-x"/>
        <exist:collection name="dashboard" created="2013-02-06T14:04:04.256Z" ow
ner="admin" group="dba" permissions="rwxrwxr-x"/>
        <exist:collection name="doc" created="2013-02-06T14:04:00.612Z" owner="a
dmin" group="dba" permissions="rwxrwxr-x"/>
        <exist:collection name="fundocs" created="2013-02-06T14:03:52.927Z" owne
r="admin" group="dba" permissions="rwxrwxr-x"/>
        <exist:collection name="eXide" created="2013-02-06T14:03:50.8Z" owner="a
dmin" group="dba" permissions="rwxrwxr-x"/>
        <exist:collection name="XSLTForms-Demo" created="2013-02-06T14:03:51.265
Z" owner="admin" group="dba" permissions="rwxrwxr-x"/>
        <exist:collection name="shared-resources" created="2013-02-06T14:03:41.0
84Z" owner="admin" group="dba" permissions="rwxrwxr-x"/>
        <exist:collection name="xsltforms" created="2013-02-06T14:03:35.469Z" ow
ner="SYSTEM" group="dba" permissions="rwxr-xr-x"/>
        <exist:collection name="demo" created="2013-02-06T14:03:42.339Z" owner="
admin" group="dba" permissions="rwxrwxr-x"/>
    </exist:collection>
</exist:result>[aretter@localhost ~]$
```

Figure 13-16. Browsing the REST Server API with cURL

The default HTTP request method in cURL is `GET`, so you can actually omit the `-X GET` parameter for conciseness if you wish.

Just as with retrieving a collection, you can retrieve the content of a resource in the database by appending its name to the collection in the request URI. For example, the following cURL command retrieves the resource *some-document.xml* from the collection */db*:

```
curl http://localhost:8080/exist/rest/db/some-document.xml
```

If you wish to redirect the output from cURL to a file, you can use the `-o` argument (e.g., `-o myfile.xml`). Also, if you wish to see the HTTP request and response details as well as the content of the response, you may also provide the `-v` argument to cURL for verbose output.

XSL transformation. The *serializer* used by the REST Server also processes any *XSL processing instructions* declared in an XML document before returning the result to you. You can control this behavior by appending the *_xsl* parameter in the query part of the URL. You can also exploit this parameter to specify your own stylesheet at call time, as Table 13-1 shows.

Table 13-1. _xsl query parameters

XSL parameter value	Explanation
`no`	Disables the processing of XSL processing instructions.
`yes`	Enables the processing of XSL processing instructions.
`uri`	You may provide a URI to an XSL stylesheet that you wish to apply. The URI can be a database URI (e.g., `/db/my-stylesheet.xslt`).

The default behavior of whether XSL processing instructions are processed or not by the serializer is configurable in *$EXIST_HOME/conf.xml* at the attribute indicated by the XPath */exist/serializer/@enable-xsl*.

For example, to apply the XSL transformation at */db/my-stylesheet.xslt* when retrieving */db/some-document.xml*, you could use the following cURL command:

```
curl http://localhost:8080/exist/rest/db/some-document.xml
?_xsl=/db/my-stylesheet.xslt
```

If you just want to know the size of a document, or when a document or collection was last modified, you can use the `HEAD` method instead of `GET`. This returns just some basic metadata in the HTTP response headers instead of the resource content or collection listing.

Storing a document

You may store an XML or binary document into a collection in eXist via the REST Server API, by submitting the content of the document you wish to store as the body of a `PUT` request. You should also specify the Internet media type in the HTTP `Content-Type` header of your request. If the collection you are `PUT`ing the document into does not allow execute and write access by *other* users, then you will also need to provide a username and password for an account that does have such access.

For example, the following cURL command will store the XML document */tmp/my-doc.xml* into the collection */db/docs/personal*:

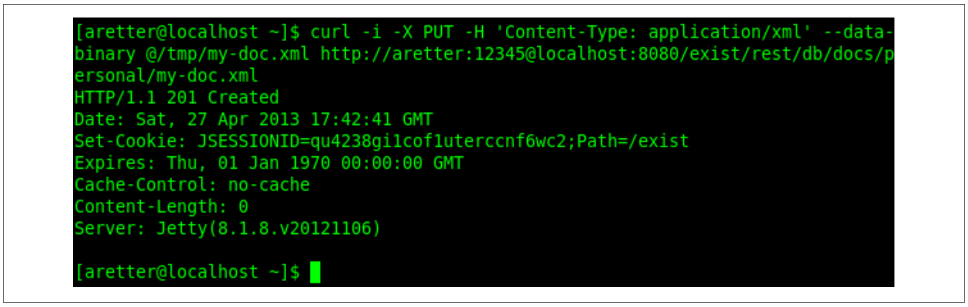
```
curl -i -X PUT
  -H 'Content-Type: application/xml'
  --data-binary @/tmp/my-doc.xml
  http://aretter:12345@localhost:8080/exist/rest/db/docs/personal/my-doc.xml
```

Its parameters are explained in Table 13-2 and its output illustrated in Figure 13-17.

Table 13-2. cURL parameters for storing a document

cURL parameters	Explanation
`-i`	PUTing a document into eXist does not return any content. You will know if the `PUT` succeeds by examining the HTTP response code; success is indicated by the code `201 Created`. The `-i` parameter causes cURL to show the HTTP response headers (including the HTTP response code).
`-X PUT`	The `-X` parameter allows you to specify the HTTP request method. In this example the method of the request is `PUT`, as we want to `PUT` the new resource in the database.
`-H 'Content-Type: application/xml'`	eXist needs to know the Internet media type of the resource you are `PUT`ing so it can store it correctly. The `-H` parameter allows you to specify an HTTP request header. We can inform eXist of the Internet media type by setting the `Content-Type` header. In this example, we use the Internet media type for an XML document.
`--data-binary @/tmp/my-doc.xml`	This parameter allows you to send binary data in the body of the request. In this example we want to send an XML document; the `@` indicates that the data should be read from the file */tmp/my-doc.xml*.
`http://aretter:12345@localhost:8080/exist/rest/db/docs/personal/my-doc.xml`	The final parameter is always the target URI of the request. In this instance, we are making the request on the */db/docs/personal* collection, where we want to store the data into a resource named *my-doc.xml*. We also specify the username `aretter` and password `12345` of an account in eXist that has execute and write access to store the document.[a]

[a]If you do not have execute and write access to store the resource indicated by the URI of the request, you will receive an HTTP response code of `401 Unauthorized`.



```
[aretter@localhost ~]$ curl -i -X PUT -H 'Content-Type: application/xml' --data-
binary @/tmp/my-doc.xml http://aretter:12345@localhost:8080/exist/rest/db/docs/p
ersonal/my-doc.xml
HTTP/1.1 201 Created
Date: Sat, 27 Apr 2013 17:42:41 GMT
Set-Cookie: JSESSIONID=qu4238gi1coflutercnf6wc2;Path=/exist
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Cache-Control: no-cache
Content-Length: 0
Server: Jetty(8.1.8.v20121106)

[aretter@localhost ~]$
```

Figure 13-17. Storing a document via the REST Server API with cURL

If you specify a collection that does not exist, eXist will automatically create the necessary collection hierarchy for you.

Once the document is stored, you can retrieve it by doing an HTTP `GET` on the resource URI; for example:

```
curl http://localhost:8080/exist/rest/db/docs/personal/my-doc.xml
```

Likewise, if you wish to see it listed in its collection, you can retrieve the collection's contents by doing an HTTP `GET` on the collection URI; for example:

```
curl http://localhost:8080/exist/rest/db/docs/personal
```

Deleting collections and documents

You may delete collections and documents in eXist via the REST Server API, by submitting `DELETE` requests whose URIs indicate the collections or documents you wish to remove. If the parent collection of the document or collection you are `DELETE`ing does not allow execute and write access by other users, then you will also need to provide a username and password for an account that does have execute and write access.

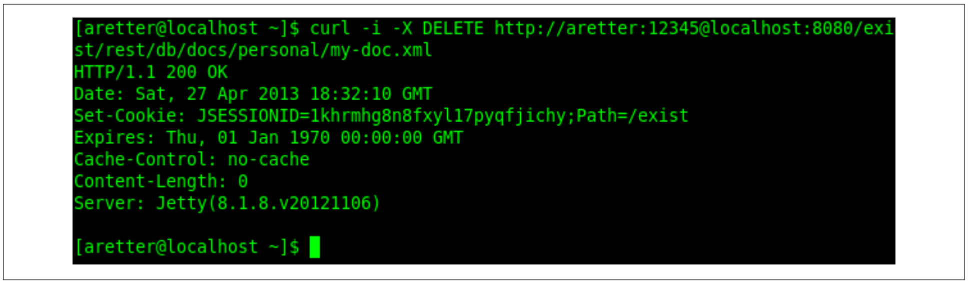
For example, the following cURL command will delete the XML document *my-doc.xml* from the collection */db/docs/personal*:

```
curl -i -X DELETE
  http://aretter:12345@localhost:8080/exist/rest/db/docs/personal/my-doc.xml
```

Its parameters are explained in Table 13-3 and its output illustrated in Figure 13-18.

Table 13-3. cURL parameters for deleting a document

cURL parameters	Explanation
`-i`	`DELETE`ing a document from eXist does not return any content. You will know if the `DELETE` succeeds by examining the HTTP response code; success is indicated by the code `200 OK`. The `-i` parameter causes cURL to show the HTTP response headers (including the HTTP response code).
`-X DELETE`	The `-X` parameter allows you to specify the HTTP request method. In this example, the method of the request is `DELETE`, as we want to `DELETE` the resource or collection from the database.
`http://aretter:12345@localhost:8080/exist/rest/db/docs/personal/my-doc.xml`	The final parameter is always the URI of the request. In this instance, we are making the request on the */db/docs/personal* collection, where we want to delete the resource named *my-doc.xml*. We also specify the username (`aretter`) and password (`12345`) of an account in eXist that has execute and write access on the collection, so we are able to delete the document.



```
[aretter@localhost ~]$ curl -i -X DELETE http://aretter:12345@localhost:8080/exi
st/rest/db/docs/personal/my-doc.xml
HTTP/1.1 200 OK
Date: Sat, 27 Apr 2013 18:32:10 GMT
Set-Cookie: JSESSIONID=1khrmhg8n8fxyl17pyqfjichy;Path=/exist
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Cache-Control: no-cache
Content-Length: 0
Server: Jetty(8.1.8.v20121106)

[aretter@localhost ~]$ 
```

Figure 13-18. Deleting a document via the REST Server API with cURL

The mechanism for deleting a collection is exactly the same as that for deleting a document, except the URI should indicate the collection path in the database and not the document path. For example, the following cURL command will delete the collection *personal* from the parent collection */db/docs*:

```
curl -i -X DELETE http://aretter:12345@localhost:8080/exist/rest/db/docs/personal
```

If you delete a collection, you remove the collection *and* all documents within it.

Querying the database

There are two approaches for sending XQueries to the REST Server API to be executed against the database: HTTP `GET` and HTTP `POST`. Both approaches offer very similar functionality and results; however:

- HTTP `GET` is most suitable for small and short XQuery or XPath expressions. You may send these expressions in the query part of the URL using the `_query` parameter. The path part of the URI indicates the context upon which to query the database (i.e., a collection or document), unless the context is set manually in XQuery through the `fn:doc` or `fn:collection` functions.
- HTTP `POST` is suitable for XQuery main modules. You may send the main module inside an XML document that describes the query in the body of the request.

Imagine that you have a collection of XML documents (*/db/people*) in eXist that contain details about people. Each document in the collection represents a single person, and among other things contains that person's name and date of birth (see Example 13-4).

Example 13-4. XML document for a person

```
<?xml version="1.0" encoding="UTF-8"?>
<person>
    <name>
        <first-name>John</first-name>
        <family-name>Smith</family-name>
    </name>
    <born>
        <date>1974-05-16</date>
        <location>
            <settlement>Tiverton</settlement>
            <country>United Kingdom</country>
        </location>
    </born>
    <residence>
        <location>
            <address-line>123 High Street</address-line>
            <settlement>Cullompton</settlement>
            <county>Devon</county>
            <country>United Kindom</country>
        </location>
    </residence>
    <contact>
        <telephone type="mobile">+44 7777 123456</telephone>
        <email>john.smith@johnsmith.com</email>
    </contact>
</person>
```

Let's look at how we would send an XQuery to the REST Server API to retrieve the names of all the people in the collection. Our XQuery might look like Example 13-5.

Example 13-5. XQuery to retrieve names of all people in the collection

```
xquery version "1.0";

/person/name
```

HTTP GET queries. To send the XQuery in Example 13-5 to the REST Server API using the simple HTTP `GET` approach, we can ignore the XQuery version declaration, as eXist will default to XQuery 1.0. However, as we are going to place the XQuery into the `_query` parameter in the URL, we should first URL-encode the XQuery to escape any URL-sensitive characters. If you are doing these operations from a programming language, there is most likely a library function already available for URL encoding; otherwise, if you are using cURL or sending the queries manually, you can use a simple URL encoder like URL Encode/Decode.

Our URL-encoded XQuery becomes:

```
%2Fperson%2Fname
```

We can now send this XQuery to the REST Server API using the following cURL command:

```
curl "http://localhost:8080/exist/rest/db/people?_query=%2Fperson%2Fname"
```

which could result in a response similar to:

```
<exist:result xmlns:exist="http://exist.sourceforge.net/NS/exist"
    exist:hits="3" exist:start="1" exist:count="3">
  <name>
    <first-name>John</first-name>
    <family-name>Smith</family-name>
  </name>
  <name>
    <first-name>George</first-name>
    <family-name>Baker</family-name>
  </name>
  <name>
    <first-name>Barbara</first-name>
    <family-name>Jones</family-name>
  </name>
</exist:result>
```

By default the REST Server API *wraps* the result of our XQuery in an `exist:result` element; this provides us with a container for our data and some metadata about the number of results found by the query (`exist:hits`) and the number of results immediately returned (`exist:start` and `exist:count`). In this case, we can see that the query found 3 hits in total, and returned the results starting from index 1 and counting up to index 3. The use of start and count should become clearer when we look at paging shortly. The format of the XML result wrapper is documented in "wrap XML grammar" on page 540.

If you do not want eXist to return your data in an exist:result element, you can turn off wrapping using _wrap=no, as in the following cURL command:

```
curl "http://localhost:8080/exist/rest/db/people
  ?_wrap=no&_query=%2Fperson%2Fname"
```

However, you should note that the XQuery we wrote will return a *sequence* of name elements, so the result of the call to the REST Server API will not be a valid XML document if you turn off wrapping. To resolve this, you could introduce a wrapper element in your own XQuery, as in:

```
xquery version "1.0";

<names>{
    /person/name
}</names>
```

After URL encoding, we can now send this XQuery to the REST Server API using the following cURL command:

```
curl "http://localhost:8080/exist/rest/db/people
  ?_wrap=no&_query=%3Cnames%3E%7B%0A++++%2Fperson%2Fname%0A%7D%3C%2Fnames%3E"
```

which could result in a response similar to:

```
<names>
  <name>
    <first-name>John</first-name>
    <family-name>Smith</family-name>
  </name>
  <name>
    <first-name>George</first-name>
    <family-name>Baker</family-name>
  </name>
  <name>
    <first-name>Barbara</first-name>
    <family-name>Jones</family-name>
  </name>
</names>
```

So far, we have just sent very simple queries to the REST Server. While placing the XQuery in the query parameter of the URL sent to the REST Server API works for small XQueries, it does not scale particularly well because:

- We have to URL-encode the XQuery that we wish to send, which makes it unreadable.

- The URL string becomes longer as the XQuery becomes longer, and URL encoding compounds this problem. Some HTTP clients and servers have limitations on the length of the URLs they can handle!

A better approach than using HTTP GET queries for anything more than the simplest XQueries is to use HTTP POST queries instead. The main advantage of HTTP GET queries is that you can easily send them from any web browser's address bar, and this advantage is negated as the queries get more complex. That said, there are plug-ins for several browsers that enable you to send more complex requests, such as Postman for Google Chrome and HttpRequester for Mozilla Firefox.

HTTP POST queries. When sending XQueries via HTTP POST to the REST Server API, we need to place them in an XML document that contains the XQuery and any parameters for the REST Server or XQuery itself. Let's look at how we would send our simple query using HTTP POST:

```
<query xmlns="http://exist.sourceforge.net/NS/exist">
    <text>
        <![CDATA[

xquery version "1.0";

/person/name

        ]]>
    </text>
</query>
```

We place the XQuery itself inside a CDATA section so as to avoid having to escape any XML-sensitive characters in our XQuery. We can now POST the XML document containing the XQuery to the REST Server API using the following cURL command:

```
curl -X POST -H 'Content-Type: application/xml'
  --data-binary @/tmp/person-name.xml
  http://localhost:8080/exist/rest/db/people
```

The result of this query is exactly the same as that of the equivalent HTTP GET example earlier, but it is much easier to send larger queries using HTTP POST. Table 13-4 explains the cURL parameters used here.

Table 13-4. cURL parameters for sample HTTP POST query

cURL parameters	Explanation
`-X POST`	The `-X` parameter allows you to specify the HTTP request method. In this example the method of the request is `POST`, as we want to `POST` the XML document containing the XQuery to the REST Server.
`-H 'Content-Type: application/xml'`	eXist needs to know the Internet media type of the resource you are `POST`ing so it can decide how to process it. The `-H` parameter allows you to specify an HTTP request header, and we can inform eXist of the Internet media type by setting the `Content-Type` header. In this example, because the XQuery is embedded in an XML document, we use the Internet media type for an XML document.
`--data-binary @/tmp/person-name.xml`	The `--data-binary` parameter allows you to send binary data in the *body* of the request. In this example we want to send an XML document; the `@` indicates that the data should be read from the file */tmp/person-name.xml*.
`http://localhost:8080/exist/rest/db/people`	The final parameter is always the URI of the request. In this instance, we are setting the context of the XQuery as the collection */db/people*.

Now, let's also look at how we would construct a version of our simple query where the results are not wrapped by the REST Server using HTTP `POST`. To achieve this, we simply add the same `wrap` parameter as before, but this time implemented as an attribute to the `query` element:

```
<query xmlns="http://exist.sourceforge.net/NS/exist" wrap="no">
    <text>
        <![CDATA[

xquery version "1.0";

<names>{
    /person/name
}</names>

        ]]>
    </text>
</query>
```

REST Server parameters and paging results. So far we have looked at just the `query` and `wrap` parameters available in the REST Server API for queries sent via either HTTP `GET` or HTTP `POST`. There are several other parameters available—which are all documented in “REST Server Parameters” on page 531—but a common use case is to be able to break the results of your query into smaller pages of results, so we will examine how to achieve that here.

The REST Server provides a mechanism whereby you can send it an XQuery and have it *cache* the results of that query. It will provide you with a *session identifier*, which you can then use in subsequent requests to pull back subsets of those results (i.e., pages).

For this example, let’s imagine that we have added many more documents about people to our */db/people* collection, and that this time we wish to find *the average age of people in each settlement*. We know that there will be lots of results as our people live all over the world, so we want to return the results *ordered by age ascending*; more importantly, however, so as not to overwhelm the end user we want to present the results in pages of *10 results at a time*.

Let’s consider the XQuery that we might wish to `POST` to the REST Server API to achieve this. Apart from it being a more complex XQuery, note that the `cache="yes"`, `start="1"`, and `max="10"` attributes are set on the `query` element. The `cache` attribute instructs the REST Server to return a session identifier for the result set generated by the query. In addition, `start` instructs the REST Server to return results from the cached set starting at position `1`, and `max` instructs the server to return up to a maximum of `10` results from the cached set:

```
<?xml version="1.0" encoding="UTF-8"?>
<query xmlns="http://exist.sourceforge.net/NS/exist"
    cache="yes" start="1" max="10">
    <text>
        <![CDATA[
xquery version "1.0";

for $settlement in distinct-values(/person/residence/location/settlement)
let $average-age := avg(
    /person[residence/location/settlement eq $settlement]/born/(year-from-date(
    current-date()) - year-from-date(xs:date(./date))))
order by $average-age ascending
return

    <settlement>
        <name>{$settlement}</name>
        <average-age>{$average-age}</average-age>
    </settlement>

        ]]>
    </text>
</query>
```

We send our more complex query to the REST Server API in exactly the same way as our simpler query, using this cURL command:

```
curl -X POST -H 'Content-Type: application/xml'
  --data-binary @/tmp/settlement-average-name.xml
  http://localhost:8080/exist/rest/db/people
```

which could result in a response that starts with the element:

```
<exist:result xmlns:exist="http://exist.sourceforge.net/NS/exist"
exist:hits="567" exist:start="1" exist:count="10" exist:session="23">
```

We have omitted the entire response body for brevity, but the important thing to note here is that the REST Server has executed our `POST`ed XQuery and found `567` results (`hits` attribute), and it is returning the first `10` results (indicated by the `start` and `count` attributes). In addition, the results have been cached and will be accessible in the future using the session identifier `23` (`session` attribute).

So, we have returned our first page of `10` results, but how do we get our second page of results? We send almost the same request as before, but this time on the `query` element we want to set the `session` attribute to the session identifier that we were given by the response of the first request and increase the value of the `start` attribute, so we end up with the following request:

```
<?xml version="1.0" encoding="UTF-8"?>
<query xmlns="http://exist.sourceforge.net/NS/exist"
    session="23" cache="yes" start="11" max="10">
    <text>
        <![CDATA[
xquery version "1.0";

for $settlement in distinct-values(/person/residence/location/settlement)

let $average-age := avg(
    /person[residence/location/settlement eq $settlement]/born/(year-from-date(
    current-date()) - year-from-date(xs:date(./date))))
order by $average-age ascending

    return

    <settlement>
        <name>{$settlement}</name>
        <average-age>{$average-age}</average-age>
    </settlement>

        ]]>
    </text>
</query>
```

which we send to the REST Server by:

```
curl -X POST -H 'Content-Type: application/xml' --data-binary
@/tmp/settlement-average-name.page2.xml
http://localhost:8080/exist/rest/db/people
```

This results in a response that starts with the element:

```
<exist:result xmlns:exist="http://exist.sourceforge.net/NS/exist"
exist:hits="567" exist:start="11" exist:count="10" exist:session="23">
```

Again, we have omitted the entire response body for brevity, but the important thing to note here is that the REST Server is now returning the second page of results—that is, `10` results (`count` attribute) starting from position `11` (`start` attribute) in the cached result set.

We include the actual query in each request we send, in case the cached query result set has expired and the query has to be recomputed. This way, we know that we will always get the response, whether it's served from the cache or calculated. To retrieve further pages, simply repeat the second query, adjusting the `start` attribute each time.

Updating the database

You may update nodes within documents in eXist via the REST Server API, by `POST`ing XUpdate documents to the URI that indicates the collection or document context within the database to update. In addition, you may manually specify individual operations against other documents or collections in your XUpdate documents by using the XQuery `fn:doc` or `fn:collection` functions. If the document you are updating does not allow write access by *other* users, then you will also need to provide a username and password for an account that does have write access.

Let's look at how we would apply an XUpdate document to the document */db/some-document.xml* in eXist via the REST Server API:

```
<hello/>
```

This XUpdate document will insert an element called `name` with the text `Adam` into each `hello` element that it finds:

```
<?xml version="1.0" encoding="UTF-8"?>
<xupdate:modifications version="1.0"
  xmlns:xupdate="http://www.xmldb.org/xupdate">
    <xupdate:append select="/hello">
        <name>Adam</name>
    </xupdate:append>
</xupdate:modifications>
```

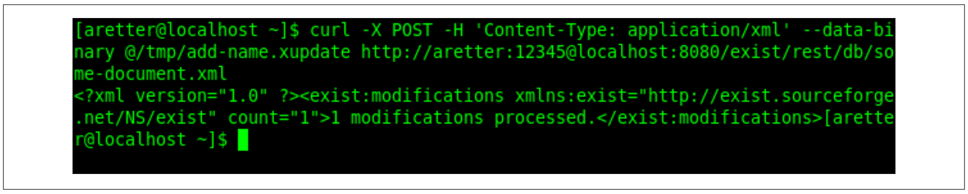
The following cURL command would apply the XUpdate document stored at */tmp/add-name.xupdate* to the XML document */db/some-document.xml* in eXist:

```
curl -X POST -H 'Content-Type: application/xml'
  --data-binary @/tmp/add-name.xupdate
  http://aretter:12345@localhost:8080/exist/rest/db/some-document.xml
```

Its parameters are explained in Table 13-5, and Figures 13-19 and 13-20 show how to perform and confirm the action, respectively.

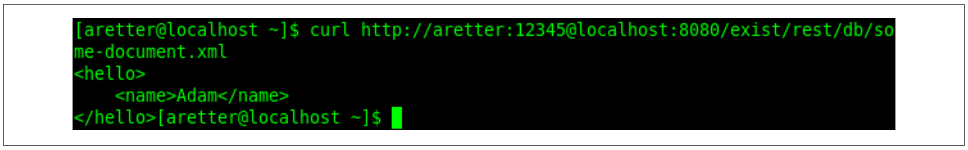
Table 13-5. cURL parameters for HTTP POST XUpdate

cURL parameters	Explanation
`-X POST`	The `-X` parameter allows you to specify the HTTP request method. In this example the method of the request is `POST`, as we want to `POST` the XUpdate document to the REST Server.
`-H 'Content-Type: application/xml'`	eXist needs to know the Internet media type of the resource you are `POST`ing so it can decide how to process it. The -H parameter allows you to specify an *HTTP request header*, and we can inform eXist of the Internet media type by setting the `Content-Type` header. In this example, because XUpdate is XML, we use the Internet media type for an XML document.
`--data-binary @/tmp/add-name.xupdate`	The `--data-binary` parameter allows you to send binary data in the *body* of the request. In this example we want to send an XUpdate document; the `@` indicates that the data should be read from the file */tmp/add-name.xupdate*.
`http://aretter:12345@localhost:8080/exist/rest/db/some-document.xml`	The final parameter is always the URI of the request. In this instance, we are processing the XUpdate against the document */db/some-document.xml*. We also specify the username (`aretter`) and password (`12345`) of an account in eXist that has write access to modify the document.



```
[aretter@localhost ~]$ curl -X POST -H 'Content-Type: application/xml' --data-bi
nary @/tmp/add-name.xupdate http://aretter:12345@localhost:8080/exist/rest/db/so
me-document.xml
<?xml version="1.0" ?><exist:modifications xmlns:exist="http://exist.sourceforge
.net/NS/exist" count="1">1 modifications processed.</exist:modifications>[arette
r@localhost ~]$
```

Figure 13-19. XUpdating a document via the REST Server API with cURL



```
[aretter@localhost ~]$ curl http://aretter:12345@localhost:8080/exist/rest/db/so
me-document.xml
<hello>
    <name>Adam</name>
</hello>[aretter@localhost ~]$
```

Figure 13-20. Retrieving a document after XUpdate via the REST Server API with cURL

For further information about XUpdate, see the XUpdate 1.0 Specification and "XUpdate" on page 105.

XUpdate is just one mechanism for updating documents in eXist. As an alternative, you can make use of the XQuery update extension in your XQueries (see "eXist's XQuery Update Extension" on page 102), which of course may be sent to the REST Server or invoked by the REST Server as stored queries, as described next.

Executing stored queries

Perhaps the most interesting and flexible feature of the REST Server API is that it allows you to invoke stored XQuery and XProc by making HTTP requests. This means that you can potentially write complex XQueries split across several main and library modules, store them into the database, and have them react to requests made to the REST Server API. This facility, coupled with eXist's extensions for XQuery, enables you to easily build your own HTTP/REST APIs in XQuery, or even entire web applications.

When working with stored queries and the REST Server API, it is very likely that you will want to use at least the `request` and `response` XQuery extension modules in your XQueries to work with the HTTP request and response. You can find more details on these in "The request Extension Module" on page 209 and "The response Extension Module" on page 211, respectively. Building web applications using this approach (among others) is discussed in Chapter 9, but for the purposes of integration we will demonstrate a simple example here.

Supplied alongside this chapter is the XQuery file *chapters/integration/rest-stored-query/image-api.xq* in the *book-code* Git repository (see "Getting the Source Code" on page 15) that, when stored into the database and subsequently called via the REST Server API, will deliver a simple custom REST API for manipulating images. To use the XQuery as well as store it into the database (for example, in the */db* collection), you also need to ensure: 1) that the *image-api.xq* file has execute access within the database by the calling user so that it may be executed; 2) that the collection */db/images* exists and is writable by the calling user; and 3) that the `image` XQuery extension module is enabled in *$EXIST_HOME/conf.xml*. The custom REST API provides the following three image manipulation functions:

- Store a JPEG image received over HTTP into the database.
- Retrieve a stored image from the database.
- Retrieve a thumbnail representation of an image from the database.

Let's now look at the XQuery code in detail, and how it performs each of these functions.

Store a JPEG image received over HTTP into the database. The API provided by the *image-api.xq* file allows you to send an HTTP `POST` to it via the REST Server API to store a JPEG image. In your HTTP request, if you set the `Content-Type` to `image/jpeg` and include the content of a JPEG image in the body of the request, it will be stored into the database and *image-api.xq* will return a `Location` and `identifier` in the HTTP response for the image.

When you make the following request with cURL:

```
curl -i -X POST -H 'Content-Type: image/jpeg' -data-binary @/tmp/cats.jpg
http://localhost:8080/exist/rest/db/image-api.xq
```

the code in our *image-api.xq* stored query handles it as follows:

```
if(request:get-method() eq "POST")then ❶
  if(request:get-header("Content-Type") eq "image/jpeg")then ❷
      let $db-path := local:store-image(request:get-data()) ❸,
      $uri-to-resource := concat(
          request:get-uri(),
          substring-after($db-path, $local:image-collection)) ❹

      return
      (
          response:set-status-code($response:CREATED) ❺,
          response:set-header("Location", $uri-to-resource) ❻,
          <identifier>{
              substring-after($db-path, concat($local:image-collection, "/"))
          }</identifier> ❼
      )
  else
      response:set-status-code($response:BAD-REQUEST) ❽
```

❶ We examine the request to see if it is an HTTP `POST` request, using the `request:get-method` function.

❷ We check that the `Content-Type` header was set to `image/jpeg`, as we only want to work with JPEG images in this example; if not, skip to ❽.

❸ We call the function `local:store-image` on the body of the `POST` request, which we obtained using the `request:get-data` function. This function has been omitted for brevity, but all you need to know right now is that it stores the image into the database, and returns a path to the image in the database.

❹ We create a URI for our newly stored image, based on the current URI of our API, which we can find by using `request:get-uri` and some substring of the path to the image in the database.

❺ We want to be good REST citizens, so we set the response status to `201 Created`, as we have just created the resource given to us in the database.

❻ When creating a resource, REST calls for the `Location` header in the response to be set with a URI to the new resource, so we do that.

❼ As an added bonus, we also return an identifier for the created resource in the body of the response; this identifier may then be used in subsequent requests to the API.

❽ If the `Content-Type` of the request was not `image/jpeg`, we do not wish to process the request, so we set the response status to `400 Bad Request`.

Retrieve a stored image from the database. The API provided by the *image-api.xq* file allows you to send an HTTP `GET` to it via the REST Server API to get a previously stored image. In your HTTP request, if the URI includes an identifier of an image previously stored by the API, then it will return the content of that image.

When you make the following request with cURL (`28068cd4-4817-4f81-ae19-5ad2c945186a.jpg` is the identifier of the image returned by the API when we stored it in the previous section):

```
curl http://localhost:8080/exist/rest/db/image-api.xq
  /28068cd4-4817-4f81-ae19-5ad2c945186a.jpg
```

the code in our *image-api.xq* stored query handles it like so:

```
else if(request:get-method() eq "GET")then ❶

   (: NOTE: thumbnail part is dealt with in the next section! :)

    else if(matches(
      request:get-uri(),
      concat(".*/", $local:uuidv4-pattern, "\.jpg$")
    ))then ❷
        let $image-name := tokenize(request:get-uri(), "/")[last()] ❸,
        $image := local:get-image($image-name) ❹
        return
            if(not(empty($image)))then ❺
                response:stream-binary($image, "image/jpeg", $image-name) ❻
            else
            (
                response:set-status-code($response:NOT-FOUND), ❼
                <image-not-found>{$image-name}</image-not-found>
            )
```

❶ We examine the request to see if it is an HTTP `GET` request, using the `request:get-method` function.

❷ We examine the URI after calling `request:get-uri` to see if it contains the identifier of an image.

❸ We extract the identifier of the image from the URI.

❹ We call the function `local:get-image` with the identifier of the image (`$image-name`). This function has been omitted for brevity, but all you need to know right now is that it retrieves an image previously stored into the database; otherwise (i.e., if there is no image with that identifier in the database), it returns an empty sequence.

❺ We test if we have an image from the database for the identifier.

❻ We have an image, so we return it in the HTTP response by calling the `response:stream-binary` function.

❼ Otherwise, there was no image in the database matching the identifier, so we set the response status to `404 Not Found` and return an explanation in the body of the response.

Retrieve a thumbnail representation of an image from the database. The API provided by the *image-api.xq* file allows you to send an HTTP `GET` to it via the REST Server API to get a thumbnail of a previously stored image. If the URI in your HTTP request includes an identifier of an image previously stored by the API prefixed by `thumbnail/`, it will return a thumbnail representation of that image. The *image-api.xq* file will generate the thumbnail on the fly, store it into the database, and return it; if the same thumbnail is requested a second time, the API serves it from the database rather than regenerating it.

To see this in action, make the following request with cURL:

```
curl http://localhost:8080/exist/rest/db/image-api.xq/
  thumbnail/28068cd4-4817-4f81-ae19-5ad2c945186a.jpg
```

The `thumbnail/` URI segment has been inserted before the identifier of the image—compare this to the URI used in the previous section.

The code in our *image-api.xq* file for handling this request is actually very similar to that for retrieving an image, except for a few minor changes. Therefore, we will only really examine the differences:

```
else if(request:get-method() eq "GET")then
    if(matches(
      request:get-uri(),
      concat(".*/thumbnail/", $local:uuidv4-pattern, "\.jpg$") ❶
    ))then
         let $image-name := tokenize(request:get-uri(), "/")[last()],
        $image := local:get-or-create-thumbnail($image-name) ❷
        return
            if(not(empty($image)))then
                response:stream-binary(
                  $image,
                  "image/jpeg",
                  concat("thumbnail-", $image-name)
                )
            else
            (
                response:set-status-code($response:NOT-FOUND),
                <image-not-found>{$image-name}</image-not-found>
            )
```

❶ This is similar to retrieving an image, except as well getting as the identifier of the image we also check the request URI for the prefix `thumbnail/`.

❷ This is similar to retrieving an image, except we now call the function `local:get-or-create-thumbnail` instead of the function `local:get-image`.

While the *rest-stored-query/image-api.xq* example shows how you can simply construct your own APIs atop the REST Server API, there is a great deal more that you can achieve, such as URL rewriting and directly producing web pages in HTML. For further details, see Chapter 9.

Using the REST Server API from Java

There are several good HTTP client libraries available for Java, including `java.net.URLConnection` in the standard Java library, but unfortunately for us, most of them take a somewhat low-level approach to HTTP, which means that you often need to build abstractions on top of them when using REST over HTTP. So instead, we will look at the Jersey client library, which is specifically designed for talking to REST Servers over HTTP—where the central abstraction is a resource. Jersey is an implementation of JAX-RS that enables you to easily construct REST services using *Java annotations*. However, it also has a client library that is very simple and elegant, and is well suited for communicating with the eXist REST Server API. At the time of writing, the latest Jersey version was 1.17.1.

There are just a few concepts that you need to understand in the Jersey client library beyond existing REST principles (such as `GET`, `PUT`, `POST`, and `DELETE`). In Jersey, we work with three kinds of objects:

`Client`

The `Client` object manages the underlying connection to the HTTP Server and any configuration required for that connection. The `Client` also allows you to construct `WebResource` objects. As it is mostly likely that we will want to authenticate with eXist when we manipulate resources via the REST Server API, we will actually make use of the Apache HTTP client integration for Jersey, as this allows us to provide authentication credentials. See Example 13-6.

Example 13-6. Constructing a suitable Client object for communicating with eXist using Jersey

```
//set up authentication
final CredentialsProvider credentialsProvider = new BasicCredentialsProvider();
credentialsProvider.setCredentials(AuthScope.ANY,
  new UsernamePasswordCredentials("admin", "my-admin-password"));

final DefaultApacheHttpClient4Config config =
  new DefaultApacheHttpClient4Config();
config.getProperties().put(
    ApacheHttpClient4Config.PROPERTY_CREDENTIALS_PROVIDER, credentialsProvider);

//construct the client
final Client client = ApacheHttpClient4.create(config);
```

`WebResource`

A `WebResource` object indicates a resource on the REST Server (although it may not exist yet) that is addressable by a URI. You may construct as many `WebResource` objects as you wish from a single client; you then perform actions on these resources, such as `PUT` or `GET`. See Example 13-7.

Example 13-7. Constructing a Jersey WebResource object for communicating with eXist

```
final String uri = "http://localhost:8080/exist/rest/db/some-document.xml";

final WebResource resource = client.resource(uri);
```

`ClientResponse`

Once you have a `WebResource` object, you can make a request to the server. Jersey offers some easy-to-use facilities to allow you to serialize/deserialize Java objects for the request/response as XML using JAXB. However, to keep things simple, we will just consider the raw response object that Jersey can provide from any request: the `ClientResponse`. The `ClientResponse` object allows you access to all of the HTTP responses sent from the REST Server, including all headers and bodies. See Example 13-8.

Example 13-8. Performing an HTTP header request against eXist with Jersey

```
final ClientResponse response = resource.head(ClientResponse.class);
final Status responseStatus = response.getClientResponseStatus();
if(responseStatus == Status.OK) {
    System.out.println(uri + " exists on the server.");
} else {
    System.err.println(uri + " does not exist on the server!");
}
```

Examples. The source code of four small complete examples of using Jersey from Java —to store a file, retrieve a file, query the database, and remove a file—is included in the folder *chapters/integration/restserver-client* of the *book-code* Git repository (see "Getting the Source Code" on page 15).

To compile the examples, enter the *restserver-client* folder and run `mvn package`.

Store example. You can then execute the StoreApp example using this command:

```
java -jar restserver-client-store/target/restserver-client-store-1.0-example.jar
```

This shows the available arguments for using the StoreApp.

A complete example of using the application might look like the following, which would upload the file */tmp/large.xml* to the collection */db/my-new-collection* in eXist:

```
java -jar restserver-client-store/target/restserver-client-store-1.0-example.jar
  localhost 8080 /tmp/large.xml /db/my-new-collection admin
```

Retrieve example. You can execute the RetrieveApp example like so:

```
java -jarrestserver-client-retrieve/target/
  restserver-client-retrieve-1.0-example.jar
```

This shows the available arguments for using the RetrieveApp.

A complete example of using the application might look like the following, which would download the resource */db/my-new-collection/large.xml* to the file in the current directory named *large.xml*:

```
java -jar restserver-client-retrieve/target/
  restserver-client-retrieve-1.0-example.jar localhost 8080
  /db/my-new-collection/large.xml admin > large.xml
```

Query example. You can execute the QueryApp example like so:

```
java -jar restserver-client-query/target/restserver-client-query-1.0-example.jar
```

This shows the available arguments for using the QueryApp.

A complete example of using the application might look like the following, which would find the family names of all of the people in all of the documents in the collection */db/my-new-collection* in the database:

```
java -jar restserver-client-query/target/
  restserver-client-query-1.0-example.jar localhost 8080
  "for \$person in //person return \$person/family" /db/my-new-collection admin
```

On non-Windows platforms, you must escape the `$` character used in XQuery for variables when sending the query in a command line from the terminal by prefixing with a `\` character. If you do not escape this symbol, the shell interpreter will try to interpret them as environment variables, which will result in an invalid XQuery and therefore an HTTP `400 Bad Request` response from the REST Server API.

Remove example. You can execute the RemoveApp example like so:

```
java -jar restserver-client-remove/target/
restserver-client-remove-1.0-example.jar
```

This shows the available arguments for using the RemoveApp.

A complete example of using the application might look like the following, which would remove the collection */db/my-new-collection* from the database:

```
java -jar restserver-client-remove/target/
restserver-client-remove-1.0-example.jar localhost 8080
/db/my-new-collection admin
```

XML-RPC API

RPC (Remote Procedure Call) allows you to call API functions in eXist from other processes; these calls are performed over HTTP. The XML aspect of the XML-RPC protocol indicates that the RPCs and their responses are encoded into XML documents, and it is these documents that are sent back and forth between eXist and the third-party process.

XML-RPC is a standardized protocol, with the XML documents used in requests and responses being well defined and documented by the XML-RPC specifications. However, the definitions of the functions available from eXist that you use in your RPC calls, their parameters, and their return types naturally differ from those in any other XML-RPC implementation. The base URI of the XML-RPC server in eXist on a default installation is *http://localhost:8080/exist/xmlrpc*.

There is nothing to stop you from using eXist's XML-RPC API from any application (such as cURL) that can, at a minimum, HTTP `POST` XML documents to eXist and process the XML document responses. However, eXist's XML-RPC API was never really designed to be used in this way. Thus, the contents of the XML documents for performing XML-RPC operations with eXist are not well documented. As it is a simple protocol of just XML over HTTP, though, if you are so inclined you can reverse-engineer it by reading the XML-RPC specification and studying the interface for eXist's XML-RPC API functions (in the `org.exist.xmlrpc.RpcApi` Java class, and the associated `RpcApi` JavaDoc). In fact, that is exactly the aim of this chapter. Rather than try to explain the entirety of eXist's XML-RPC API, in which there are over 122 available functions, we explain the methodology and tools needed to make use of this API.

Another option available to you is to use a network sniffing tool such as Wireshark to examine XML-RPC network traffic sent to and from eXist. For example, let's take a look at the first XML-RPC request made by eXist's Java Admin Client after you click Login. This traffic was captured using Wireshark:

```
POST /exist/xmlrpc HTTP/1.1 ❶
Content-Type: text/xml ❷
User-Agent: Apache XML RPC 3.1.3 (Sun HTTP Transport)
Authorization: Basic YWRtaW46 ❸
Cache-Control: no-cache
Pragma: no-cache
Host: localhost:8080 ❹
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive
Content-Length: 273

<?xml version="1.0" encoding="UTF-8"?>
<methodCall xmlns:ex="http://ws.apache.org/xmlrpc/namespaces/extensions">
    <methodName>existsAndCanOpenCollection</methodName> ❺
    <params>
        <param>
            <value>/db</value> ❻
        </param>
    </params>
</methodCall>
```

❶ Shows that this is just an HTTP `POST` to */exist/xmlrpc*

❷ Shows that we are just sending XML in the body of the request, which is exactly what we would expect for XML-RPC

❸ Shows that we are authenticating with the server using basic authentication (e.g., our encoded admin username and password)

❹ Shows the server we are HTTP `POST`ing to—that is, localhost port 8080

❺ Shows the name of the Java function in eXist that we are calling—that is, `existsAndCanOpenCollection`

❻ Shows that we are sending a single parameter value to the function—that is, `/db`

From the preceding Wireshark output we can infer that we are calling a function (also known as a method) in eXist using XML-RPC. That function will check for the existence of the collection */db* in the database and verify that the authenticated user has permission to open that collection.

Let's now look at the response to that request sent back to the client from eXist:

```
HTTP/1.1 200 OK ❶
Date: Wed, 08 May 2013 10:31:16 GMT
Set-Cookie: JSESSIONID=omg0pkl0xdvf1i26iv33z86r1;Path=/exist
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Content-Length: 287
Content-Type: text/xml ❷
Server: Jetty(8.1.9.v20130131)

<?xml version="1.0" encoding="UTF-8"?>
<methodResponse xmlns:ex="http://ws.apache.org/xmlrpc/namespaces/extensions">
    <params>
        <param>
            <value>
                <boolean>1</boolean> ❸
            </value>
        </param>
    </params>
</methodResponse>
```

❶ Shows that our request was successful (`200 OK`)

❷ Shows that we are receiving just XML in the body of the response, which is exactly what we would expect for XML-RPC

❸ Shows that the function we called returned a single parameter whose Boolean value is `1`—that is, `true`

From the preceding Wireshark output we can infer both that our function call was successful and that our function returned a positive result for the parameters provided to it. In this case, that means that the */db* collection does indeed exist and that our authenticated user has permission to read that collection.

Let's compare this to the Java definition of the `existsAndCanOpenCollection` function in eXist's XML-RPC API that was just called by our XML-RPC request:

```
/**
 * Determines whether a Collection exists in the database
```

```
     * and whether the user may open the collection
     *
     * @param collectionUri The URI of the collection of interest
     *
     * @return true if the collection exists and the user can open it,
     * false if the collection does not exist
     *
     * @throws PermissionDeniedException
     * If the user does not have permission to open the collection
     */
    boolean existsAndCanOpenCollection(final String collectionUri)
      throws EXistException, PermissionDeniedException;
```

The definition of the Java function `existsAndCanOpenCollection` should come as no surprise after seeing the XML-RPC dumps produced by Wireshark. We can clearly see that the method name, parameters, and method response in the XML-RPC documents match the Java definition. This means you can look at any Java function defined in eXist's XML-RPC API and with relative ease infer what the XML-RPC document to call it and the response that you will get back should look like.

But wait—as previously mentioned, the XML-RPC API in eXist was not designed with the idea in mind that developers would directly send and receive XML documents to and from it. Rather, as XML-RPC is a standardized protocol, it was intended to allow any developer to use an XML-RPC client library from her programming language of choice to talk to eXist. An XML-RPC client library makes life much easier for developers, as they can simply make standard function calls in their programming language, and the XML-RPC client will take care of serializing these to XML, sending them to the XML-RPC server API (i.e., eXist) over HTTP, receiving the responses, deserializing the XML back into the various primitives and objects defined in their programming language, and returning these results as those of the function calls they initially made.

So, you may be wondering why we briefly studied the raw wire protocol of XML-RPC if there are client libraries that we can use to avoid this. Well, in practice there are many XML-RPC libraries available for many different programming languages, but they are in various states of maturity. Understanding the underlying XML-RPC protocol itself (which is relatively simple) gives us a great tool for gaining insight when debugging communication problems with eXist using XML-RPC. For reference, the Linux Documentation Project has an excellent page on using XML-RPC from various programming languages, complete with practical examples.

Using the XML-RPC client API from Java

There are several options available for XML-RPC libraries in Java. eXist itself makes use of the Apache XML-RPC library, both for its XML-RPC server and as its underlying client in the Java Admin Client for remote connections. Here, though, we will look at the Redstone XML-RPC library, as it is much smaller, simpler, and in many ways easier to use than the Apache library. At the time of writing, the latest version was 1.1.1.

The Redstone XML-RPC library offers two methods of use to a client (as does the Apache library):

Classic XML-RPC client API (Example 13-9)
: Basically, you tell the client about the server method and the parameters that you wish to send, and then make a call to that method with the client against the server. The client returns an object, which you then interrogate and cast to get your result.

Example 13-9. Redstone classic XML-RPC client

```java
final URL url = new URL("http://localhost:8080/exist/xmlrpc");
final XmlRpcClient rpc = new XmlRpcClient(url, true);

final Object result = rpc.invoke(
  "existsAndCanOpenCollection", new Object[] { "/db" });
```

Dynamic proxy XML-RPC client API (Example 13-10)
: This is a much more modern approach than the classic one and much easier to use. Basically, your server defines a Java interface (i.e., the interface `org.exist.xmlrpc.RpcAPI` for eXist), and you make a copy of that interface to your client application. You then ask the XML-RPC library to create a proxy to the server using that interface. The client gives you a standard Java object, which implements the interface. You can then use this Java object just like any other, and all the client/server communication is hidden from you. When you call a function on the object, the client takes care of all of the communication with the server and returns the result.

Example 13-10. Redstone dynamic proxy XML-RPC client

```java
final URL url = new URL("http://localhost:8080/exist/xmlrpc");
final RpcAPI rpc = (RpcAPI)XmlRpcProxy.createProxy(
    url, "Default", new Class[] { RpcAPI.class }, true);

final Boolean result = rpc.existsAndCanOpenCollection("/db");
```

In the dynamic proxy approach, calling the RPC method is much simpler because the interface forms the code contract as opposed to naming the method (with a string), providing an arbitrary number of arguments in an array, and receiving an untyped result in the nonproxy approach. As the functions, the number of arguments, their types, and the type of the result are known statically at compile time, it's much harder to make mistakes when you call the API—any mistake will prevent your client program from compiling.

The only serious advantage of the classic XML-RPC approach over a dynamic proxy approach from Java is that with the classic approach, you do not need to copy the Java interface (and any of its dependencies) from the server to the client application.

Examples. The source code of two small complete examples of using the Redstone XML-RPC library from Java (to store a file and retrieve a file) is included in the folder *chapters/integration/xmlrpc-client* of the *book-code* Git repository (see "Getting the Source Code" on page 15). One example demonstrates the classic XML-RPC approach, while the other demonstrates the dynamic proxy approach.

To compile the examples, enter the *xmlrpc-client* folder and run `mvn package`.

Classic store example. You can then execute the StoreApp example like so:

```
java -jar xmlrpc-client-store/target/xmlrpc-client-store-1.0-example.jar
```

This shows the available arguments for using the StoreApp.

A complete example of using the application might look like the following, which would upload the file */tmp/large.xml* to the collection */db/my-new-collection* in eXist:

```
java -jar xmlrpc-client-store/target/xmlrpc-client-store-1.0-example.jar
  localhost 8080 /tmp/large.xml application/xml /db/my-new-collection admin
```

The XML-RPC StoreApp example takes an extra parameter compared to the StoreApp examples for other APIs, as specifying the Internet media type of the resource is mandatory for uploading files via eXist's XML-RPC API. You may use `application/xml` for XML documents; for anything else, if you do not know the Internet media type it is recommended that you use `application/octet-stream`, which will store the document into eXist as an untyped binary resource.

Proxy store example. The proxy store example is externally exactly the same as the classic store example; its implementation just varies as described previously.

You can execute the ProxyStoreApp example like so:

```
java -jar xmlrpc-proxy-client-store/target/
  xmlrpc-proxy-client-store-1.0-example.jar
```

This shows the available arguments for using the ProxyStoreApp.

A complete example of using the application might look like the following, which would upload the file */tmp/large.xml* to the collection */db/my-new-collection* in eXist:

```
java -jar xmlrpc-proxy-client-store/target/
  xmlrpc-proxy-client-store-1.0-example.jar localhost 8080
  /tmp/large.xml application/xml
  /db/my-new-collection admin
```

Using the XML-RPC client API from Python

As XML-RPC is such a prevalent and well-supported protocol, we felt that a simple example for a non-Java programming language might be beneficial to those of you who are not familiar with Java. We have created a direct port of the dynamic proxy XML-RPC example into Python, the source code of which is included in the file *chapters/integration/xmlrpc-client/StoreApp.py* of the *book-code* Git repository.

Python has a built-in XML-RPC library called `xmlrpclib`, which is very simple to use (see Example 13-11). It works on the principle of dynamic proxies, but as Python is more relaxed in its compilation here than Java, you will not see compile-time errors if you try to call an RPC method that does not exist. This is because Python has no knowledge at compile time of the RPC methods made available by eXist, as you have not needed to provide it an interface like you do with Java. The version of Python used was 2.7.2.

Example 13-11. Python dynamic proxy XML-RPC client

```python
import xmlrpclib

rpc = xmlrpclib.ServerProxy("http://localhost:8080/exist/xmlrpc")

result = rpc.existsAndCanOpenCollection("/db")
print "Collection exists and we can open it: %s" % result
```

Python XML-RPC proxy store example. The source code of a small example of using XML-RPC from Python to store a file is included in the folder *chapters/integration/xmlrpc-client* of the *book-code* Git repository (see "Getting the Source Code" on page 15).

The Python store example is externally exactly the same as the Java dynamic proxy example, and can be found in the *StoreApp.py* file. You can execute the Python Store-App example like so:

```
python StoreApp.py
```

A complete example of using the application might look like the following, which would upload the file */tmp/large.xml* to the collection */db/my-new-collection* in eXist:

```
python StoreApp.py localhost 8080
  /tmp/large.xml application/xml
  /db/my-new-collection admin
```

XML:DB Remote API

The XML:DB API was developed by the XML:DB Initiative in the early 2000s with the goal of creating a common standardized API for communicating with XML databases. While the API is no longer actively developed, it arguably fulfilled its goal and gained adoption from several XML database vendors. The XML:DB API is really just a set of Java interfaces that a vendor must implement. eXist either implements these as function calls on its internal API for local embedded use (see "XML:DB Local API" on page 366), or—more interestingly here for its remote use—uses its XML-RPC implementation (see "XML-RPC API" on page 342) for network communication.

The main advantage of the XML:DB API is that it provides a complete Java client library that you can use seamlessly from your Java applications without concern for how the network communication with the eXist server is achieved. In practice there is not much of a semantic difference between this and using a dynamic proxied XML-RPC client approach. Whether you should use the XML:DB API or an XML-RPC dynamic proxied client API really comes down to a matter of choice with regard to the style of code that you wish to write. We feel that using the XML-RPC approach offers greater flexibility, as you have full access to the API underlying the XML:DB API; also the XML:DB API is becoming more limited due to its stagnation.

The downside of the XML:DB API is that it is only easily usable from Java, unless you are willing to reverse-engineer the XML-RPC messages sent by the XML:DB API and produce your own client library for a different programming language.

The base URI of the XML:DB Remote API server in eXist on a default installation is *xmldb:exist://localhost:8080/exist/xmlrpc*. Even though the URI scheme is `xmldb`, the URI points to eXist's XML-RPC server, and the XML-RPC server itself uses HTTP. Thus, XML:DB (as far as eXist is concerned) is really just more XML-RPC performed over HTTP.

Using the XML:DB Remote API from Java

eXist provides a remote client library implementation of the XML:DB API, which can be used from your own Java applications. If you wish to use it from your own applications, you need to make sure the libraries listed in Table 13-6 are available on your classpath.

Table 13-6. Dependencies for eXist remote XML:DB Java applications

Library	Description
$EXIST_HOME/lib/core/xmldb.jar	XML:DB API library.
$EXIST_HOME/exist.jar	eXist core. Contains the eXist XML:DB client library implementation.
$EXIST_HOME/lib/core/xmlrpc-client-3.1.3.jar	Apache XML-RPC Client library. Dependency of eXist's XML:DB client library.
$EXIST_HOME/lib/core/xmlrpc-common-3.1.3.jar	Apache XML-RPC common code. Dependency of the Apache XML-RPC client library.
$EXIST_HOME/lib/core/commons-io-2.4.jar	Apache Commons I/O library. Dependency of eXist's XML:DB client library.

There are just four main concepts that you need to understand in the XML:DB API to make XML:DB requests to eXist:

`Driver`*s*
: The XML:DB API makes use of drivers so that the same API can be used by different vendors, and each vendor just needs to provide a driver. eXist provides the `Driver` class `org.exist.xmldb.DatabaseImpl` (see Example 13-12).

Example 13-12. Registering eXist's XML:DB driver

```
final Class<Database> dbClass =
    (Class<Database>) Class.forName("org.exist.xmldb.DatabaseImpl");
final Database database = dbClass.newInstance();
database.setProperty("create-database", "true");

DatabaseManager.registerDatabase(database);
```

`Collection`*s*
: A `Collection` in the XML:DB API maps onto a collection in the eXist database. `Collections` are the primary means for interacting with the XML:DB API. Accessing a `Collection` requires authentication, after which all subsequent operations on that collection and any subresources or subcollections of that `Collection` use the same credentials. Administrative and query services can also be retrieved from a `Collection`. See Example 13-13.

Example 13-13. Accessing a remote XML:DB collection

```
Collection collection =
      DatabaseManager.getCollection(
      "xmldb:exist://localhost:8080/exist/xmlrpc/db",
      "admin", "");
```

`Resources`
: A `Resource` in the XML:DB API maps onto a document in the eXist database. eXist's implementation of the XML:DB API allows you to work with both its XML and binary documents. See Example 13-14.

Example 13-14. Retrieving a Resource from an XML:DB remote collection

```
Resource resource = collection.getResource("some-document.xml");
```

`Services`
: A `Service` in the XML:DB API allows you to perform extended operations against the database. The XML:DB API provides services for collection management, XPath/XQuery, and XUpdate services. In addition, eXist provides some eXist-specific XML:DB services for user management, database instance management, and index queries. You retrieve a `Service` from a connection to a `Collection` by specifying its name and version. See Example 13-15 and Table 13-7.

Example 13-15. Obtaining a query service from an XML:DB remote collection

```
XPathQueryService queryService =
      collection.getService("XPathQueryService", "1.0");
```

Table 13-7. eXist XML:DB services

Service name(s)	Java class (org.exist.xmldb)	Description
`XPathQueryService` `XQueryService`	`RemoteXPathQueryService`	XML:DB XPath Service. In eXist, XQuery is also offered.
`CollectionManagementService` `CollectionManager`	`RemoteCollectionManagementService`	XML:DB Collection Management Service.
`XUpdateQueryService`	`RemoteXUpdateQueryService`	XML:DB XUpdate Service.
`UserManagementService`	`RemoteUserManagementService`	eXist User Management Service extension for XML:DB.

Service name(s)	Java class (org.exist.xmldb)	Description
`DatabaseInstanceManager`	`RemoteDatabaseInstanceManager`	eXist Database Instance Management Service extension for XML:DB.
`IndexQueryService`	`RemoteIndexQueryService`	eXist Index Query Service extension for XML:DB.

With the XML:DB API in eXist, you are responsible for cleaning up after yourself!

That is to say, if you open a collection, you *must close the collection* when you are finished with it; likewise, if you open a resource, you *must free that resource* when you are finished with it.

You can close a collection by calling its `close` method, and you can free a resource by casting it to an `org.exist.xmldb.EXistResource` and calling its `freeResources` method.

Examples. The source code of four small complete examples of using the XML:DB Remote API from Java—to store a file, retrieve a file, query the database, and remove a file—are included in the folder *chapters/integration/xmldb-client* of the *book-code* Git repository (see "Getting the Source Code" on page 15).

To compile the examples, enter the *xmldb-client* folder and run `mvn package`.

Store example. You can then execute the StoreApp example like so:

```
java -jar xmldb-client-store/target/xmldb-client-store-1.0-example.jar
```

This shows the available arguments for using the StoreApp.

A complete example of using the application might look like the following, which would upload the file */tmp/large.xml* to the collection */db/my-new-collection* in eXist:

```
java -jar xmldb-client-store/target/xmldb-client-store-1.0-example.jar
  localhost 8080 /tmp/large.xml true /db/my-new-collection admin
```

The XML:DB StoreApp example takes an extra parameter compared to the StoreApp examples for other APIs, as the API requires you to know if you are storing an XML or binary document. You may use `true` for XML documents, and `false` for binary documents.

Retrieve example. You can execute the RetrieveApp example like so:

```
java -jar xmldb-client-retrieve/target/xmldb-client-retrieve-1.0-example.jar
```

This shows the available arguments for using the RetrieveApp.

A complete example of using the application might look like the following, which would download the resource *db/my-new-collection/large.xml* to the file in the current directory named *large.xml*:

```
java -jar xmldb-client-retrieve/target/xmldb-client-retrieve-1.0-example.jar
  localhost 8080 /db/my-new-collection/large.xml admin > large.xml
```

Query example. You can execute the QueryApp example like so:

```
java -jar xmldb-client-query/target/xmldb-client-query-1.0-example.jar
```

This shows the available arguments for using the QueryApp.

A complete example of using the application might look like the following, which would find the family names of all of the people in all of the documents in the collection */db/my-new-collection* in the database:

```
java -jar xmldb-client-query/target/xmldb-client-query-1.0-example.jar
  localhost 8080 "for \$person in //person return \$person/family"
  /db/my-new-collection admin
```

On non-Windows platforms, you must escape the `$` character used in XQuery for variables when sending the query in a command line from the terminal by prefixing with a `\` character. If you do not escape this symbol, the shell interpreter will try to interpret them as environment variables, which will result in an invalid XQuery and therefore an `org.exist.xmldb.XMLDBException` response from the XML:DB server API.

Remove example. You can execute the RemoveApp example like so:

```
java -jar xmldb-client-remove/target/xmldb-client-remove-1.0-example.jar
```

This shows the available arguments for using the RemoveApp.

A complete example of using the application might look like the following, which would remove the collection */db/my-new-collection* from the database:

```
java -jar xmldb-client-remove/target/xmldb-client-remove-1.0-example.jar
  localhost 8080 /db/my-new-collection admin
```

RESTXQ

RESTXQ itself is not an API; rather, it is a framework that enables you to build your own APIs. The beauty of this is that you can construct small and elegant application-specific APIs for the Web or internal purposes using REST over HTTP.

"Building Applications with RESTXQ" on page 215 covers the specifics of building REST APIs with RESTXQ, so we will not repeat those here. For the purposes of integrating your custom RESTXQ REST APIs with other applications or processes, the main requirement is a decent HTTP client, for which we refer you back to the information in "REST Server API" on page 319.

With RESTXQ the developer declares a series of HTTP request constraints against an XQuery function by use of *XQuery 3.0 annotations*; the function with constraints is then known as a *resource function*. When eXist receives an incoming HTTP request it checks all of the known resource functions to see if the HTTP request could be serviced by one of them; if so, the function is executed, and parameters from the HTTP request may be extracted and injected into the function call as parameters to that function. The resource function is then (apart from user-defined processing) responsible for constructing an appropriate HTTP response.

For comparison with the REST Server API, we include with this chapter the XQuery file *restxq-stored-query/image-api.xqm*, which is a port of the *rest-stored-query/image-api.xq* file, discussed in "Executing stored queries" on page 335. We hope that this will aid you in recognizing the different coding styles for using RESTXQ and stored queries with the REST Server API.

To use the RESTXQ version of *image-api.xq*, you simply need to store it into any collection in the database for which RESTXQ is enabled (by default, this is all database collections apart from those of specific applications in subcollections of */db/apps* that have chosen to disable RESTXQ). Also, you need to ensure that: 1) the calling user has execute access within the database to the *image-api.xqm* file, so that it may be executed; 2) the collection */db/images* exists and is writable by the calling user; and 3) the Image XQuery extension module is enabled in *$EXIST_HOME/conf.xml*.

Recall from "Executing stored queries" on page 335 that the custom *image-api* REST API performs these three functions:

- Stores a JPEG image received over HTTP into the database
- Retrieves a stored image from the database
- Retrieves a thumbnail representation of an image from the database

Let's now look at the XQuery code in detail, and how it performs each of these functions.

Store a JPEG image received over HTTP into the database

The API provided by the *image-api.xqm* file allows you to send an HTTP `POST` to it via the RESTXQ API to store a JPEG image. In your HTTP request, if you set the `Content-Type` to `image/jpeg` and include the content of a JPEG image in the body of the request, it will be stored into the database and *image-api.xqm* will return a `Location` and `identifier` in the HTTP response for the image.

Consider the following example, where we use cURL to make a request to a RESTXQ resource function that stores a JPEG image into the database:

```
curl -i -X POST -H 'Content-Type: image/jpeg' --data-binary @/tmp/cats.jpg
  http://localhost:8080/exist/restxq/image
```

Let's look at how the code in our *image-api.xqm* stored query handles this request:

```
declare
    %rest:POST❶("{$image-data}") ❷
    %rest:path("/image") ❸
    %rest:consumes("image/jpeg") ❹
function ii:store-image($image-data) { ❺
    let $image-name := util:uuid() || ".jpg"
    let $db-path :=
            xmldb:store($ii:image-collection, $image-name, $image-data,
            "image/jpeg") ❻
    let $uri-to-resource := rest:uri() || "/" || $image-name ❼
    return
    (
        <rest:response> ❽
            <http:response status="{$ii:HTTP-CREATED}"> ❾
                <http:header name="Location" value="{$uri-to-resource}"/>
            </http:response>
        </rest:response>
        ,
        <identifier>{$image-name}</identifier> ❿
    )
};
```

❶ We declare that we are only interested in processing HTTP `POST` requests.

❷ We request to have the body of the `POST` request extracted into the function parameter `$image-data`.

❸ We declare that we are only interested in processing HTTP requests that have a URI (relative to the RESTXQ API) of `/image`.

❹ We declare that we are only interested in consuming HTTP requests that have a `Content-Type` of `image/jpeg`.

❺ The `$image-data` will receive the body of the HTTP `POST` when the function is executed, as we declared in ❷.

❻ We call the function `xmldb:store` on the body of the `POST` request. This function stores the image into the database, and returns a path to the image in the database.

❼ We construct a public, dereferenceable URI to the stored image. Of particular interest here is the call to `rest:uri`, which gives us the absolute URI of the executing resource function.

❽ The response of the function will be a sequence, where the first item will instruct RESTXQ about the HTTP response and the second item will be the body of the HTTP response.

❾ We instruct RESTXQ to set the HTTP response code to `201 Created` and add an HTTP header declaring a URI to the location of the stored image.

❿ As an added bonus, we also return an identifier for the created resource in the body of the response; this identifier may then be used in subsequent requests to the API.

Retrieve a stored image from the database

The API provided by the *image-api.xqm* file allows you to send an HTTP `GET` to it via the RESTXQ API to get a previously stored image. If the URI in your HTTP request includes an identifier of an image previously stored by the API, then it will return the content of that image.

Consider the following example, where we use cURL to make a request to a RESTXQ resource function that returns an image from the database:

```
curl http://localhost:8080/exist/rest/db/image/
  24a85a52-5031-4bac-8843-4c7e7701905b.jpg
```

`24a85a52-5031-4bac-8843-4c7e7701905b.jpg` is the identifier of the image returned by the API when we stored it in the previous section.

Let's look at how the code in our *image-api.xqm* stored query handles this request:

```
declare
    %rest:GET ❶
    %rest:path("/image/{$image-name}") ❷
    %rest:produces("image/jpeg") ❸
    %output:method("binary") ❹
function ii:get-image-rest($image-name) { ❺
    let $image := ii:get-image($image-name) ❻
```

```
        return
            if(not(empty($image)))then
                $image ❼
            else
                <rest:response>
                    <http:response status="{$ii:HTTP-NOT-FOUND}"> ❽
                        <http:header name="Content-Type" value="application/xml"/>
                    </http:response>
                </rest:response>
    };
```

❶ We declare that we are only interested in processing HTTP `GET` requests.

❷ We declare that we are only interested in processing HTTP requests that have a URI (relative to the RESTXQ API) starting with `/image` and followed by *any path segment*, which should be extracted into the function parameter `$image-name`.

❸ We declare that we are only interested in consuming HTTP requests that can accept a response with `Content-Type image/jpeg`.

❹ We declare that we would like any body returned by our resource function to be serialized to the HTTP response as binary.

❺ The `$image-name` will receive the value of a path segment from the URI when the function is executed, as we declared in ???.

❻ We call the function `ii:get-image` with the identifier of the image (`$image-name`). This function has been omitted for brevity, but all you need to know right now is that it retrieves an image previously stored into the database; otherwise (i.e., if there is no image with that identifier in the database), it returns an empty sequence.

❼ We have an image, so we return it from the resource function to be serialized to the HTTP response.

❽ Alternatively, if there was no image in the database matching the identifier, we set the response status to `404 Not Found`.

Retrieve a thumbnail representation of an image from the database

The API provided by the *image-api.xqm* file allows you to send an HTTP `GET` to it via the RESTXQ API to get a thumbnail of a previously stored image. If the URI in your HTTP request includes an identifier of an image previously stored by the API prefixed by `thumbnail/`, it will return a thumbnail representation of that image. The *image-api.xqm* file will generate the thumbnail on the fly, store it into the database, and return it; if the same thumbnail is requested a second time, the API serves it from the database rather than regenerating it.

Consider the following example, where we use cURL to make a request to a RESTXQ resource function to return an image thumbnail:

```
curl http://localhost:8080/exist/restxq/image/thumbnail/
  24a85a52-5031-4bac-8843-4c7e7701905b.jpg
```

Observe the `thumbnail/` before the identifier of the image, in comparison to the URI used in "Retrieve a stored image from the database" on page 356.

The code in our *image-api.xqm* file for handling this request is actually very similar to that for retrieving an image, except for a few minor changes. Therefore, we will only really examine the differences here:

```
declare
    %rest:GET
    %rest:path("/image/thumbnail/{$image-name}") ❶
    %rest:produces("image/jpeg")
    %output:method("binary")
function ii:get-or-create-thumbnail($image-name) {
    let $thumbnail-image-name := "thumbnail-" || $image-name, ❷
    $thumbnail-db-path := $ii:image-collection || "/" || $thumbnail-image-name
    return

        (: does the thumbnail already exist in the database? :)
        if(util:binary-doc-available($thumbnail-db-path))then
            (: yes, return the thumbnail :)
            ii:get-image($thumbnail-image-name)

        else
            (: no, does the original image of which we want a
               thumbnail exist in the database? :)
            let $image := ii:get-image($image-name)
            return
                if(not(empty($image)))then
                    (: yes, create the thumbnail :)
                    let $thumbnail :=
                        image:scale($image, (400, 200), "image/jpeg"),
                    $thumbnail-db-path :=
                        xmldb:store(
                              $ii:image-collection,
                              $thumbnail-image-name,
                              $thumbnail,
                              "image/jpeg")
                    return
                        $thumbnail
                else
                    <rest:response>
                        <http:response status="{$ii:HTTP-NOT-FOUND}">
                            <http:header name="Content-Type"
                             value="application/xml"/>
                        </http:response>
                    </rest:response>
};
```

❶ This is similar to our code for retrieving an image, except as well as the identifier of the image we declare that we are only interested in URI paths that also have a `/thumbnail` segment.

❷ Instead of retrieving an image, we now create or retrieve a thumbnail. The details of this are out of scope here, and the code is not too difficult to understand; the main point of interest is the call to the extension `image:scale`, which will actually generate the thumbnail image.

Hopefully, you will agree that the RESTXQ version is simpler and easier to understand than the REST Server API version. For example, in this specific example we have not had to handle unwanted requests and return an HTTP `400 Bad Request` or HTTP `406 Method Not Allowed`, as the RESTXQ API takes care of that for us.

XQJ

XQJ is a standardized Java API developed by the JCP (Java Community Process) as JSR-225. A JSR (Java Specification Request) is centered solely on Java, and thus the API is not really suitable for direct use in other programming languages. The implementation of the XQJ server in eXist is really just a few extensions to eXist's REST Server, with any XQJ client expected to communicate using HTTP via the REST Server API. If you like the XQJ API but do not like Java, then theoretically there is nothing to stop you from implementing an XQJ-like client in any language, and it should not be too difficult providing you understand the REST Server API.

XQJ JSR-225 focuses solely on XQuery: it allows you to send an XQuery to the server, have it executed, and receive the results. It also allows you to prepare XQuery expressions that can be parameterized and executed later (similar to *prepared statements* in JDBC). While XQJ does *not* directly provide any facilities for managing documents or the database, it is possible to achieve similar functionality by using eXist's XQuery `xmldb` extension module (see the entry for `xmldb` in Appendix A).

eXist has chosen only to implement a server API for use by XQJ; it does not provide an XQJ client implementation. This is mainly because there is an excellent and freely available XQJ client implementation from Charles Foster at *http://www.xqj.net*, which you may use in your own Java programs.

There are just four main concepts that you need to understand in the XQJ API to make requests to eXist—*data sources*, *connections*, *expressions*, and *result sequences*:

`XQDataSource`
: The data source provides the main driver of XQJ and defines how you connect to the server. With the `net.xqj.exist.ExistXQDataSource` implementation from *xqj.net*, you need to set two properties to be able to connect to eXist (see Example 13-16):

  `serverName`
  : This is the hostname or IP address of the eXist server that you wish to connect to. If you are running your XQJ client on the same machine as eXist, you may use either `localhost` or `127.0.0.1`.

  `port`
  : This is the TCP port that the eXist server you wish to connect to is listening on. If you have not reconfigured this setting in eXist, it will be `8080` by default.

Example 13-16. Setting up the XQDataSource for connecting to eXist

```
final XQDataSource xqs = new ExistXQDataSource();
xqs.setProperty("serverName", "localhost");
xqs.setProperty("port", "8080");
```

`XQConnection`
: The connection represents an XQJ-connected session with the server and is obtained from the data source. When requesting the connection from the data source, you should provide your username and password for accessing eXist. The eXist XQJ implementation uses REST, so there is no persistent connection; rather, HTTP calls are made as needed by the `XQConnection` object. However, you should always call `close` on the `XQConnection` object to clean up any retained objects. See Example 13-17.

Example 13-17. Opening an authenticated XQConnection to eXist

```
XQConnection connection = dataSource.getConnection("admin", "mypassword");
```

`XQExpression`
: The expression represents an XQuery expression that may be sent to the server and executed. It is also possible to use `XQPreparedExpression` if you wish to execute the same expression multiple times with different parameters (e.g., external variable bindings).

`XQResultSequence`
: As a result of executing an `XQExpression` or `XQPreparedExpression`, a result sequence is generated that may be iterated over to retrieve results from the server.

Examples

The source code of a simple example of using the *xqj.net* XQJ API from Java to query the database is included in the folder *chapters/integration/xqj-client* of the *book-code* Git repository (see "Getting the Source Code" on page 15).

To compile the example, enter the *xqj-client* folder and run `mvn package`.

Query example. You can then execute the QueryApp example like so:

```
java -jar xqj-client-query/target/xqj-client-query-1.0-example.jar
```

This shows the available arguments for using the QueryApp.

A complete example of using the application might look like the following, which would find the family names of all of the people in all of the documents in the collection */db/my-new-collection* in the database:

```
java -jar xqj-client-query/target/xqj-client-query-1.0-example.jar localhost 8080
  "for \$person in //person return \$person/family" /db/my-new-collection admin
```

On non-Windows platforms, you must escape the `$` character used in XQuery for variables when sending the query in a command line from the terminal by prefixing with a `\` character. If you do not escape this symbol, the shell interpreter will try to interpret them as environment variables, which will result in an invalid XQuery and therefore an `XQJException: XQJQS001 - Invalid XQuery syntax` response from the XQJ API.

Deprecated Remote APIs

From eXist 2.0 onward, several APIs that were available in previous versions are now deprecated. These APIs have been deprecated either because the eXist developers felt that they were infrequently used by the community, because they had been replaced by more modern APIs, or because the contributors of these APIs no longer supported them. We'll look at a few of them here.

Atom Servlet

The Atom Servlet in eXist provides an implementation of the IETF Atom syndication format and publishing protocol. The Atom Servlet was originally written in Java, but its developers felt that a newer implementation written in XQuery would not only offer better support, but also be easier to maintain.

Currently the Atom Servlet is still present in eXist, but if you are not already using it, we would advise you not to start! An Atom API implemented in XQuery is already under development and should hopefully be released in the near future as a replacement for the Atom Servlet.

SOAP API

Since eXist 0.8, the Axis and Admin Servlets have provided a rudimentary SOAP API for eXist. The Axis Servlet provides retrieval and query services, while the Admin Servlet provides services for storing and removing documents and collections. Both Servlets are implemented with what is now quite an old version of Apache Axis, and use the RPC-encoded form of SOAP. Around late 2006 it was widely expected that the SOAP API written in Java would be replaced by XQuery web services implemented for the SOAP Server (see "SOAP Server" on page 362), but unfortunately that work was never completed.

The use of SOAP today is often considered bloated and convoluted, and hence is often much discouraged in favor of REST. The SOAP API in eXist was deprecated with the release of eXist 2.0. Instead, it is recommended to use either the RESTXQ API, the REST Server API, or the XML-RPC API. If enough interest in SOAP appears again from the community, it is most likely that a new SOAP implementation will be developed based on XQuery 3.0 annotations influenced by JAX-WS, in a similar fashion to RESTXQ (see "RESTXQ" on page 353).

For Microsoft .NET developers there is still something of an advantage in using the SOAP API because of the wizard-driven web service client proxy generation tools offered by Microsoft Visual Studio. While we would suggest using the REST API if you are investing in eXist-db in the medium to long term, the SOAP API can be the easiest and fastest route to a working application for .NET developers in the short term.

SOAP Server

The SOAP Server was developed in 2006 as a mechanism for transparently wiring SOAP requests and responses to XQuery functions. The SOAP Server automatically generates WSDL (Web Services Description Language) for an XQuery library module and marshals and demarshals the function parameters and results from and into a SOAP envelope. The SOAP Server attempted to deliver both *RPC* and *Document Literal* forms of SOAP web services transparently.

The SOAP Server was a useful experiment in enabling XQuery to deliver enterprise-style web services, but it was always underdeveloped and never lived up to its potential to replace the SOAP API (discussed in the previous section). The SOAP Server is still available in eXist 2.0, but it has been deprecated for some time and it is not recommended for production use. If you wish to provide SOAP web services from XQuery, it is recommended that you either build on top of RESTXQ (see "RESTXQ" on page 353) and manage the SOAP envelopes and WSDL generation yourself, or collaborate with the eXist community to build a successor.

Remote API Libraries for Other Languages

While many of the APIs discussed in this chapter are programming language–agnostic, the majority of our examples are provided in Java. There are also various other APIs, libraries, and bindings to make working with eXist from languages other than Java easier.

Please note that these are third-party, open source community offerings, and as such they are not maintained by the eXist project, nor have we personally assessed the quality of these offerings. Rather, we include a list of them for completeness and to act as signposts for you.

Community APIs for eXist by programming language

- JavaScript

  existdb-node by Wolfgang Meier
  : Connects to eXist via its REST API

- Perl

  XML-ExistDB by Mark Overmeer
  : Connects to eXist via its XML-RPC API

  PheXist by Oscar Celma
  : Connects to eXist via its SOAP API. Also available for PHP

- Python

  pyexist by Samuel Abels
  : Connects to eXist via its REST API

  EULexistdb by the Digital Programs and Systems Software Team of Emory University Libraries
  : Connects to eXist via its XML-RPC API. Can also be used in conjunction with Django

zopyx.existdb for Plone by Andreas Jung
: Connects to eXist via its REST API. An eXist plug-in for Plone CMS

- PHP

php-eXist-db-Client by CuAnnan
: Connects to eXist via its XML-RPC API

PheXist by Oscar Celma
: Connects to eXist via its SOAP API. Also available for PHP

- Ruby

eXist API by Jenda Sirl
: Connects to eXist via its XML-RPC API

rb_exist by Miquel Sabaté Solà
: Connects to eXist via its REST API. It was inspired by pyexist

- Scala

XQuery for Scala by Dino Fancellu
: Connects to eXist via its XQJ API

Local APIs

A local API enables you to embed eXist into your own Java application by placing the eXist libraries and configuration files in the classpath of your application and making function calls to eXist via one of its local APIs. When eXist is embedded in your application, both your own application's code and eXist's application code run within the same JVM process.

While there is nothing to stop you from calling eXist's own classes and functions directly, this is strongly discouraged and not officially supported by the eXist development team. Rather, you are advised to use one of the two available local APIs, described in "XML:DB Local API" on page 366 and "Fluent API" on page 369.

So, which local API should you use? There are a few factors to consider:

- Do you want to be able to switch your application between local and remote eXist instances? If so, then use the XML:DB API, as it is a single API to learn that supports either local or remote eXist servers.
- If you will only ever use eXist locally in embedded operations, then the Fluent API provides a more modern and simpler API for working with eXist.

When you embed eXist into your own application, because eXist shares the same JVM process and memory space as your application, should your application exhaust the memory available to the JVM or crash, this can affect the integrity of your eXist database. Take care when creating and freeing resources within your application and when exiting the JVM.

Whichever local API you choose, one challenge when embedding eXist into your own application is ensuring that you have all of the dependencies and configuration files that eXist relies on bundled with your application and available on the classpath. To a certain extent, the libraries bundled with eXist that you will also need to bundle with your application will depend on which features of eXist you wish to use, but at an absolute minimum you will need the runtime dependencies listed in Table 13-8.

Table 13-8. Minimum dependencies for embedding eXist 2.1 in your own application

Library	Description
$EXIST_HOME/exist.jar	Contains the eXist core implementation.
$EXIST_HOME/start.jar	Contains the eXist startup helpers. Dependency of eXist core.
$EXIST_HOME/lib/core/xmldb.jar	XML:DB API library.
$EXIST_HOME/lib/core/commons-io-2.4.jar	Apache Commons I/O library. Dependency of eXist's XML:DB client library and eXist core.
$EXIST_HOME/lib/core/pkg-repo.jar	EXPath PKG Repository library. Dependency of eXist core.
$EXIST_HOME/lib/core/commons-pool-1.6.jar	Apache Commons Pool library. Dependency of eXist's XML:DB client library.
$EXIST_HOME/lib/core/quartz-2.1.6.jar	Quartz Scheduler library. Dependency of eXist core.
$EXIST_HOME/lib/core/gnu-crypto-2.0.1-min.jar	GNU Crypto minimum library. Dependency of eXist core.
$EXIST_HOME/lib/core/commons-codec-1.7.1.jar	Apache Commons Codec library. Dependency of eXist core.

Library	Description
$EXIST_HOME/lib/core/antlr-2.7.7.jar	Antlr Parser Generator library. Dependency of eXist core.
$EXIST_HOME/lib/core/log4j-1.2.17.jar	Log4J logging library. Dependency of eXist core.
$EXIST_HOME/lib/endorsed/xercesImpl-2.11.0.jar	Apache Xerces2 XML Parser library. Dependency of eXist core.
$EXIST_HOME/lib/endorsed/xml-resolver-1.2.jar	Apache XML Commons Resolver library. Dependency of eXist core.
$EXIST_HOME/tools/aspectj/lib/aspectrt-1.7.1.jar	AspectJ AOP library. Dependency of eXist core.
$EXIST_HOME/lib/optional/servlet-api-3.0.jar	Java Servlet API. Dependency[a] of eXist core.
$EXIST_HOME/conf.xml	eXist's configuration file.
$EXIST_HOME/log4j.xml	eXist's log4j logging configuration file.

[a]An accidental requirement of eXist 2.0 and 2.1, which will no longer be needed as a dependency for embedded operation in future versions.

XML:DB Local API

Unlike the XML:DB Remote API, which sends data back and forth across the network to eXist using the XML-RPC protocol, the Local API instead talks directly to eXist's internal Java API through function calls within the same JVM process. It is relatively trivial to switch your code between the local and remote modes of operation of the XML:DB API, so if you want to learn a single API and are unsure of which to choose or wish to use both local and remote modes, it can be a good choice. In addition, as the XML:DB API is standardized, you could potentially use it to talk to other XML document systems as well as eXist.

In addition to the runtime dependencies set out in Table 13-8, you will also need the one in Table 13-9.

Table 13-9. Additional dependency for using the XML:DB Local API

Library	Description	Scope
$EXIST_HOME/lib/core/xmlrpc-client-3.1.3.jar	Apache XML-RPC client library. Dependency of eXist's XML:DB API, even when using XML:DB local mode!	Runtime

The XML:DB Local API is almost identical to the XML:DB Remote API (see "XML:DB Remote API" on page 349), so we will only discuss where it differs from the Remote API. Therefore, reading "XML:DB Remote API" on page 349 first should be considered a requirement for understanding the Local API. Differences you need to be aware of are:

Collections

Conceptually, collections in the Local API are exactly the same as in the Remote API; the only difference is the URI format that you use to access them. As the database is running in the same JVM there is no remote server, so you need *not* provide the server name, port, or endpoint in the URI. Instead, you just need the collection path. See Example 13-18.

Example 13-18. Opening an XML:DB local collection to eXist

```
Collection collection =
    DatabaseManager.getCollection("xmldb:exist://db", "admin", "");
```

Database shutdown

When you first access an embedded database collection, an eXist embedded database is automatically started for you. To maintain database consistency, *you are responsible for cleanly shutting down the eXist database* either before your application terminates or when you have finished with the database inside your application. See Example 13-19.

Example 13-19. Shutting down eXist with the XML:DB Local API

```
final DatabaseInstanceManager manager =
     (DatabaseInstanceManager) coll.getService("DatabaseInstanceManager", "1.0");
try {
     coll.close();
} finally {
     manager.shutdown();
}
```

Here's a tip: to ensure that the database is always shut down during normal operation of your application, it is recommended that you set up and tear down the database using a `try/finally` block. That is to say, all of your database interaction should take place inside an encapsulating `try` block, and your final collection `close` and subsequent Database Instance Manager `shutdown` call should both happen inside the same `finally` block that corresponds to the initial `try` block. See Example 13-20.

Example 13-20. Ensuring clean shutdown when using the XML:DB Local API

```
Collection coll = null;
try {
    coll =
         DatabaseManager.getCollection("xmldb:exist:///db" username, password);

    //TODO all of your database interaction code is called from here

} finally {
    if(coll != null) {
        final DatabaseInstanceManager manager =
            (DatabaseInstanceManager) coll.getService(
            "DatabaseInstanceManager", "1.0");
        try {
              coll.close();
        } finally {
              manager.shutdown();
        }
    }
}
```

Example

The source code of a small example of using the XML:DB Local API from Java to store a file, query the database, and remove a file is included in the folder *chapters/integration/xmldb-embedded* of the *book-code* Git repository (see "Getting the Source Code" on page 15).

To compile the example, enter the *xmldb-embedded* folder and run `mvn package`.

XML:DB local example. You can then execute the ExampleApp example like so:

```
java -jar xmldb-embedded-example/target/xmldb-embedded-example-1.0-example.jar
```

This shows the available arguments for using the ExampleApp.

A complete example of using the application might look like the following:

```
java -jar xmldb-embedded-example/target/xmldb-embedded-example-1.0-example.jar
  /db/my-new-collection /tmp/test.xml "//thing" admin
```

Given the preceding arguments, the example application would perform the following steps:

1. Start up the embedded eXist database.
2. Get a reference to the collection */db/my-new-collection* in eXist (the collection will be created if it does not already exist).
3. Upload the file */tmp/test.xml* to the collection */db/my-new-collection* in eXist (again, the collection will be created if it does not already exist).
4. Execute the query `//thing` against the */db/my-new-collection* collection, and print the results.
5. Remove the */db/my-new-collection/test.xml* file.
6. Shut down the eXist database.

Fluent API

The Fluent API was developed by PiotrKaminski and contributed to the eXist project in 2007. The Fluent API has the goal of making it much simpler to use eXist from within your own Java applications as an embedded database. It follows the design principle of a Fluent interface, which should make Java code interacting with eXist more readable. The current best source of Fluent API documentation is within the Fluent API JavaDocs.

The Fluent API, just like the XML:DB Local API, talks directly to eXist's internal Java API through function calls within the same JVM process.

In addition to the runtime dependencies set out in Table 13-8, you will also need the one listed in Table 13-10.

Table 13-10. Additional dependency for using the Fluent API

Library	Description	Scope
$EXIST_HOME/lib/extensions/exist-fluent.jar	Fluent API Library	Compile

There are just four main concepts that you need to understand in the Fluent API to interact with eXist:

`Database`*s*

The Fluent API makes use of `Database` to represent a distinct connection by a user to an embedded eXist instance. Connecting to a database requires authentication, after which all subsequent operations on that database and its folders or documents use the same credentials. Typically, you will work with a single `Database` instance. See Example 13-21.

Example 13-21. Starting an eXist embedded instance with the Fluent API

```
Database.startup(new File("conf.xml"));
Database db = Database.login("admin", "");
```

`Folder`*s*

A `Folder` in the Fluent API maps onto a collection in the eXist database. Folders are the primary means for interacting with the Fluent API. From a folder you may manage subfolders and documents. See Example 13-22.

Example 13-22. Getting a Folder reference from the Fluent API

```
Folder folder = db.getFolder("/db");
```

`Document`*s*

A `Document` in the Fluent API maps onto a document in the eXist database. The Fluent API allows you to work with both eXist's XML and binary documents. Binary documents will be of type `org.exist.fluent.Document`, and XML documents are of a subtype of that, `org.exist.fluent.XMLDocument`. See Example 13-23.

Example 13-23. Retrieving a Document from the Fluent API

```
Document doc = folder.documents().get("some-document.xml");
```

`QueryService`*s*

A `QueryService` in the Fluent API allows you to execute XQueries against folders or documents in the database. A `QueryService` may be retrieved from a database, a folder, or a document object (if you want finer-grained control over the query context). The result of executing a query with the `QueryService` is an instance of `org.exist.fluent.ItemList`, which you may iterate over to obtain individual `org.exist.fluent.Item` instances, from which you may then retrieve a result value. See Example 13-24.

Example 13-24. Querying a folder with the Fluent API

```
ItemList results = folder.query.all("//my-node");

for(Item result : results) {
  System.out.println(result.value);
}
```

Example

The source code of a small example of using the Fluent API from Java to store a file, query the database, and remove a file is included in the folder *chapters/integration/fluent-embedded* of the *book-code* Git repository (see "Getting the Source Code" on page 15).

This example is a direct port of the XML:DB Local API example from the previous section, and hopefully will allow you to easily compare the code of the two approaches and decide which you prefer.

To compile the example, enter the *fluent-embedded* folder and run `mvn package`.

Fluent API example. You can then execute the ExampleApp example like so:

```
java -jar fluent-embedded-example/target/fluent-embedded-example-1.0-example.jar
```

This shows the available arguments for using the ExampleApp.

A complete example of using the application might look like the following:

```
java -jar fluent-embedded-example/target/fluent-embedded-example-1.0-example.jar
  /db/my-new-collection /tmp/test.xml "//thing" admin
```

Given the preceding arguments, the example application would perform the following steps:

1. Start up the embedded eXist database.
2. Get a reference to the folder (collection) */db/my-new-collection* in eXist (the folder will be created if it does not already exist).
3. Upload the file */tmp/test.xml* to the folder */db/my-new-collection* in eXist (again, the folder will be created if it does not already exist).
4. Execute the query `//thing` against the */db/my-new-collection* folder, and print the results.
5. Remove the */db/my-new-collection/test.xml* file.
6. Shut down the eXist database.

CHAPTER 14
Tools

Working with eXist often means working with various tools that help ease development. For instance, writing code in an integrated development environment (IDE) like eXide or oXygen is far easier than hacking in Notepad or vi. Other tools, such as the eXist Ant extensions, are useful for automating tasks.

This chapter explores some of the common tools that you can use to work with eXist. Undoubtedly there are many more, so be warned: our personal preferences are shining through!

Java Admin Client

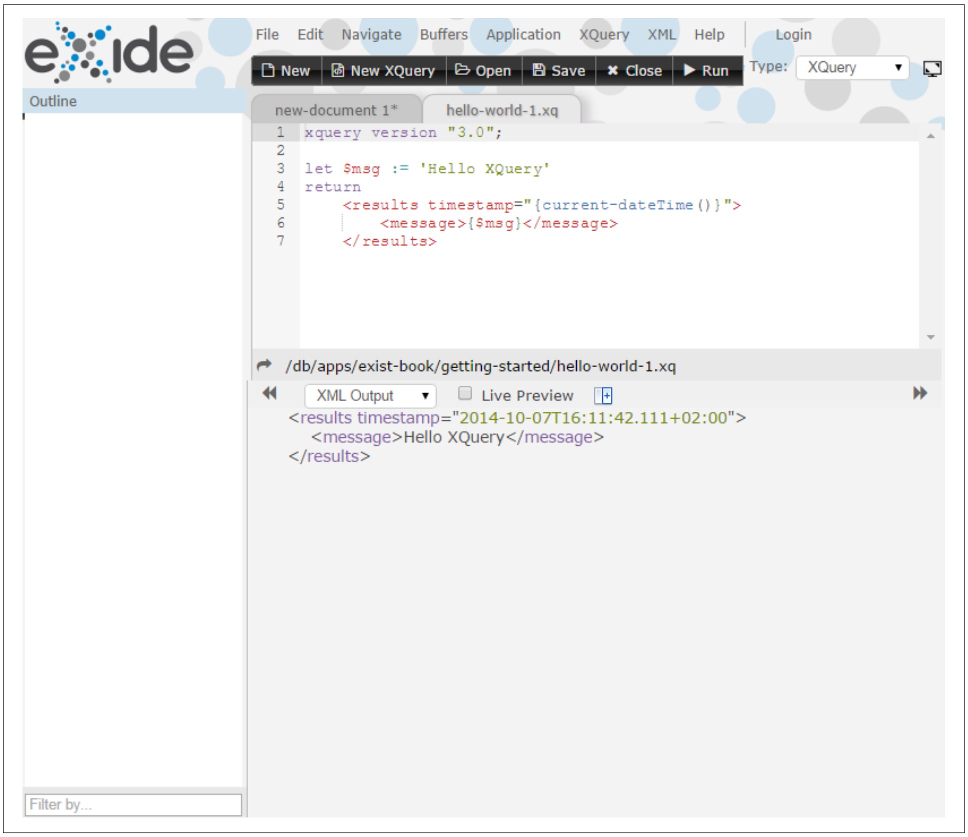
eXist comes with a small program called the Java Admin Client. It allows you to do maintenance work on the database like backups and restores, imports and exports, checking and setting properties, creating collections, and more. It's a standard Java Swing GUI application and its functionality mostly speaks for itself.

The Java Admin Client has a lot of functionality hidden underneath right mouse clicks, which is very useful to know if you can't immediately figure out how to do things.

The Java Admin Client has already popped up in this book in several places. See, for instance, "The Java Admin Client" on page 29, and "User and Group Management with the Java Admin Client" on page 145.

eXide

One of the additional packages you can install in your eXist installation is eXide (Figure 14-1). eXide is a complete development environment for working with eXist. You can do a lot with eXide, from editing a single file (as long as it's stored in the database) to creating and maintaining complete applications.



```
xquery version "3.0";

let $msg := 'Hello XQuery'
return
    <results timestamp="{current-dateTime()}">
        <message>{$msg}</message>
    </results>
```

/db/apps/exist-book/getting-started/hello-world-1.xq

```
<results timestamp="2014-10-07T16:11:42.111+02:00">
  <message>Hello XQuery</message>
</results>
```

Figure 14-1. The eXide IDE

Some interesting features of eXide are:

Application outline
: The lefthand pane shows you an outline of your application/module and lets you jump to a declaration with a single click.

Content completion
: Type the beginning of what you want (e.g., a function name or a variable) and press Ctrl-Space (or Cmd-Space on a Mac) to see a list of possible completions.

Refactoring
: Highlight a block of code and press Ctrl-Shift-X (or Cmd-Ctrl-X on a Mac) to extract the block into a function. You can also extract a code block into a variable by pressing Ctrl-Shift-E (or Cmd-Ctrl-E on a Mac).

Quick function navigation
: Put your cursor in a function name and press Ctrl-F3 (or Cmd-F3 on a Mac) to go to its definition, even if it's in another module.

Inserting code snippets
: There is a variety of easily inserted code snippets available. For instance, type **`for-return`** and press Ctrl-Space (or Cmd-Space on a Mac). A basic `for $item in $inseq return $item` code snippet will pop up. Press Tab to select all the variables in this template one by one, so you can easily replace them with your own code (press Esc to stop this behavior). You'll find the available code snippets in */db/apps/eXide/templates/snippets.xml*.

Working with applications
: The Application menu gives you access to working with complete applications. You can create a shiny new one from a template, synchronize it with a version on disk, download it into an eXist package file (see "Packaging" on page 227), and more.

One of the features of eXide is that you can jump-start development of an application. Choose "New application" from the Application menu and fill in the forms, and eXide will generate an application framework for you. If you've installed the eXist documentation package, information about this is available at *http://localhost:8080/exist/apps/doc/development-starter.xml*.

The generated applications use what is called the *HTML Templating Framework*. This framework lets you separate out:

- The page template (the surrounding HTML code that defines the fixed parts of the page)
- The body of the page
- The code that generates the dynamic parts of the page

For further information, all of this is documented in *http://localhost:8080/exist/apps/doc/templating.xml* on your local installation, or in the eXist documentation online.

oXygen

The oXygen XML Editor is likely the most popular XML IDE in professional XML circles. It provides extensive support for editing XML and related files in general;

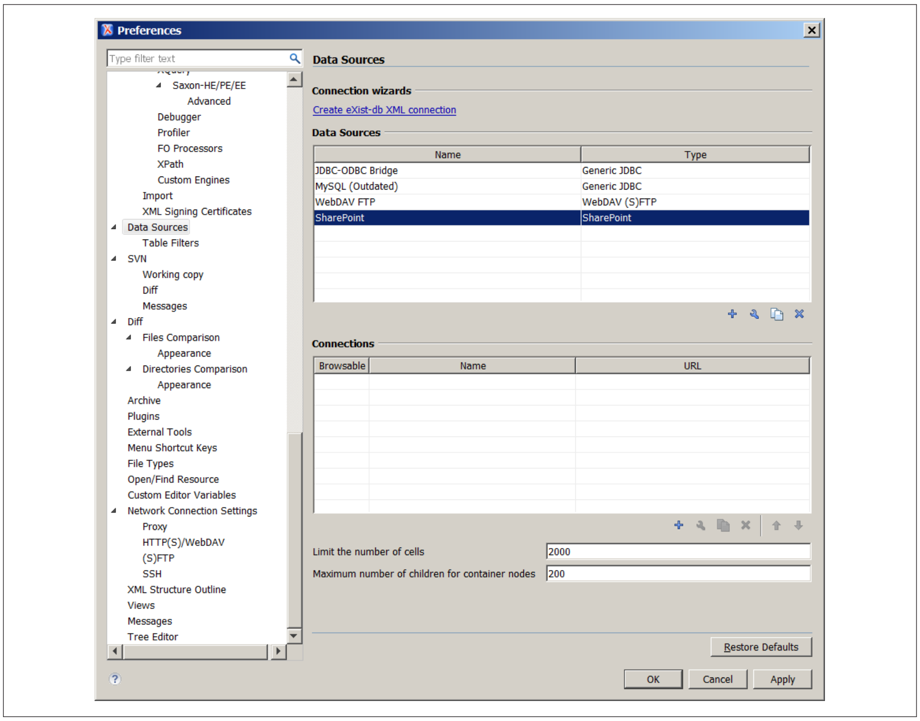
adds features like visual editing of schemas and author mode (for those who don't want to see angle brackets); and can connect to databases, both SQL and NoSQL, including eXist. While oXygen is a commercial product, you can request a free trial license to try it out before purchasing.

A large proportion of this book was actually written and edited using oXygen version 15.0. As these things go, the information here might already be outdated because a newer version of oXygen may have been released since this book's publication. If you can't relate what we say here to what you see on your screen, please refer to the corresponding oXygen documentation.

Connecting with oXygen Using WebDAV

The most basic way of connecting oXygen with eXist is by using WebDAV. It allows you to work with the collections and resources in the database as if they were directories and files on the filesystem. To open a WebDAV connection, do the following:

1. Go to the Options→Preferences menu and look for the Data Sources section. You'll see something like Figure 14-2.



Data Sources

Connection wizards

Create eXist-db XML connection

Data Sources

Name	Type
JDBC-ODBC Bridge	Generic JDBC
MySQL (Outdated)	Generic JDBC
WebDAV FTP	WebDAV (S)FTP
SharePoint	SharePoint

Connections

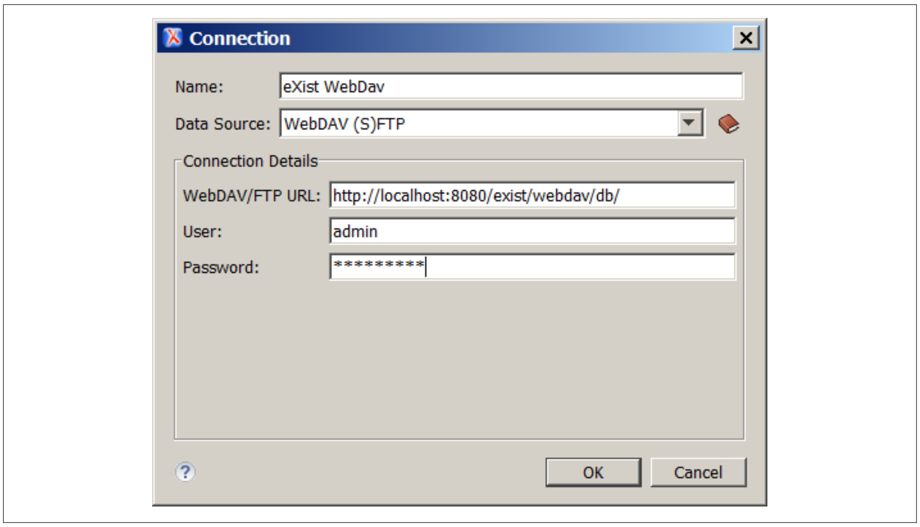
Browsable	Name	URL

Limit the number of cells 2000

Maximum number of children for container nodes 200

Figure 14-2. The virgin oXygen Data Sources dialog

2. Click the + icon under the Connections table and fill in the dialog, choosing WebDAV (S)FTP as the data source. The URL to use in a default installation is *http://localhost:8080/exist/webdav/db/*. When you're developing, it's probably easiest to connect as the `admin` user, as shown in Figure 14-3.



Name: eXist WebDav
Data Source: WebDAV (S)FTP
WebDAV/FTP URL: http://localhost:8080/exist/webdav/db/
User: admin
Password: *********

Figure 14-3. Configuring a WebDAV connection in oXygen

3. Click the OK button on all dialogs and open the Data Source Explorer (the Window→Show View→Data Source Explorer menu item). And hey presto! There is your eXist database, exposed to you in all its beautiful detail.

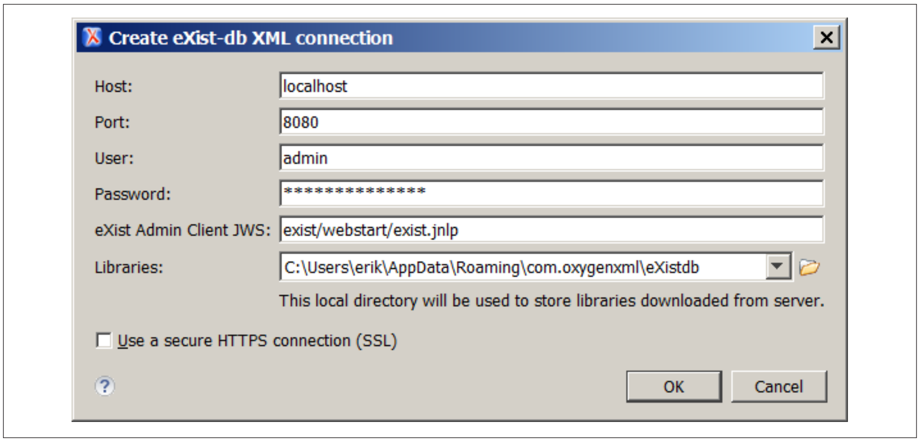
Natively Connecting with oXygen

Besides going in through WebDAV, you can also connect oXygen to eXist natively. This gives you many benefits, like the ability to get lists of available extension module functions in your editor, validate queries with eXist (eXist's XQuery dialect is checked, not straight XQuery), and even execution of XQuery files directly from within the editor.

Setting up a native connection with eXist is extremely simple:

1. Go to the Options→Preferences menu and look for the Data Sources section. Refer back to Figure 14-2.
2. Click the link at the top of the dialog box labeled "Create eXist-db XML connection."

3. For a default setup (see Figure 14-4), you only have to change the user to `admin` and fill in the password. Then click the OK button.



Create eXist-db XML connection
Host: localhost
Port: 8080
User: admin
Password: **************
eXist Admin Client JWS: exist/webstart/exist.jnlp
Libraries: C:\Users\erik\AppData\Roaming\com.oxygenxml\eXistdb
This local directory will be used to store libraries downloaded from server.
Use a secure HTTPS connection (SSL)
OK
Cancel

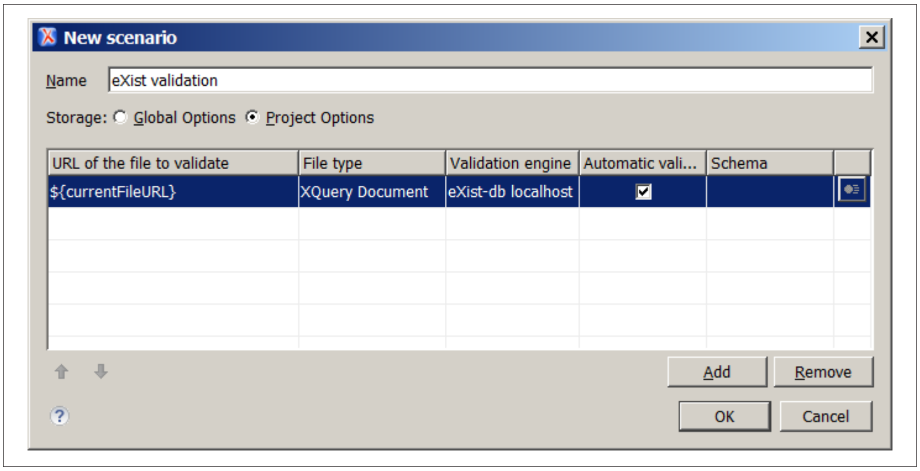
Figure 14-4. Adding an eXist-db XML connection to oXygen

As you can see when you're back on the Data Sources setup screen, oXygen created not only a data source for eXist but also an accompanying connection. So, we're ready to go!

4. Click the OK button on all dialogs and open the Data Source Explorer (the Window→Show View→Data Source Explorer menu item). And there it is: your native eXist connection, also called `eXist-db localhost`.

Now, this looks like you've opened another WebDAV connection to eXist. However, a native connection allows you to do much more (in describing this, we assume you know the basics of oXygen, like creating validation and transformation scenarios; if not, refer to its excellent help facilities):

- As soon as you've created an eXist connection, you'll notice your XQuery editor has become more intelligent: for instance, type **`xmldb:`** and press Ctrl-Space, and the list of eXist XMLDB XQuery extension module functions pops up.
- You can validate an XQuery file using the eXist validator instead of the Saxon one used by default in oXygen. This gives you the benefit of the validator knowing, for instance, all eXist XQuery extension functions and so it will not show your file as invalid when it makes use of those. To do this, open an XQuery file and create a validation scenario that uses the eXist native connection you just created as its validation engine (see Figure 14-5).


New scenario

Name: eXist validation

Storage: Global Options [✓ Project Options]

URL of the file to validate	File type	Validation engine	Automatic vali...	Schema
${currentFileURL}	XQuery Document	eXist-db localhost	✓	

Figure 14-5. Creating a validation scenario for use with eXist

- To execute an XQuery script directly from oXygen, create a transformation scenario that uses your eXist connection as its transformer. Transform the XQuery script using this scenario and it will run inside eXist, returning the results to oXygen.

Ant and eXist

Many Java programmers know the Ant build tool. It is used for automating build processes, including maintenance work like creating files and directories, as well as creating zip files.

eXist contains a library that extends Ant so you may work with the database from inside your Ant scripts. This is useful for automating common tasks like backups, restores, and data import/export. More advanced capabilities include running XQueries and user management.

For simple scripts, Ant and the eXist Ant extensions are sufficient. However, if you need more complicated functionality, like iterating over collections and resources, you also need Ant-Contrib, a library containing generic extensions for common Ant functionality.

Trying the Ant Examples

There is an example *build.xml* script in the folder *chapters/tools* of the *book-code* Git repository (see "Getting the Source Code" on page 15). It contains several targets that illustrate what you can do with Ant and eXist together. To try it, do the following:

1. Make sure you have Ant installed and that the `ant` command is in your `path` (type **`ant -version`** to check).
2. Open the *build.xml* file and check if the properties listed under the heading `BASE INFORMATION` apply to your situation. You will have to change at least the `admin` password property for the examples to work.
3. Open a command window and navigate to the directory where you stored the *build.xml* file.
4. The command `ant -p` will give you an overview of the targets defined.
5. `ant` *`targetname`* will execute a target (e.g., `ant ListMainCollections`).

Preparing an eXist Ant Build Script

If you would like to use the eXist Ant extensions inside your Ant build script, follow these steps:

1. Define the `http://exist-db.org/ant` namespace. The recommended namespace prefix is `xdb`. The easiest way to do this is to add the namespace definition to the root `project` element of your Ant build file (usually called *build.xml*):

   ```
   <project xmlns:xdb="http://exist-db.org/ant">
   ```

2. Before you use the extension in your build script, you must tell Ant where it can find the eXist extension libraries. The following code example assumes that on the operating system level the `EXIST_HOME` environment variable is set and points to the directory where you have installed eXist:

   ```
   <property environment="Env"/>
   <fail unless="Env.EXIST_HOME">Environment variable EXIST_HOME not set
   </fail>

   <path id="classpath.core">
     <fileset dir="${Env.EXIST_HOME}/lib/core">
       <include name="*.jar"/>
     </fileset>
     <pathelement path="${Env.EXIST_HOME}/exist.jar"/>
     <pathelement path="${Env.EXIST_HOME}/exist-optional.jar"/>
     <pathelement path="${Env.EXIST_HOME}"/>
   </path>
   ```

3. If you want to use Ant-Contrib as well, add the following code:

   ```
   <taskdef resource="net/sf/antcontrib/antlib.xml">
     <classpath>
       <pathelement
        location="$EXIST_HOME/tools/ant/lib/ant-contrib-1.0b3.jar"/>
   ```

```
        </classpath>
      </taskdef>
```

All eXist's Ant tasks share the following attributes, so you might want to put the values in Ant properties:

`uri`
: This must be an XMLDB URI (see "XMLDB URIs" on page 92) that points to the database and collection you want to work with. For example, to point to the main */db* collection for a default eXist installation:

    ```
    xmldb:exist://localhost:8080/exist/xmlrpc/db
    ```

`user` *and* `password`
: The credentials of the eXist user for accessing the database. You will most likely want to use an account with `admin` privileges. If you don't specify these attributes, the default `guest` account will be used.

`failonerror`
: Whether or not an error should stop the build script (default: `true`).

The first three attributes are the most important ones. For example, at the top of the *build.xml* file in the accompanying book example code, you'll find properties defined for them:

```
<property name="BaseUri" value="xmldb:exist://localhost:8080/exist/xmlrpc/db"/>
<property name="Username" value="admin"/>
<property name="Password" value="secret"/>
```

Using Ant with eXist

The eXist Ant extension contains a large number of tasks to work with the database. We're not going to list them all here as they are already well documented in the eXist online documentation. Instead, we will leave you with a few tantalizing examples. All examples given assume the preparations described in the previous section are in the script also. These will not be repeated in every listing.

Basic example: Listing the main collections

The following basic (and probably un-useful) example serves as the "Hello World" into the Ant extension—it lists the collections on the */db* level:

```
<target name="ListMainCollections" description="Lists all collection in /db">
  <xdb:list uri="${BaseUri}"
      user="${Username}" password="${Password}" collections="true"
      outputproperty="Collections"/>
  <echo>Main collections: ${Collections}</echo>
</target>
```

Backup and shutdown

This target creates a full backup of your database, zips it, and then shuts down eXist:

```
<target name="BackupShutdown"
 description="Back up the full database and shut down">
  <delete dir="backup"/>
  <mkdir dir="backup"/> ❶
  <xdb:backup uri="${BaseUri}"
   user="${Username}" password="${Password}" dir="backup"/> ❷
  <zip destfile="backup.zip" basedir="backup"/> ❸
  <delete dir="backup"/> ❹
  <xdb:shutdown uri="${BaseUri}" user="${Username}" password="${Password}"/> ❺
</target>
```

From top to bottom, this code does the following:

❶ Makes sure we have an empty *backup* directory by first deleting and then (re)creating it.

❷ Makes eXist back up into this directory. If you look inside after the backup you'll find a directory/file structure much like your database's collection/resource structure. Added are *__contents__.xml* files that contain important eXist properties (like security settings).

❸ Zips this directory into a *backup.zip* file (this is easier to handle, and you can restore directly from such a ZIP file).

❹ Removes the *backup* directory (since everything is in the ZIP now)

❺ Shuts down eXist.

The preceding example performs a full database backup, but you can just as easily make partial backups—for instance, from your extremely important project collection. Simply have the `uri` attribute point to the right collection, like so:

```
<xdb:backup uri="${BaseUri}/apps/myimportantproject" ...
```

Of course, there is also an `xdb:restore` task that lets you restore a backup. This task can use a backup ZIP file directly (no need to unpack it first).

Create separate backups for all subcollections

The following example shows you how to iterate over lists returned by some of eXist's Ant tasks using the Ant-Contrib extension. It extends the previous example by not creating a full backup, but a separate backup of each subcollection of */db*:

```
<target name="SeparateBackups"
    description="Make separate backups of all subcollections of /db">
  <xdb:list uri="${BaseUri}" user="${Username}" password="${Password}"
```

```
        collections="true" outputproperty="SubCollections"/>
    <echo>Subcollections to backup: ${SubCollections}</echo>
    <foreach list="${SubCollections}" param="SubCollection"
     target="BackupCollection"/>
  </target>

  <target name="BackupCollection">
    <echo>Backup of /db/${SubCollection}</echo>
    <property name="BackupTempDir" value="backup-${SubCollection}"/>
    <delete dir="${BackupTempDir}"/>
    <mkdir dir="${BackupTempDir}"/>
    <xdb:backup uri="${BaseUri}/${SubCollection}" user="${Username}"
        password="${Password}" dir="${BackupTempDir}"/>
    <zip destfile="backup-${SubCollection}.zip" basedir="${BackupTempDir}"/>
    <delete dir="${BackupTempDir}"/>
  </target>
```

Ant-Contrib's `foreach` task iterates over a comma-separated list of values and then calls the `BackupCollection` task for each one of them.

Run an XQuery from Ant

You can run an XQuery script from within Ant and return the results in a property. For instance:

```
  <target name="RunXQuery" description="Run an XQuery from Ant">
    <xdb:xquery uri="${BaseUri}" user="${Username}" password="${Password}"
        outputproperty="QueryOutput" query="system:get-exist-home()"/>
    <echo>Query result: ${QueryOutput}</echo>
  </target>
```

This little task will run the query in the `query` attribute and, in this case, return the home directory of eXist. If your query gets more complicated, it will probably be easier to store it in a separate file and run it from there. You can accomplish this using the standard Ant `loadfile` task:

```
  <loadfile property="XQueryScript" srcFile="myscript.xq"/>
  <xdb:xquery uri="${BaseUri}" user="${Username}" password="${Password}"
      outputproperty="QueryOutput" query="${XQueryScript}"/>
```

CHAPTER 15
System Administration

When you're using eXist day to day and eventually deploying it into a production environment, there are several topics that are useful to understand, both for those in a developer role and (even more so) for those in a system administration or DevOps role. In this chapter, we'll look at some of the tools provided by or used with eXist that aid in ensuring stable and performant operation.

Logging

eXist uses Apache log4j as its mechanism for logging information and issues. Log4j provides a logging hierarchy that is configurable without changing the eXist code. Understanding how log4j works and how to configure it can help you get the most out of your eXist logfiles. Log4j provides several levels of logging. When eXist wishes to log a message, it decides at which level to log the message, and your log4j configuration then decides how that message is handled. The levels that log4j provides and at which eXist logs various messages are displayed in Table 15-1, with the most fine-grained at the top.

Table 15-1. Log4j logging levels

Level	Description
Trace	Used for tracing the execution of complex parts of the eXist code. It is very unlikely that you will need to receive log messages at this level, unless debugging a serious issue.
Debug	Similar to Trace, but less fine-grained. Again, it is very unlikely that you will need to receive log messages at this level unless you are investigating an issue with eXist.
Info	Used for logging status information about eXist for the information of users and system administrators. This is the default logging level in eXist.

Level	Description
Warn	Used for reporting unexpected or nonoptimal behavior that is noncritical; that is, your operation or query will most likely still perform correctly.
Error	Used for reporting errors while performing a database or query operation. These messages indicate that something failed and the user or system administrator may need to take action.
Fatal	Used for reporting critical failures within eXist. Rarely used in eXist, but can report a corruption in the database or an index.

Levels in log4j are inherited upward, meaning that logging at the most fine-grained Trace level will log all message levels (i.e., also the Debug, Info, Warn, Error, and Fatal levels); likewise, logging at the Info level would actually log messages from the Info, Warn, Error, and Fatal levels.

In eXist the log4j configuration file is *$EXIST_HOME/log4j.xml*, and it is configured to log at the Info level by default. If you wish to adjust this, you can change the priority levels in the log4j configuration file as follows:

```
<root>
  <priority value="debug"/>
  <appender-ref ref="exist.core"/>
</root>
```

Log4j allows you to direct log messages from different parts of the database to different receivers. By default, in eXist all of the receivers are files, and the logfiles are written into the directory *$EXIST_HOME/webapp/WEB-INF/logs*. See Table 15-2.

Table 15-2. eXist logfiles

Logfile	Description
exist.log	The main logfile of eXist; all messages that are not directed to any of the other logfiles end up here. It contains details of the database server, the database and index status and health, and XQuery execution.
xacml.log	The logfile for the XACML (eXtensible Access Control Markup Language) engine in eXist. XACML support in eXist is deprecated.
xmldb.log	The logfile for XML:DB API operations; typically these log messages also appear in *exist.log*.
urlrewrite.log	The logfile for the XQuery URL rewriting engine. When you are developing XQuery apps that use URL rewriting, it can be useful to study both *exist.log* and this file when you have issues.
profile.log	If you switch on XQuery profiling using `util:enable-profiling` or the profiling pragma, the results of the profiling can be logged to this file at the Trace level. This approach is deprecated in favor of the profiling tool in the Admin Web Application (see "Checking Index Usage" on page 282).
scheduler.log	This file logs messages related to scheduled tasks in eXist. These include the database flush-to-disk tasks and also your own XQuery or Java scheduled jobs. If you are having trouble running your own scheduled jobs, this is the place to look for feedback.

Logfile	Description
ehcache.log	When using Ehcache with eXist, the Ehcache log messages are redirected to this file. Ehcache in eXist at this time is not recommended for general use.
betterform.log	When using betterFORM as your XForms engine in eXist, the log messages of betterFORM are redirected to this logfile. It can be useful for assisting in developing and debugging XForms applications.
restxq.log	If you are using RESTXQ in eXist, then all operations of the RESTXQ framework and server are logged here, including any issues with compiling your XQuery. Note that when your XQuery runs, any log messages it produces will be sent to *exist.log*.
backup.log	All log messages related to creating database backups or restoring database backups are written to this file.
mdStorage.log	If you are using the new `metadata` storage module, then any log messages generated by this module will be written to this file. For more information on the `metadata` module, see `metadata`.

The logfiles in eXist are all plain text, so you may view them with any plain-text editor. If you prefer a GUI tool with colored highlighting of the log levels, then Apache Chainsaw may be worth a look.

On Unix/Linux/Mac systems, you can monitor log messages by tailing and following the appropriate logfile. For example:

```
tail -f $EXIST_HOME/webapp/WEB-INF/logs
```

On Windows systems, you can configure log4j to log to the Windows Event Log if you wish by using `org.apache.log4j.nt.NTEventLogAppender`. See the log4j documentation for the exact configuration details.

JMX

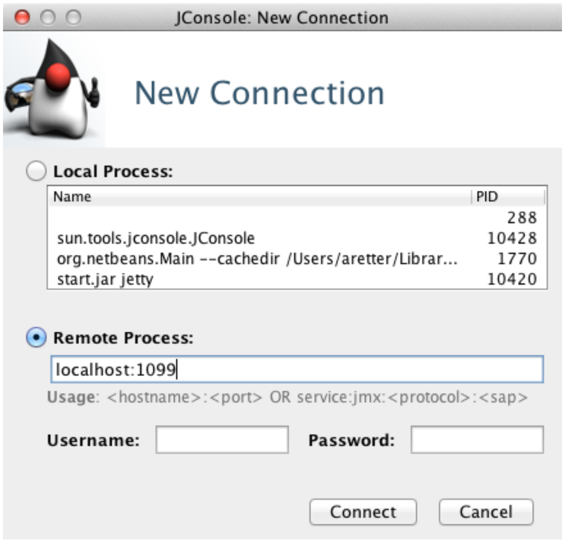
Java Management Extensions (JMX) is a Java technology that allows an application to expose monitoring information and management options to other applications. eXist acts as a JMX server and can service requests from any JMX client. At present, eXist mainly exposes monitoring information and very little in the way of management services via JMX. To use eXist with JMX, you must enable and configure JMX via options passed to the JVM when you start eXist.

If you are using either *$EXIST_HOME/startup.sh* or *$EXIST_HOME/startup.bat* for starting eXist, you can just pass the additional `-j` argument with a TCP port number for the JMX server to listen on. For example:

```
$EXIST_HOME/startup.sh -j 1099
```

When you enable JMX using the `-j` setting in eXist, it disables JMX authentication and does not transport JMX over SSL. Thus, this approach should be used only in a secure, controlled environment! It is perfectly possible to use eXist's JMX with authentication and SSL. For details of the JVM options, refer to the JMX documentation and make the necessary changes to either *$EXIST_HOME/bin/batch.d/check_jmx_status.bat* on Windows, or *$EXIST_HOME/bin/functions.d/jmx-settings.sh* on other platforms.

You can then connect using any JMX client, such as the simple command-line client provided with eXist or JConsole (which is provided with your JDK and shown in Figure 15-1). The eXist documentation for JMX is itself very reasonable, and rather than reproduce it here, we recommend you consult it for further information.


JConsole: New Connection

New Connection

Local Process:

Name	PID
	288
sun.tools.jconsole.JConsole	10428
org.netbeans.Main --cachedir /Users/aretter/Librar...	1770
start.jar jetty	10420

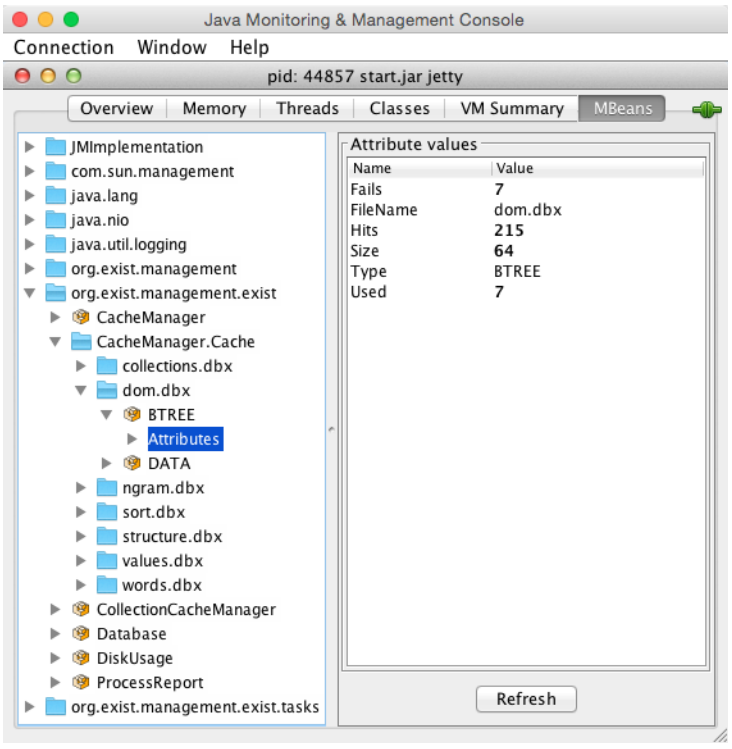
[✓ Remote Process:] localhost:1099

Usage: <hostname>:<port> OR service:jmx:<protocol>:<sap>

Username:
Password:

Figure 15-1. Connecting to eXist's JMX server using JConsole

When you are browsing eXist with JConsole, it is not always obvious how to view the monitoring information. As Figure 15-2 shows, you need to select the collection of attributes from the navigation tree under the service that you are interested in.



Name	Value
Fails	7
FileName	dom.dbx
Hits	215
Size	64
Type	BTREE
Used	7

Figure 15-2. Examining eXist's cache utilization using JConsole

Memory and Cache Tuning

The total memory available to eXist is set through the Java JVM's `-Xmx` setting. This setting determines the maximum heap size available to a Java application when a JVM is started. If a Java application attempts to use more memory than is available, it receives the dreaded `OutOfMemoryError` from the JVM. While an application may not crash immediately, it is more than likely impossible to continue running successfully after receiving an `OutOfMemoryError`!

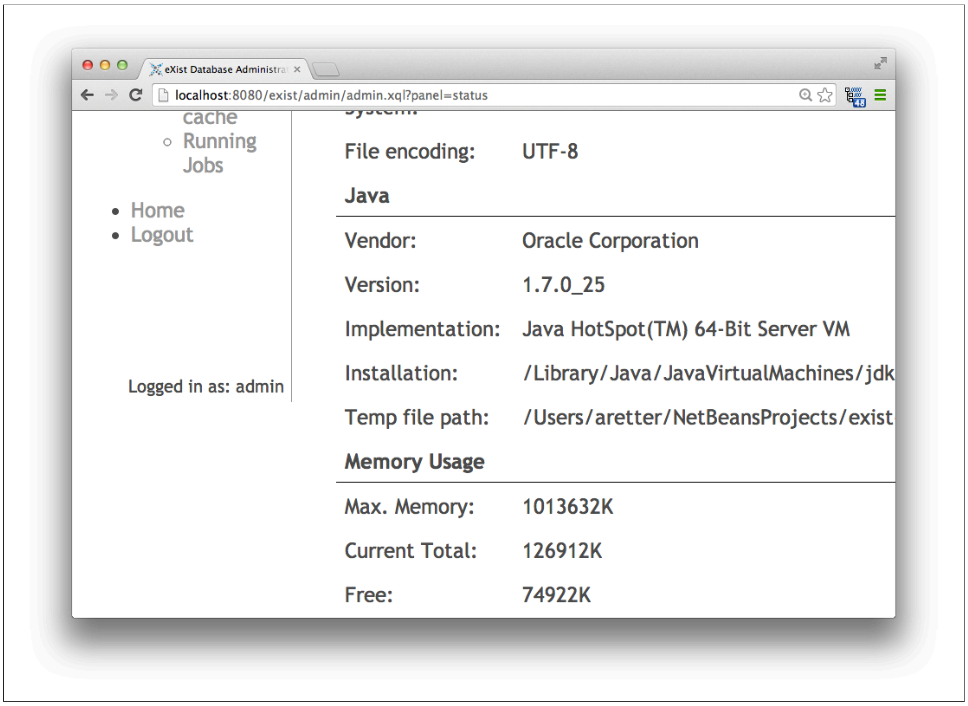
In eXist, all memory is allocated on the heap, so it is important to ensure that you have enough memory allocated for your database. It is very difficult for us to guide you in establishing the best `-Xmx` setting, as every dataset and query workload is different. If you used the eXist installer, then you will have configured the maximum memory available to eXist during installation (see "Things to Decide Before Installing" on page 20). If you did not use the installer, the `-Xmx` setting will be present in the script that you use to start up eXist.

Understanding Memory Use

As well as maintaining the stability of your database, ensuring you have enough memory available also has an impact on the performance of your queries. If there is not enough memory available, then queries may run slowly as old objects are garbage-collected by the JVM. Because your database size and query profiles may change over time, it is important to monitor eXist's memory use. Unfortunately, monitoring a JVM process with tools provided by your operating system (such as *taskmgr.exe* on Windows or *top* on Unix/Linux) rarely gives you a detailed understanding of Java memory usage. However, there are several tools available for monitoring memory use of a JVM and eXist.

Web Admin Status

The somewhat antiquated Web Admin Application shipped with eXist provides a quick overview at the bottom of its status page, which you can access from *http://localhost:8080/exist/admin/admin.xql?panel=status* (see Figure 15-3).


eXist Database Administra
localhost:8080/exist/admin/admin.xql?panel=status
cache
Running Jobs
Home
Logout
Logged in as: admin

File encoding:	UTF-8
Java	
Vendor:	Oracle Corporation
Version:	1.7.0_25
Implementation:	Java HotSpot(TM) 64-Bit Server VM
Installation:	/Library/Java/JavaVirtualMachines/jdk
Temp file path:	/Users/aretter/NetBeansProjects/exist
Memory Usage	
Max. Memory:	1013632K
Current Total:	126912K
Free:	74922K

Figure 15-3. eXist Web Admin Status page

At first glance, the memory status reported by the Web Admin Status page may look confusing. This is in part because it is a direct reflection of how Java reports and allocates its memory; however, once you understand how to interpret it, it is relatively simple. These figures are the result of calling some XQuery functions from eXist's `system` module; to understand how to interpret these, see "XQuery" on page 391.

XQuery

eXist provides three XQuery functions that may be used to interrogate the JVM in its `system` extension module (see `system`). The functions are:

`system:get-memory-max`
: Reports the maximum memory available to the JVM running eXist (i.e., the value of the `-Xmx` setting).

`system:get-memory-free`
: Reports memory that is allocated, but free and available for reuse.

`system:get-memory-total`
: Reports the currently allocated memory within the JVM. This is made up of both memory that is in use and memory that is free for reuse. Subtracting `system:get-memory-free` from this tells you exactly how much memory eXist is using.

The following simple XQuery reports on the memory status:

```
xquery version "3.0";

declare function local:human-units($bytes) {
    let $unit := if($bytes > math:pow(1024, 3)) then
        (math:pow(1024, 3), "GB")
    else if($bytes > math:pow(1024, 2)) then
        (math:pow(1024, 2), "MB")
    else
        (1024, "KB")
    return
        format-number($bytes div $unit[1], ".00") || " " || $unit[2]
};

<memory>
    <max>{local:human-units(system:get-memory-max())}</max>
    <allocated>
        <in-use>{
            local:human-units(
                system:get-memory-total()
                - system:get-memory-free()
            )
        }</in-use>
        <free>{local:human-units(system:get-memory-free())}</free>
        <total>{local:human-units(system:get-memory-total())}</total>
    </allocated>
    <available>{
        local:human-units(
            system:get-memory-max()
            - system:get-memory-total()
            - system:get-memory-free()
        )
    }</available>
</memory>
```

The figures of interest here are really `max` and `available`. These report how much memory eXist may use in total before it receives an `OutOfMemoryError` and the amount of remaining memory available to eXist, respectively, while `allocated` gives a breakdown of the current memory use.

VisualVM

VisualVM is a truly excellent GUI tool (built with the NetBeans platform) that allows you to connect to any JVM and peer inside it to see exactly what is happening. VisualVM can provide a great deal of information about a running JVM; however, as this is not a book on VisualVM, we will just look at the memory statistics that it can provide.

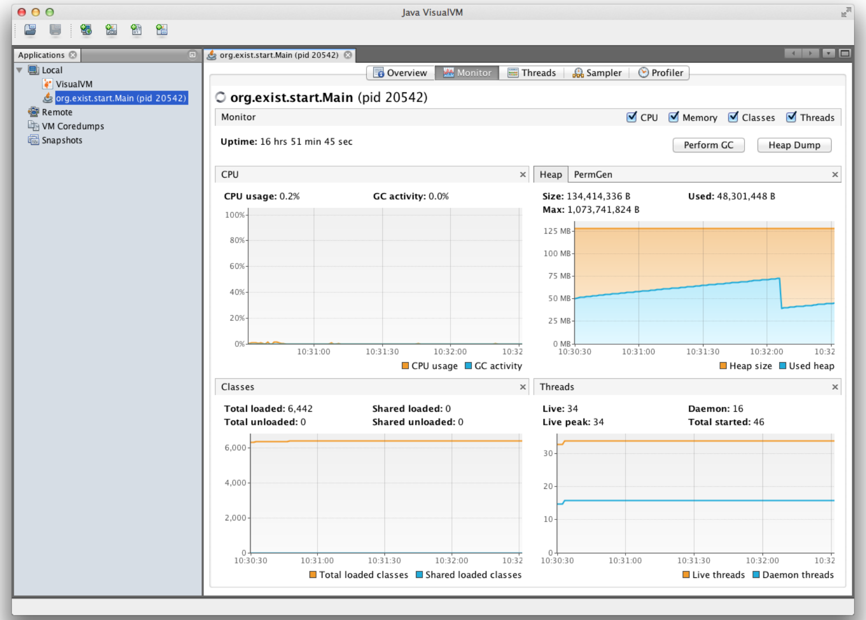
If you have a modern JDK (version 7 or later) from Oracle, then VisualVM is provided with it; otherwise, VisualVM may be available as a package for your system or can be downloaded from *https://visualvm.java.net*. Once you have VisualVM installed, you start it by simply running the `jvisualvm` command.

If you are running VisualVM on the same machine as eXist, VisualVM can directly connect to any local Java process. Thus, upon starting VisualVM, you will see a list of running Java processes, from which you can simply select `org.exist.start.Main` (see Figure 15-4).

Conversely, if you wish to connect to a remote eXist instance with VisualVM, you can either install *jstatd* on your eXist server and then connect remotely using VisualVM, or connect remotely from VisualVM via JMX (see "JMX" on page 387).

Java Mission Control

Java Mission Control (JMC), which is built on the Eclipse rich client platform and shown in Figure 15-5, is another project that in some ways is similar to VisualVM. It was added to Oracle's JDK in JDK 7 update 40 (quite some time after VisualVM). While it is likely that VisualVM will be adopted in the OpenJDK in the future, it is unlikely that the same will happen for JMC, as it contains technology from what was previously known as JRockit and is not open source. Therefore, JMC can be considered proprietary to Oracle JDKs at present.



org.exist.start.Main (pid 20542)

Uptime: 16 hrs 51 min 45 sec

CPU usage: 0.2%
GC activity: 0.0%

Size: 134,414,336 B
Used: 48,301,448 B
Max: 1,073,741,824 B

Total loaded: 6,442
Total unloaded: 0
Shared loaded: 0
Shared unloaded: 0

Live: 34
Live peak: 34
Daemon: 16
Total started: 46

Figure 15-4. VisualVM inspecting eXist memory use

JMC includes a facility called the Java Flight Recorder (JFR) that can record and report information about events emitted by the JVM and applications running on the JVM. At present, as JFR is very new technology, eXist does not emit any specific events, but much can be gathered from the standard events emitted by the JVM itself. As with VisualVM, JMC can monitor local or remote JVMs (through the use of JMX).

If you have Oracle JDK 7u40 or later installed, you can start JMC simply by running the `jmc` command from the *bin* folder of the JDK.

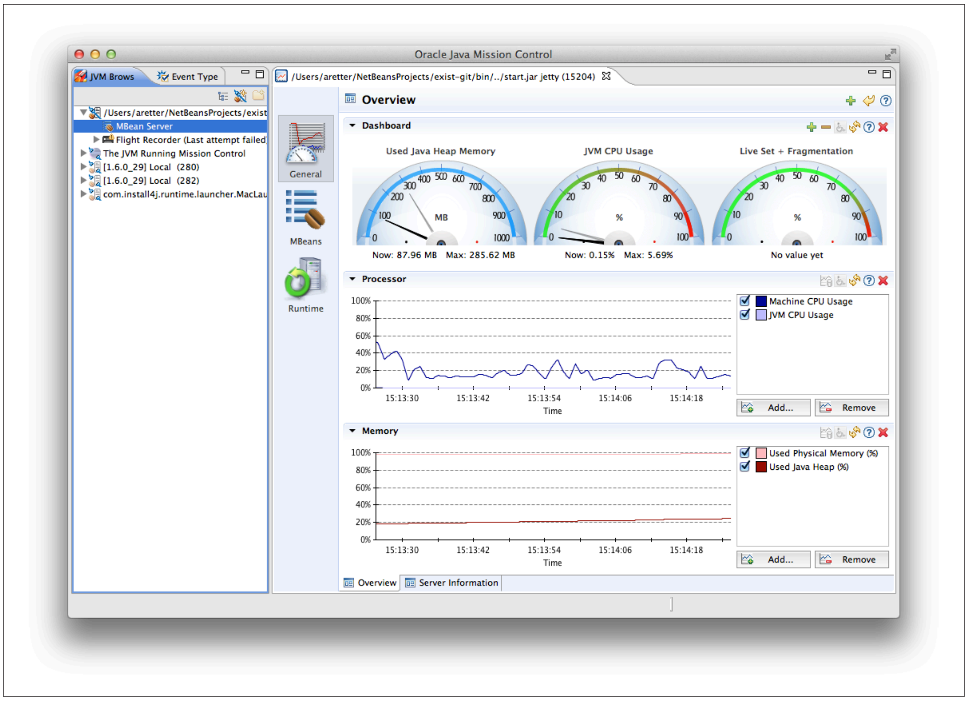


Figure 15-5. Java Mission Control inspecting eXist

Cache Tuning

When the database is operating optimally, all caches will be completely filled with data relevant to the most frequent node retrievals (and therefore query results). For further background on how caching works in eXist, "Paging and Caching" on page 85 is recommended preliminary reading. As eXist attempts to fill the entire cache space, and this memory is taken away from the memory available to the rest of eXist, administrators must be careful not to allocate caches that are sized in such a way that they starve the rest of eXist of memory; this potentially leads to out-of-memory errors and database crashes.

As Figure 15-6 shows, it is recommended that the combined size of the general cache (`cacheSize`) and collections cache (`collectionCache`) should not exceed one-third of the total memory available to eXist, unless you are working in environments where the total memory is greater than 8 GB. Even then, you should take care to ensure that eXist is not memory-starved by its caches. A large cache does not automatically result in better performance!

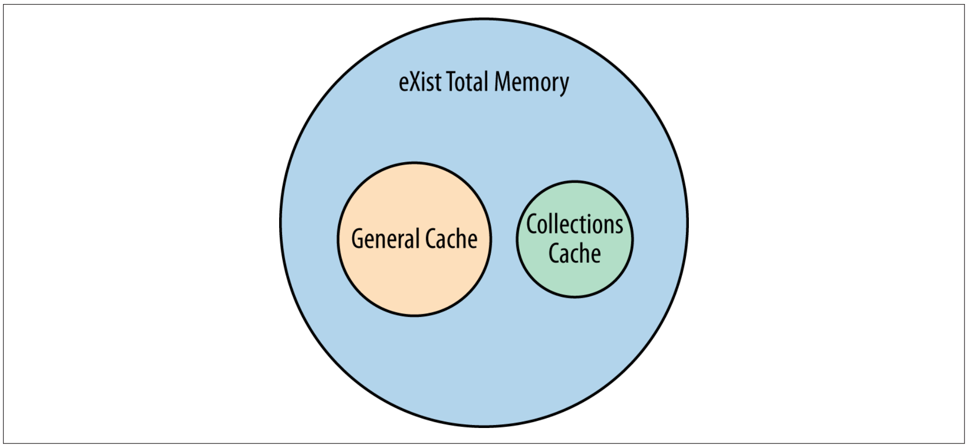


Figure 15-6. How eXist's caches take from memory available to eXist

The collections cache should be large enough to keep the metadata of frequently queried collections in memory. The default cache settings in eXist are far too conservative for all but local development with small datasets. To determine if increasing the collections cache size will make a difference for you, you can record the time it takes to execute the simple query `collection("/db")/`*`someRootElem`* before and after changing the size. Remember that you must restart eXist after changing any memory or cache settings in order for them to take effect!

A common warning sign that a cache in eXist has become too small (due to more intensive querying or storing and querying a larger dataset) is a sudden drop in query performance and/or a noticeable increase in disk I/O when you're uploading large sets of documents into the database. This is caused by cache thrashing; increasing the memory available to the Cache Manager and restarting eXist may resolve the issue.

When testing query performance, remember that a query may be slower the first time it runs, as the relevant pages may not yet be cached in memory. When testing performance or profiling a query in eXist, you should run it several times and average the best runtimes to eliminate both cache warm-up times and JVM JIT compilation.

You can view the real-time behavior of the caches in eXist by either connecting to eXist using a JMX client such as JConsole, which is provided with the JVM (see "JMX" on page 387), or from an XML feed of the JMX output by accessing the URI *http://localhost:8080/exist/status/*.

During normal operation, the reported number of cache *hits* should always exceed the *misses* (fails) by an order of magnitude. Remember that the caches need to warm up by filling before you will see the optimum cache hits! You can determine cache capacity by comparing the reported cache size against the used size of the cache. If the number of cache misses starts approaching the number of cache hits, it is a sure sign that the cache size needs to be increased.

Backup and Restore

One of the most important aspects of managing any database system is to ensure that you have a robust backup policy in place. Should your server fail from a hardware or software issue, it is often essential that you can rebuild the server and restore a backup of your data. eXist provides two different types of backup:

Data copy

This is simply a copy of all files from eXist's data folder (typically *$EXIST_HOME/webapp/WEB-INF/data*) to some other location. The data must always be copied when the database is in a consistent state, and the files are not being written to. You can do this either manually (including via a system scheduler such as cron on Unix systems or as a Windows scheduled task) when the database is shut down, or automatically by using eXist's scheduler.

The eXist scheduled job `org.exist.storage.DataBackup` can be enabled in *$EXIST_HOME/conf.xml* and will attempt to create a copy of the database every time it is run; it will also ensure that the database is in a consistent state when it is run by switching into *protected mode*.

Data export

This is an export of the database, which means that all XML documents are serialized from the binary database files back into individual XML files, and a copy is made of all binary documents. A data export is a serialized copy of the database collection hierarchy and may target either a destination folder or a ZIP file.

> On some combinations of operating system and JRE, eXist may have trouble creating data export backups to ZIP files that are larger than 4 GB. In this scenario, it is recommended that you export to a destination folder instead of a ZIP file.

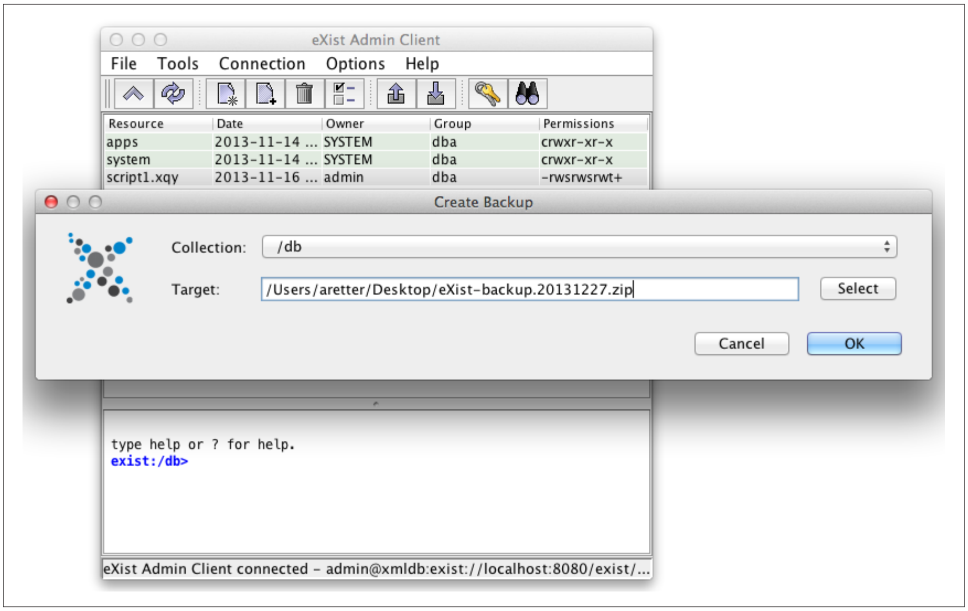
While the data copy variety of backup is always performed server-side, the data export backup may be performed either client-side or server-side.

Client-Side Data Export Backup

When backups are executed by a client of the database over the network, the database is *not* switched to protected mode. This means that the database may be accessed and modified during the backup. While individual documents will be consistent at the time of serialization, consistency across documents is not guaranteed! The advantage here is that the database continues functioning normally and servicing users while the backup is occurring, but the backup is not a snapshot of the database at a single point in time. The other disadvantage is that a client-side backup will not attempt to back up documents or collections that may be damaged in some way; rather, they will be skipped. Client-side backups are always initiated via the XML:DB API. For more rigorous backup options, see "Server-Side Data Export Backup" on page 400.

Java Admin Client backup

The Java Admin Client (see "Java Admin Client" on page 373) provides a convenient way to perform a client-side data export backup of the database. The Java Admin Client allows you to choose the collection hierarchy to back up. By choosing the */db* collection, you can back up the entire database, which includes all users and collection configurations (indexes, triggers, etc.). Alternatively, you may choose just to back up a specific data collection hierarchy. You can open the Backup dialog in the Java Admin Client either from the toolbar, by clicking the Backup icon, or by selecting the Tools→Backup menu item (see Figure 15-7).



Resource	Date	Owner	Group	Permissions
apps	2013-11-14 ...	SYSTEM	dba	crwxr-xr-x
system	2013-11-14 ...	SYSTEM	dba	crwxr-xr-x
script1.xqy	2013-11-16 ...	admin	dba	-rwsrwsrwt+

Collection: /db

Target: /Users/aretter/Desktop/eXist-backup.20131227.zip

```
type help or ? for help.
exist:/db>
```

eXist Admin Client connected - admin@xmldb:exist://localhost:8080/exist/...

Figure 15-7. Java Admin Client Backup dialog

Restoring a backup with the Java Admin Client is even simpler: you just need to open the Restore Backup dialog by clicking the Restore icon or choosing the Tools→Restore menu item, and then selecting the previous backup that you created.

Command-line backup

Command-line scripts for performing a client-side data export backup (and restore) of the database are provided with eXist in the form of the *$EXIST_HOME/bin/backup.sh* file (for Unix, Linux, and Mac platforms) and *$EXIST_HOME/bin/backup.bat* file (for Windows platforms). These scripts take several arguments, which are demonstrated in Example 15-1 and explained in Table 15-3.

Example 15-1. Backing up the entire database (in ZIP format) from the command line

```
$EXIST_HOME/bin/backup.sh --user admin --password some-password --backup /db
--destination /export/backups/exist-db.201312271159.zip
```

Table 15-3. Command-line backup arguments

Argument	Description	Mandatory/optional
`-u` or `--user`	The username for connecting to the database. Typically, must be a `dba` user when performing a backup.	Mandatory, but if omitted, the default username `admin` is used
`-p` or `--password`	The password for the user connecting to the database.	Mandatory, unless the user does not have a password
`-b` or `--backup`	The collection hierarchy to back up (e.g., */db*).	Mandatory
`-d` or `--destination`	The destination for the backup. Either a folder path, or a filename ending with *.zip* to create a ZIP file backup.	Mandatory
`-o` or `--option`	Any additional options for the backup client that are needed to connect to the eXist server. For example, if using SSL: `-ossl-enable=true -ouri=xmldb:exist://localhost:8443/exist/xmlrpc`.	Optional

When you're restoring a backup from the command line, the same scripts take slightly different arguments, as you can see in Example 15-2 and Table 15-4.

Example 15-2. Restoring a database backup (in ZIP format) from the command line

```
$EXIST_HOME/bin/backup.sh --user admin --password some-password
--restore /export/backups/exist-db.201312271159.zip
```

Table 15-4. Command-line restore arguments

Argument	Description	Mandatory/optional
`-u` or `--user`	The username for connecting to the database. Typically, must be a `dba` user when performing a backup.	Mandatory, but if omitted, the default username `admin` is used
`-p` or `--password`	The password for the user connecting to the database.	Mandatory, unless the user does not have a password
`-r` or `--restore`	The location of the backup to restore. Either a folder path, or a filename ending with *.zip* to restore a ZIP file backup.	Mandatory
`-P` or `--dba-password`	The password of the `admin` user in the database backup that you are restoring.	Mandatory, if the backup includes the collection */db/system/security/exist*
`-o` or `--option`	Any additional options for the backup client that are needed to connect to the eXist server. For example, if using SSL: `-ossl-enable=true` `-ouri=xmldb:exist://localhost:8443/exist/xmlrpc`.	Optional

Ant backup task

You can use the Ant extension tasks for eXist (see "Using Ant with eXist" on page 381) for performing client-side data export backup and restore of the database.

Once you have the Ant extensions configured in your Ant script (typically *build.xml*), then you can configure the Ant backup extension using the Ant code shown in Example 15-3. Its parameters are listed in Table 15-5.

Example 15-3. Backing up the entire database with Ant

```
<xdb:backup user="admin" password="some-password"
  uri="xmldb:exist://localhost:8080/exist/xmlrpc/db"
  dir="/export/backups/exist-db.201312271206.zip"/>
```

Table 15-5. Ant backup task parameters

Parameter	Description
`user`	The username for connecting to the database. Typically, must be a `dba` user when performing a backup.
`password`	The password for the user connecting to the database.
`uri`	The XML:DB API URI to the database collection that you wish to back up. For example, `xmldb:exist://localhost:8080/exist/xmlrpc/db` would back up the entire database, while `xmldb:exist://localhost:8080/exist/xmlrpc/db/my-collection` would back up the */db/my-collection* collection hierarchy.
`dir`	The destination for the backup. Either a folder path, or a filename ending with *.zip* to create a ZIP file backup.

Conversely, you can configure the Ant restore extension using the Ant code shown in Example 15-4. Its parameters are outlined in Table 15-6.

Example 15-4. Restoring a database backup with Ant

```
<xdb:restore user="admin" password="some-password"
  uri="xmldb:exist://localhost:8080/exist/xmlrpc"
  file="/export/backups/exist-db.201312271206.zip"/>
```

Table 15-6. Ant restore task parameters

Parameter	Description	Mandatory/optional
`user`	The username for connecting to the database. Typically, must be a `dba` user when restoring a backup.	Mandatory
`password`	The password for the user connecting to the database.	Mandatory
`uri`	The XML:DB API URI to the database server (e.g., `xmldb:exist://localhost:8080/exist/xmlrpc/db`).	Mandatory
`dir`	The location of the backup to restore if it is a folder.	Mandatory, if the restore source is a folder
`file`	The location of the backup to restore if it is a ZIP file.	Mandatory, if the restore source is a ZIP file
`restorePassword`	The password of the `admin` user in the database backup that you are restoring.	Mandatory, if the backup includes the collection */db/system/security/exist*

Server-Side Data Export Backup

Server-side backups in eXist are always performed by the eXist Scheduler (see "Scheduled Jobs" on page 435), which executes the backup as a *system task*. Whether the job is scheduled in advance as a one-off or repeatable operation, or is triggered directly at some point, it will always be managed by the Scheduler.

System tasks always ensure that the database is in *protected mode*. This means that the database is in a consistent state because all pending transactions have completed, the database journal has been flushed to persistent storage, and the system task will execute in isolation, which blocks and queues all incoming transactions until the task has finished executing.

As a server-side backup is executed in protected mode, the database is unavailable for general use while the backup is performed. Server-side backups initially perform a consistency and sanity check against the database, and this information is used to inform the backup content. The consistency and sanity check can detect problems in the database storage and still allow collections and documents that may have become damaged to be exported in the backup, thus ensuring the most rigorous backup of your data.

Documents that still exist in the database but through some issue have become *detached* from a collection—and so are effectively invisible—may still be exported as part of the backup content. These documents will be placed into the special collection */db/lost_and_found*. Details of the consistency check are logged into the same folder as the destination of the backup.

Scheduled backups

If you wish to create a scheduled backup—either as a one-off task or a periodic operation—you can configure this in eXist's configuration file (*$EXIST_HOME/conf.xml*) within the `scheduler` element indicated by the XPath */exist/scheduler*, by enabling and configuring the `org.exist.storage.ConsistencyCheckTask` scheduled job (see Example 15-5 and Table 15-7). For more information on scheduling jobs and the job configuration syntax, see "Scheduling Jobs" on page 436. This configuration has to be set before the eXist server is started, or you have to restart eXist after making changes. If you wish to schedule a new backup job without restarting eXist, you can do so using the XQuery `scheduler` extension module (see `scheduler`). Any scheduled backup will back up the entire database.

Example 15-5. Scheduled backup configuration: daily 1 a.m., fortnightly incremental backup

```
<job type="system" name="daily-backup" class="org.exist.storage.ConsistencyCheckTask"
cron-trigger="0 0 01 * * ?">
  <parameter name="output" value="/export/backups"/>
  <parameter name="backup" value="yes"/>
  <parameter name="incremental" value="yes"/>
  <parameter name="incremental-check" value="yes"/>
  <parameter name="max" value="14"/>
</job>
```

Table 15-7. Scheduled consistency check and backup job parameters

Parameter	Description	Mandatory/ optional
`output`	The destination folder for the logfiles of the Consistency Check task; if performing a backup, also the location for backup files. If you're using a relative path, it is interpreted relative to *$EXIST_HOME/webapp/WEB-INF/data*.	Mandatory
`backup`	After the consistency check, should a backup of the database be performed? Either `yes` or `no`.	Optional, default is `no`
`incremental`	If performing a backup, should a full backup or an incremental backup (just documents that have changed since the last backup) be performed? Either `yes` for an incremental backup, or `no` for a full backup.	Optional, default is `no`

Parameter	Description	Mandatory/ optional
incremental-check	If performing an incremental backup, should a consistency check be performed first? Disabling the Consistency Check task for incremental backups can lead to faster backup times, but has the downside of creating a less rigorous backup of the database. Either yes or no.	Optional, default is yes
max	The maximum number of incremental backups before another full backup should be created.	Optional, default is 5
zip	Should the backup be written to a ZIP file? Either yes for a ZIP file or no for a full folder export.	Optional, default is yes

Backups from XQuery

It is also possible to almost immediately request a server-side data export backup of the database on an ad hoc basis from XQuery, by triggering the Consistency Check system task. We describe this as *almost immediately* because, as you may recall from earlier, server-side backups are always scheduled, and the database may need to finish processing any other current requests before it can enter protected mode to perform the backup. You can trigger the Consistency Check system task from XQuery by using the `system:trigger-system-task` function (see Example 15-6). For further details of the `system` XQuery extension module, see `system`. The function takes the same parameters as described in Table 15-7.

Example 15-6. Triggering a full server-side data export backup from XQuery

```
let $parameters :=
    <parameters>
        <param name="output" value="/export/backups"/>
        <param name="backup" value="yes"/>
        <param name="incremental" value="no"/>
        <param name="zip" value="no"/>
    </parameters>
return
    system:trigger-system-task("org.exist.storage.ConsistencyCheckTask", $parameters)
```

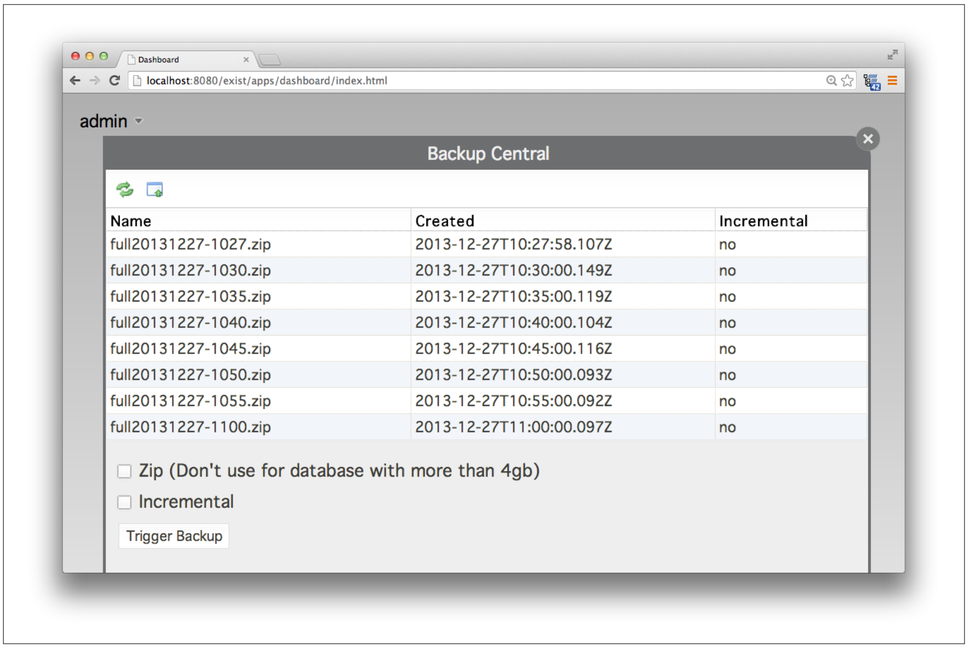
Conversely, you can restore a database backup from XQuery by calling the `system:restore` function, as Example 15-7 shows.

Example 15-7. Restoring a backup from XQuery

```
system:restore("/export/backups/full20131228-1035", "some-password", "some-password")
```

Dashboard Backup app

Recently, a simple backup application has been added to the eXist dashboard that allows you to see previously created backups, and also to trigger a new immediate backup (see Figure 15-8). The backend of this application is written in XQuery and uses the same underlying functions as discussed in the previous section.



Name	Created	Incremental
full20131227-1027.zip	2013-12-27T10:27:58.107Z	no
full20131227-1030.zip	2013-12-27T10:30:00.149Z	no
full20131227-1035.zip	2013-12-27T10:35:00.119Z	no
full20131227-1040.zip	2013-12-27T10:40:00.104Z	no
full20131227-1045.zip	2013-12-27T10:45:00.116Z	no
full20131227-1050.zip	2013-12-27T10:50:00.093Z	no
full20131227-1055.zip	2013-12-27T10:55:00.092Z	no
full20131227-1100.zip	2013-12-27T11:00:00.097Z	no

☐ Zip (Don't use for database with more than 4gb)
☐ Incremental
Trigger Backup

Figure 15-8. Dashboard Backup application

Restoring a Clean Database

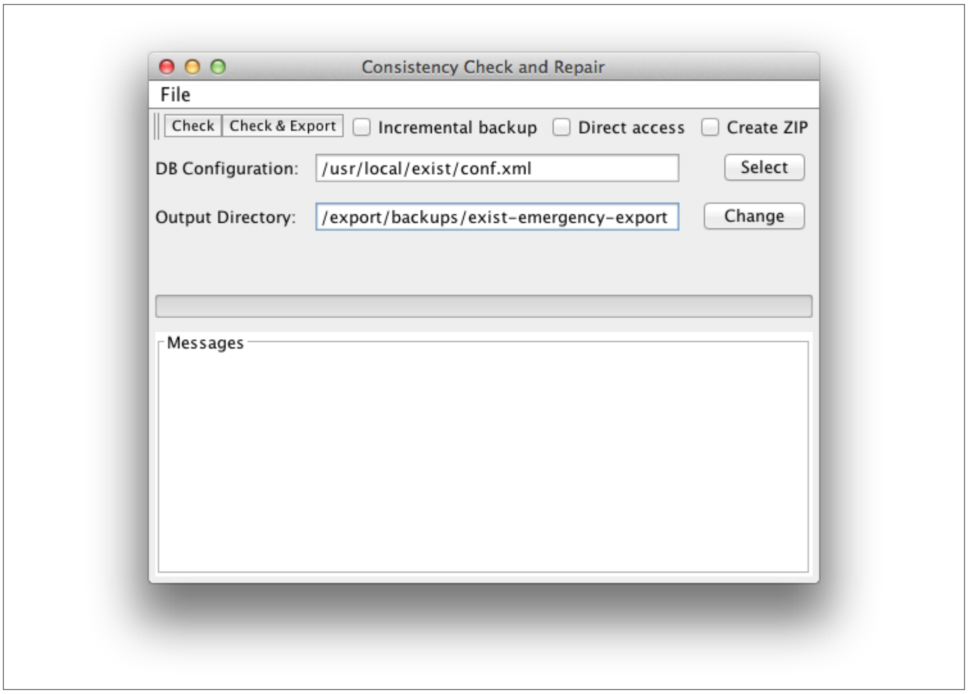
When you are restoring a backup of a database in eXist, documents and collections in the existing database (before the restore is performed) are not overwritten unless they also exist in the backup that is being restored. This behavior is intentional, as it allows you to back up individual collection hierarchies and manage them independently. You can then choose to restore different collection hierarchies at different times.

If you wish to start with an empty database and then restore your backup so that the database contains only the data of your backup, then you need to first shut down eXist and remove the database files. *Remember, this will cause you to lose all of your data!* To remove the database, simply delete all the files and folders (excluding *README*, *RECOVERY*, and *export*) from your eXist data directory and journal directory: unless you have reconfigured this, they are one and the same directory, and located at *$EXIST_HOME/webapp/WEB-INF/data*. When you then restart eXist you will have an empty database, and you may restore your backup as normal.

Emergency Export Tool

The eXist database storage subsystem is designed in such a way that it should protect the integrity of your data during the vast majority of hardware or software failures. However, sometimes in the real world unexpected and unexplained events do take place. After a crash, when restarting, eXist will try to recover if necessary by examining its database journal. Under rare circumstances, however, eXist may not be able to automatically recover, at which point it will refuse to start up to avoid causing further damage to the database.

Should eXist refuse to start, there are two steps that you must take. First and most importantly, assuming that your data is important to you, you must ensure that you have a recent copy of your database. If you do not have a recent backup, then you can use the Emergency Export tool to examine the consistency of the database and create a backup (see Figure 15-9).



DB Configuration: /usr/local/exist/conf.xml

Output Directory: /export/backups/exist-emergency-export

Figure 15-9. Emergency Export tool GUI

You must ensure that eXist is completely shut down (even if it refused to start properly) before running the Emergency Export tool, because the tool is independent of eXist and accesses the database files directly.

You can start the Emergency Export tool as a GUI tool by executing:

```
java -jar $EXIST_HOME/start.jar org.exist.backup.ExportGUI
```

Alternatively, you can run the Emergency Export tool as a command-line utility without the GUI by executing:

```
java -jar $EXIST_HOME/start.jar org.exist.backup.ExportMain
```

When run from the command line, the Emergency Export tool has a number of parameters that may be specified (these are also mirrored in the GUI); these are detailed in Table 15-8.

Table 15-8. Emergency Export tool command-line arguments

Argument	Description
`-d` or `--dir`	The destination folder for the logfiles of the Consistency Check task; if performing an export (backup), also the location for exported files.
`-c` or `--config`	The *$EXIST_HOME/conf.xml* config file for the eXist instance you wish to perform the emergency export on.
`-D` or `--direct`	Uses a more aggressive approach to directly access nodes in the database without examining some indexes. Can be useful if there is also index corruption!
`-x` or `--export`	Performs a full data export backup from the database files.
`-i` or `--incremental`	If performing an export, and an existing backup is present in the destination folder, this specifies that only an incremental backup will be performed.
`-n` or `--nocheck`	Skips performing a consistency check, and just attempts to export all data.
`-z` or `--zip`	If performing an export, then this indicates that the database content should be exported to a ZIP file as opposed to a folder.

If the first step was to ensure that you have a backup of your data, then the second step is obviously to seek support in getting your database server running again.

If you have a complete backup of your database and are certain that all your data is present, you can simply restore the backup to a clean database, as discussed in the previous section. If you do not have a complete backup of your database and are having trouble recovering your data or restarting eXist, then you should refer to "Getting Support" on page 413. Obviously, if you have Consistency Check reports available from running the Emergency Export tool, then you should consider submitting the most relevant of these as part of your supporting documentation when requesting assistance.

Installing eXist as a Service

In "Downloading and Installing eXist" on page 19 we looked at installing eXist in a development or desktop environment, through use of the eXist installer. In this section we look at how you install eXist into a server environment, where eXist is configured to integrate with your operating system's service management, so that eXist is started and stopped correctly when your server powers up or shuts down.

When you are installing eXist into a server environment that is servicing real users, one of your major concerns should be security, so we would suggest Chapter 8 as prerequisite reading.

Regardless of your operating system, you first need to place a copy of the eXist distribution onto your server, which you can do using either:

- The eXist IzPack installer, as discussed in "Downloading and Installing eXist" on page 19, and typically running it in headless (non-GUI) mode, as shown in "Installing eXist" on page 22. If you are using a Windows server or X graphics environment, then you may instead opt to use the GUI mode.
- The source code, and compiling either the `develop` branch if you want the absolute bleeding edge of eXist development, or one of the release tags (such as `eXist-2.1`) if you want a stable release. This is discussed in "Building eXist from Source" on page 485.

You may, of course, place eXist anywhere you wish on your system, but it is common for eXist to be installed into *C:\Program Files\eXist* on Windows systems and */usr/local/exist* or */opt/exist* on Linux/Unix/Mac systems.

Solaris

We will discuss installing eXist as a service on the Solaris operating system first, as the approach for service management on Solaris is unique, whereas the same tool is used for all other supported platforms.

Since version 10, Solaris has included a facility called the Service Management Framework (SMF), which is responsible for starting and stopping all services on the system and reporting on service errors. To a certain extent, SMF can also be configured to automatically restart failing service trees as part of Solaris's predictive self-healing technologies.

eXist can be installed into Solaris's SMF and be controlled through the standard Solaris service management commands: `svccfg`, `svcadm`, and `svcs`. eXist has been tested with SMF on Solaris 10 and 11, OpenSolaris, and OpenIndiana, but it is likely it will also work with any Illumos-based distribution that uses SMF and provides Java 1.6 or newer.

Manifests and service scripts for SMF are shipped with eXist and provided in the *$EXIST_HOME/tools/Solaris* folder. As comprehensive documentation for integrating eXist with SMF is already provided in the file *$EXIST_HOME/tools/Solaris/README.txt*, we will not attempt to reproduce it here.

Windows Linux and Other Unix

eXist ships with a third-party tool from Tanuki Software called the Java Service Wrapper. eXist uses the open source GPLv2 version of the Java Service Wrapper, which is written in C and Java. The C component provides native integration with specific operating systems, while the Java component provides abstractions atop the C component to allow you to use the same configuration and management options across all platforms.

eXist provides configuration files and scripts for the Java Service Wrapper that allow you to easily install eXist as a service on any platform that the tool supports—i.e., Windows, Linux (x86), and Mac OS X. The Java Service Wrapper can also support IBM AIX (PPC), z/OS, z/Linux, FreeBSD (x86), HPUX, and Linux (PPC, ARM) by way of a *Delta Pack* that can be downloaded from the Tanuki Software website. While the Java Service Wrapper also supports Solaris, eXist provides its own solution for the Solaris platform, as discussed in "Solaris" on page 406.

A preconfigured Java Service Wrapper ships with eXist and is provided in the *$EXIST_HOME/tools/wrapper* folder. Before installing eXist as a service, you should consider the access rights that eXist has to your system and the security concerns of running eXist as a service, as discussed in "Reducing Collateral Damage" on page 175.

To install eXist as a service on Windows platforms, you can execute the following command from a Windows console (*cmd.exe*):

```
C:\> %EXIST_HOME%\tools\wrapper\bin\install.bat
```

After successful installation as a service on Windows, eXist will show up in the Windows Services as "eXist Native XML Database."

To install eXist on Linux, Mac OS X, and other Unix platforms, you can execute the following command from a terminal as `root` or an equally privileged user:

```
# $EXIST_HOME/tools/wrapper/bin/install.sh
```

After successful installation as a service, eXist will be installed as the named service `eXist-db`. You should then be able to manage the service with your platform's service management tools. On Linux this would typically be the `service` and/or `chkconfig` commands; for example, running `service eXist-db status` should report the running status of eXist.

When you are installing on Linux, if your distribution uses the Upstart event-based startup system (e.g., Ubuntu, Linux Mint, ChromeOS, RHEL/CentOS 6, Scientific Linux, and Oracle Linux), you should edit *$EXIST_HOME/tools/wrapper/bin/install.sh* before running it and set the flag `USE_UPSTART` to `true`.

Hosting and the Cloud

The eXist developers are often asked what is needed to host eXist on a third-party server and where users can find hosting companies or cloud providers that support or enable eXist to be used. The basic requirements from a hosting provider are the same as those needed to run eXist on your own computer or server: you simply need a system that offers a JRE (or JDK, if you wish to build eXist from source code) of at least version 6, and allows you remote console access to install eXist.

Unfortunately, full access to a remote host (physical or virtualized) usually comes at a higher cost than shared application server solutions where you can just upload some PHP or ASP code. However, eXist should be viewed as an application server platform in its own right, as it offers far more functionality and convenience through XQuery, XSLT, XProc, and XForms than lower-level programming languages such as PHP or ASP, which must also be coupled with database access to even begin to approach what eXist offers. In this section, we will examine some of the options available for hosting eXist.

Entic

Entic is a virtual private server (VPS) provider in San Jose, California that has provided the VPSes that have powered the eXist Solutions website and Adam Retter's own websites since 2008.

Entic provides very reasonably priced Solaris zones. These are billed monthly, depending on the resources allocated to them. With Entic you purchase a pool of RAM, CPU, and disk space, which you can then allocate between as many or as few VPSes as you wish. The VPSes can be dynamically allocated and resized on the fly from the Entic website with zero downtime.

eXist has support for running within the Service Management Framework (SMF) of Solaris systems (see *$EXIST_HOME/tools/Solaris/README.txt*). Entic was initially chosen for eXist due to eXist's excellent performance on Sun (now Oracle) JVMs running on Sun Solaris atop Sun x64 servers. Whether this performance metric still stands remains to be tested.

Two final points in favor of Entic are that it's directly connected to the Internap Internet backbone, therefore providing plenty of low-latency bandwidth, and that its support staff are fantastic, being both personal and very flexible in their approach.

Amazon EC2

Amazon is probably the largest and best known of all the cloud providers and offers a plethora of services, including virtual machines. Amazon's Elastic Compute Cloud (or EC2, as it is better known) is a cloud computing platform where virtual machines can easily be created and destroyed.

EC2 supports a wide choice of operating systems, including Solaris, various Linux distributions (including EC2's own Amazon Linux), FreeBSD, NetBSD, and Windows. Amazon EC2 virtual machine instances are created from Amazon machine images (AMIs), which contain the operating system and any preinstalled software. A wide variety of AMIs are available from the AWS Marketplace. The cost of running a virtual machine on EC2 is typically calculated by the hour and depends on four main factors:

Instance sizing
: Amazon offers a variety of virtual machine instance sizes. Some are optimized for CPU processing, others for large memory use, and still others for storage and I/O performance. Which to use with eXist will depend entirely on your use of eXist, the database size, and the queries that you are performing. It is entirely feasible with EC2 to start on one type of instance, and then at a later stage create an image of your virtual machine and migrate to a larger or smaller instance as necessary.

Instance location
: Amazon has several data centers located in the US, South America, Europe, and Asia. Amazon charges differently depending on which data center you wish to run your instance in. The cheapest data center is usually on the East Coast of the US, but it may make more sense to locate your servers closer to your customers to reduce latency, or you may be required to stay within a political region to comply with governmental data protection laws.

On-demand or reserved instances
: If you are running servers on EC2 for production purposes or over a longer time frame than a few weeks, it could be cheaper to prepurchase reserved CPU compute hours rather than paying EC2's on-demand rate.

Software
: Depending on the AMI that you choose to use to create your virtual machine instance, there may be an extra cost associated with provided and/or preconfigured software packages.

One of the nice things about EC2 is that it provides a free tier. If you qualify, then the free tier gives you access to a *micro* virtual machine instance for one year that you can use for gaining experience and also for trying out different software on EC2. The current `t2.micro` instance at the time of writing provides:

- Linux (Amazon, Red Hat, or SLES) or Windows

- One vCPU (equivalent to one Hyper-Thread of an Intel Xeon processor operating at 2.5 GHz burstable to 3.3 GHz)
- 1 GB of RAM
- 30 GB of Elastic block storage
- Basic security group firewall

It is entirely possible to install and run eXist (and even compile it) on the EC2 free-tier micro instance; this has been tested with Amazon Linux, but other operating systems should also work. Obviously the free tier is not going to allow you to run a huge or busy eXist service, but for very small applications, low-traffic websites, or experimentation it is perfectly fine.

Make sure to correctly configure your EC2 instance types, and be aware of the associated costs! If you are expecting to use the free tier, we would recommend that you verify you are actually receiving the service for free. Amazon's free-tier arrangement is subject to change, and we cannot be held responsible for unexpected costs incurred from using EC2.

eXist AMI

You may be wondering how you install eXist onto an Amazon EC2 instance. For your convenience, Adam Retter has created an EC2 AMI that can be used for easily starting your own eXist server on Amazon EC2. The AMI is based on Amazon Linux with eXist 2.1 and NGINX installed. It is available from the AWS Marketplace free of charge /TBD: https://aws.amazon.com/marketplace/seller-profile/ref=dtl_pcp_sold_by?id=7a9e551e-66bf-4093-bf25-a34318b5fec5.

eXist Solutions is planning to make available Amazon EC2 AMIs for eXist LTS (Long-Term Support edition), but these are not yet available (see "Commercial Support" on page 415).

Installation. eXist has been cloned directly from its GitHub repository into */usr/local/exist* and compiled using OpenJDK 7. eXist has been configured so that it stores its database files and journal into */exist-data*. This means that should you wish to, at any point in the future you can easily update to a newer version of eXist from GitHub and recompile because the application code and its data are stored in separate directory trees.

Service. eXist has been configured to be started and stopped when the instance starts and stops; it has been integrated into the system *upstart* via eXist's Java Service

Wrapper integration. If you wish to start or stop eXist, you can run `sudo service eXist-db start` or `sudo service eXist-db stop`.

eXist will run under the `exist` user account on the machine, and the folders */usr/local/exist* and */exist-data* must have appropriate read/write permissions for the `exist` user. By default, these are set up and configured correctly for you. If you want to access or modify files in those folders, it is best to `su` to the `exist` user.

eXist has been configured to listen on the standard 8080 TCP port, but for security the EC2 firewall has been enabled to allow access to the machine instance only on ports 80 (HTTP) and 22 (SSH). Thus, if you want to access eXist from the Web, be aware that it has been proxied behind the Nginx web server on port 80, as described in "Reverse proxying" on page 185. The Nginx configuration file is located at */etc/nginx/nginx.conf*. It is currently configured to forward just the REST Server interface; however, commented out in that same file is an example of forwarding the RESTXQ Server interface. If you wish to start or stop Nginx, you can run `sudo service nginx start` or `sudo service nginx stop`.

Administering. As only the REST Server is accessible from the Web on the EC2 instance, you may wonder how you can work with your eXist server and administer it using all of the functionality that you are used to. The answer is to forward a TCP port from your own development/admin machine to your eXist server when you need to get at the full eXist server. You can do so by making use of *port tunneling* in SSH.

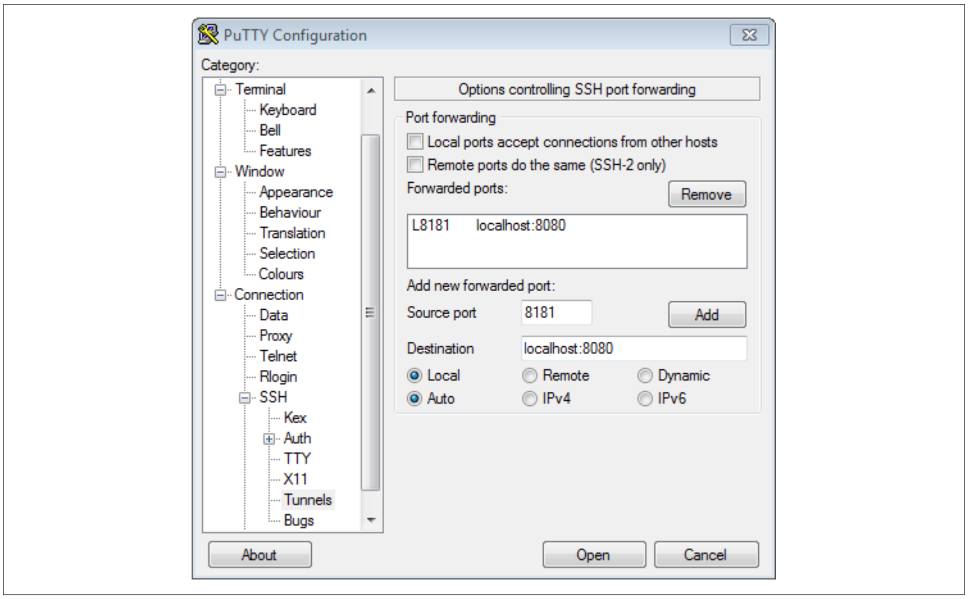
For example, consider the following OpenSSH client command:

```
ssh -i ~/ec2-keys/aretter.pem -L8181:localhost:8080
  ec2-user@ec2-107-22-152-27.compute-1.amazonaws.com
```

This command would connect by SSH to your EC2 instance (in this example, the EC2 host is `ec2-107-22-152-27.compute-1.amazonaws.com`) and forward the TCP port 8181 on your local machine to port 8080 (the port that eXist is listening on) on the remote EC2 instance. This means that you can use the hostname *localhost* and the port 8181 on your local machine to connect to the remote eXist using your web browser, the Java Admin Client, WebDAV, and so on.

The eXist `admin` user password for the EC2 instance that you create, will be set to the ID of your instance the first time it is run. It is strongly recommended that your first task should be to change this!

The SSH command shown is for Unix, Linux, and Mac systems, but you can achieve the same result from Windows systems using an SSH client for Windows (e.g., PuTTY, shown in Figure 15-10).



Options controlling SSH port forwarding

Port forwarding

- [ ] Local ports accept connections from other hosts
- [ ] Remote ports do the same (SSH-2 only)

Forwarded ports:

```
L8181      localhost:8080
```

Add new forwarded port:

Source port: 8181

Destination: localhost:8080

[✓ Local] Remote Dynamic

[✓ Auto] IPv4 IPv6

Figure 15-10. Port tunneling with PuTTY on Windows

Other Cloud Providers

There are literally hundreds of hosting and cloud providers out there, each offering different server infrastructures, resources, and pricing models. It would take far too much time to evaluate them all for use with eXist. The good news, however, is that many of these platforms provide either complete virtual servers or Java application containers in which eXist can be installed. To round off this section, we will mention just two more cloud providers that we think are interesting and offer something special.

GreenQloud

GreenQloud is based in Iceland, as are its data centers, and this enables its cloud service to be powered by 100% renewable energy. Energy is taken from hydroelectric and geothermal power sources, and GreenQloud's servers are cooled by the local naturally cold air. GreenQloud's compute and storage clouds offer seriously green options, and its innovative dashboard application allows you to closely monitor your energy use, savings, and carbon footprint.

Although they have comparable price points, depending on your server requirements, GreenQloud can be cheaper than Amazon EC2. Amazon EC2 offers far more storage than almost any other cloud provider, but with GreenQloud you get more RAM and CPU. GreenQloud also offers an API that is compatible with Amazon's

EC2 API so that you can easily migrate from EC2 to GreenQloud, or more easily run a hybrid mixed cloud if you don't want to keep all your eggs in one basket.

eXist operates perfectly well in the GreenQloud environment, and if Earth-friendly credentials are important to your business it is an excellent choice. Evolved Binary Ltd is currently evaluating GreenQloud for its virtualized servers.

Digital Ocean

Digital Ocean is a relatively new cloud provider based in New York City. Apart from services that offer great value for money when compared to Amazon EC2 and others, it offers two features of particular interest for eXist users.

The first is that all of Digital Ocean's servers provide 100% Solid State Disk (SSD) storage, which is great news for databases such as eXist where SSD can dramatically improve performance when data is accessed from disk.

The second feature is Digital Ocean's built-in support for Docker. Docker is an *application container technology*, which means that any application, such as eXist, packaged via Docker can be deployed onto any physical, virtual, or cloud service that supports Docker. Digital Ocean allows you to spin up new machines using Docker applications. As there is also already support for Docker on Amazon EC2, it would be really interesting to see a Docker container for eXist that could then be used to deploy eXist to almost anywhere.

Getting Support

Even though you have an excellent book available on the topic, sometimes when you're working with eXist—either as a developer, a system administrator, or somewhere in between—you may need to seek additional support by asking a question, reporting and getting a fix for a bug, or requesting help to understand a particular feature. You have two main options for getting support with eXist: community or commercial.

Whether you look to the community for support or seek some sort of commercial support arrangement, when you're reporting problems or a bug, there are several pieces of supporting information that you should consider submitting with your request:

- Include the exact version of eXist that you are using—either the release version number or, if you're using a version compiled from source code, the Git commit revision number. You can obtain the Git revision by running the command `git show --summary -abbrev-commit`.
- Indicate which operating system you are using, and the version, (e.g., Windows XP Service Pack 3a). Also include whether the operating system is 32-bit or

64-bit. Specify the vendor, version, and CPU architecture of the Java JRE or JDK that you are using as well. The Java version can be obtained by running the command `java -version`, and vendor information by examining the *java.vm* properties when running `java -XshowSettings:properties -version`.

- Be sure to report how much memory is available to the operating system, including how much is allocated to eXist (i.e., the `-Xmx` setting), and how much is allocated to the general cache and to the collections cache in *$EXIST_HOME/conf.xml*.
- When reporting errors or bugs with eXist, make sure to include any pertinent WARN- or ERROR-level log messages from the logfiles, in particular *$EXIST_HOME/webapp/WEB-INF/logs/exist.log*.
- If you believe that eXist has frozen in some way (deadlock/livelock) and is not correctly responding to incoming requests, then a thread dump of the eXist process can be invaluable to the developers. You can obtain a thread dump through one of several methods, such as:
  - Using the *jstack* tool that comes with your JRE. For example, by running `jstack -l 51617 > exist-jstack.txt`, where `51617` is the process ID (PID) of eXist, you can get a file *exist-jstack.txt* containing a thread dump. You can find out the PID of eXist by running the command `jps`.
  - By sending the QUIT signal using the `kill` command on Unix/Linux/Mac systems. For example, by running `kill -s QUIT 19241 > exist-dump.txt`, where `19241` is the PID of eXist, you can get a file *exist-dump.txt* containing a thread dump.
- If the JVM running eXist has crashed, then a file whose name starts with *hs_err_pid* will be created. You should keep a copy in case it's requested by the eXist developers when you're reporting a problem. Ultimately, you may be asked to send a copy to the JRE/JDK vendor if the file reveals a bug in the JVM.
- Remember, when reporting bugs, you need to be able to explain how to reproduce the issue to the community. If a developer cannot reproduce your issue easily, then it becomes hard or impossible for her to assist you. This is particularly important when reporting problems with your XQuery scripts. You should be concerned with creating an absolutely minimal example XQuery script that isolates and reproduces the problem you are experiencing. The easier you can make it for a developer to reproduce your issue, the more likely you will be to get a quick fix. Simply sending several thousand lines of XQuery code and saying that there is a problem somewhere will most likely *not* result in a solution to your problem anytime soon!

Community Support

As discussed in "Contributing to the Community" on page 13, eXist has an excellent open source community, with developers and users supporting each other. Should you have a problem you need help with, we would recommend first looking for a solution in an existing report in the eXist mailing list archive, and if you do not find anything there, then posting to the *eXist-open* mailing list.

If you believe you have found a bug in eXist, it is good practice to discuss it on the mailing list, and to have it confirmed. Once you have a confirmed bug, or if you have a new feature request, then we would encourage you to log it in the GitHub issue tracker for eXist. In this manner, issues are accounted for and managed publicly.

Commercial Support

Commercial support for eXist is available from eXist Solutions. eXist Solutions comprises many of the core developers of eXist and contributes the vast majority of its resources and funding back into advancing the eXist open source project.

eXist Solutions offers an LTS (Long Term Support) version of eXist, which is a stable version supported for at least two years, with significant bug fixes and new features backported from newer versions of eXist. With the LTS version of eXist, you can also purchase a commercial support contract that allows your organization to email or telephone eXist Solutions staff directly when the need arises. While there are several ready-made support contracts available, eXist Solutions is always happy to provide a custom-tailored contract for your organization's needs.

It is also worth mentioning that eXist Solutions can provide developers to assist you should you need additional resources when building your eXist applications, and consultancy services to help improve and/or tune your eXist applications for production use.

CHAPTER 16

Advanced Topics

This chapter introduces some of the more advanced things that can be done with eXist, such as scheduled tasks, triggers, and internal modules. Many, although not all, of the examples in this chapter are developed in Java as plug-ins to eXist. We do not attempt to teach Java in any way in this chapter; rather, where the examples are in Java, we have tried to make them simple, self-explanatory, and easily usable so that even those without Java experience can learn from and make use of them. However, some programming experience will undoubtedly assist you.

If you wish to extend and use additional external libraries for the Java projects in this chapter whose code is deployed into *$EXIST_HOME/lib/user*, be aware that if eXist also uses the same libraries (consult the files in the subfolders of *$EXIST_HOME/lib*), you must ensure that you use exactly the same versions of the libraries as provided with eXist. To achieve this, you must set the correct `version` on the `dependency` in your *pom.xml* file, and also set the `scope` of the `dependency` to `provided`. This is required because your Java code will be running in the same JVM as eXist and therefore uses the same class loader; in this manner of operation, it is only possible to have a single version of a specific class.

XQuery Testing

Creating tests to assert the correctness of your code and to prevent regressions in the future can be desirable for many reasons. Over the last decade there has been a serious focus within software engineering to provide better testing tools for programmers. There have also been a number of testing philosophies and methodologies developed, such as test-driven development (TDD) and behavior-driven development (BDD), which provide guidance on how to deliver robust and well-tested software.

Many programming languages offer various integrated or third-party tools and libraries for facilitating the structured testing of applications. While XQuery is a very high-level functional language, it should certainly not be considered exempt from good testing practices. Thus far, there have been many attempts at creating frameworks for testing XQuery modules and functions, including XTC, XQUT, XSpec, XRay, and Unit Module. Most of these frameworks are, unfortunately, implementation-specific, due to the current lack of *reflective* capabilities in XQuery. There are many different types of tests that can be constructed, but all of the XQuery test frameworks to date have focused on the *unit testing* type of tests. Unit tests are designed to allow you to assert the behavior of a small unit of code, ideally in isolation from the rest of the system. In this chapter, we will also focus on writing unit tests in XQuery.

For many years eXist has provided an XQuery testing mechanism within its own test suite to its core developers, allowing them to write tests in XQuery to assert the correct behavior of eXist. Called XQuery Test Runner, it was never particularly designed with the needs of other XQuery implementations or developers in mind, and it proved somewhat clunky as test suite descriptions had to be written in a separate file using a specific XML DSL.

As of version 2.1 eXist now officially provides XQSuite, a unit test framework designed to be used by any XQuery developer and in such a way that it could be implemented by any XQuery 3.0 vendor. XQSuite provides a standard set of XQuery 3.0 *function annotations* that set up test parameters and make assertions about the results. One of the most interesting characteristics of XQSuite is that it allows you to place your tests directly onto the function you wish to test. Should you wish, however, you can also choose to keep your tests separate from your code (in a different module), by creating wrapper functions that have the XQSuite annotations that simply call your functions under test. Even if you do choose to place the XQSuite annotations onto the functions under test, your code is still potentially portable to platforms other than eXist, as the XQuery 3.0 specification states that if an implementation does not understand an annotation, it can just ignore it.

Perhaps the best way to learn how to use XQSuite is to look at some examples. The examples used here are supplied in the *chapters/advanced-topics/xquery-testing* folder of the *book-code* Git repository (see "Getting the Source Code" on page 15). Imagine that you have an XQuery library module for producing identifiers, as shown in Example 16-1 (see the file *id.xqm*).

Example 16-1. Simple module for producing identifiers (id.xqm)

```
xquery version "3.0";

module namespace id = "http://example.com/record/id";

import module namespace util = "http://exist-db.org/xquery/util";

declare variable $id:ERR-PRESENT := xs:QName("id:ERR-PRESENT");


declare function id:insert($record as element(record)) as element(record) {
  if($record/id)then
    fn:error($id:ERR-PRESENT, "<id> is already present in record!", $record/id)
  else
    <record>
    {
      $record/@*,
      <id>{id:generate()}</id>,
      $record/node()
    }
    </record>
};

declare function id:generate() as xs:string {
  let $id := id:random()
  return
    if(exists(collection("/db/records")/record/id[. eq $id]))then
      id:generate()
    else
      $id
};

declare function id:random() as xs:string {
    codepoints-to-string(
        ((1 to 8) ! util:random(26))
          ! (.[. lt 10] + 48, .[. ge 10] + 87)
    )
};
```

Let's first write some test cases for the function `id:insert`. Here are three things that leap to mind that we might like to write test cases for:

- If we send it a document that already has an `id` element, it raises the error `id:ERR-PRESENT`.
- If we send it a document without an `id` element, it adds an `id` element for us with some sort of identifier.

- If we send it a document without an `id` element, it adds an `id` element for us with some sort of identifier; otherwise, the result document is indiscernible from the original.

  That is to say, it has the same *descendant nodes* (we check this as the function is really creating a copy of the input and modifying it).

Arguably, the second and third test cases just described have some overlap and could be merged into one. However, having two separate tests gives us more granularity in understanding the problems if or when tests fail. For instance, it is entirely possible that the second test case could pass while the third test case fails, which tells us the issue is with copying the descendant nodes and not with the generation of the `id` element. Writing fine-grained tests that allow you to quickly discover where bugs or regressions occur can help expedite bug fixes.

Let's now look at how we might write our first test case, which was:

> If we send it a document that already has an `id` element, it raises the error `id:ERR-PRESENT`.

We will begin by modifying the `id:insert` function by adding some XQSuite annotations (see the file *id-1.xqm*), as shown in Example 16-2.

Example 16-2. id.xqm with first test case (id-1.xqm)

```
xquery version "3.0";

module namespace id = "http://example.com/record/id";

import module namespace util = "http://exist-db.org/xquery/util";
declare namespace test = "http://exist-db.org/xquery/xqsuite";

declare variable $id:ERR-PRESENT := xs:QName("id:ERR-PRESENT");


declare
  %test:args("<record><id>existing</id></record>") ❶
    %test:assertError("id:ERR-PRESENT") ❷
function id:insert($record as element(record)) as element(record) {

  if($record/id)then
    fn:error($id:ERR-PRESENT, "<id> is already present in record!", $record/id)
  else
    <record>
    {
      $record/@*,
      <id>{id:generate()}</id>,
      $record/node()
    }
    </record>
};

(: Unchanged remaining code omitted for brevity... :)
```

❶ The `%test:args` annotation provides values to the function's arguments when the test case is run. If your function takes more than one argument, you can supply them one after another—for example, `%test:args("arg1", "arg2", "argN")`.

❷ The `%test:assertError` annotation asserts that, for the previously provided `args`, the function must throw the error (code) that is named—in this case, `id:ERR-PRESENT`.

OK, so now we have seen how to annotate our function with some parameters for our test case and an assertion about the expected result of executing that function using those parameters. However, we have not yet run our test case—so how can we actually have eXist execute our test case and return a report of whether it succeeded or failed? Well, of course, we have to write another little bit of XQuery! We'll create an XQuery main module so we can directly execute it, and from this XQuery we will invoke XQSuite against the functions in our XQuery library module, *id-1.xqm*. Such an XQuery main module is known as a *test runner* (see the file *test-runner.xq*) and is demonstrated in Example 16-3.

Example 16-3. Test runner XQuery (test-runner.xq)

```
xquery version "3.0";

import module namespace inspect = "http://exist-db.org/xquery/inspection"; ❶
import module namespace test = "http://exist-db.org/xquery/xqsuite" at
  "resource:org/exist/xquery/lib/xqsuite/xqsuite.xql"; ❷

let $modules := (
  xs:anyURI("/db/apps/exist-book/chapters/advanced-topics/xquery-testing/id-1.xqm")
) ❸
let $functions := $modules ! inspect:module-functions(.) ❹
return
  test:suite($functions) ❺
```

❶ We import the eXist XQuery inspection module, so that we can reflectively find all the functions in our module we wish to test.

❷ We import the XQSuite test framework XQuery module, so that we can subsequently run our test suite.

❸ We prepare a sequence of URIs of all modules that we are interested in testing. You can add more modules to this sequence if you wish.

❹ We call `inspect:module-functions` for each module we wish to test, to get a list of all functions in all our modules.

❺ We call `test:suite`, passing in all the functions. XQSuite will operate only on those functions that have XQSuite annotations, so we need not worry specifically about which functions we pass in.

Running the test runner should yield an xUnit result document that looks something like:

```
<testsuites>
    <testsuite package="http://example.com/record/id"❶
      timestamp="2014-07-02T17:55:43.747+01:00"
      failures="0" ❷
      tests="1" ❸
      time="PT0.023S"> ❹

        <testcase name="insert" class="id:insert"/>

    </testsuite>
</testsuites>
```

❶ Note that the `package` name used in the xUnit output matches the namespace of our module under test.

❷ Here we can see how many of our tests failed. In this case there were `0 failures`, so everything must have passed (succeeded).

❸ Here we can see the total number of tests run. In this case there was `1 test` run. The number of tests passed is calculated by `passed = tests - failures`; therefore, there was `1 passed` test.

❹ Here you can see the time taken to run a specific test suite. This can be useful for measuring performance loss or gains when refactoring code.

Great! Now we have one test case that we can run that helps us prove the correctness of our code and guard against regressions. What if we want more than one test case per function? Let's now look at how we would add our second test case, which was:

> If we send it a document without an `id` element, it adds an `id` element for us with some sort of identifier.

We again modify our `id:insert` function to insert the additional test case (see the file *id-2.xqm*), as shown in Example 16-4.

Example 16-4. id.xqm with two test cases (id-2.xqm)

```
declare
  %test:args("<record><id>existing</id></record>")
    %test:assertError("id:ERR-PRESENT")
    %test:args("<record/>") ❶
    %test:assertXPath("$result/exists(id)") ❷
    %test:assertXPath("not($result/empty(id))") ❸
function id:insert($record as element(record)) as element(record) {
  if($record/id)then
    fn:error($id:ERR-PRESENT, "<id> is already present in record!", $record/id)
  else
    <record>
    {
      $record/@*,
      <id>{id:generate()}</id>,
      $record/node()
    }
    </record>
};
```

❶ We have added a second `%test:args` annotation to our function for our second test case. Every `%test:args` or set of `%test:arg` annotations, delimited by one or more `assert` annotations, forms a distinct test case.

❷❸ For this test case we have two assertions—first that the result contains an element `id`, and second that the `id` has some child content. You may have as many assertions about the result of the function for each test case as you wish.

In this test case we are using `%test:assertXPath` instead of `%test:assertError`, as before. `%test:assertXPath` allows us to evaluate an arbitrary XPath expression against the `$result` of the function.

Finally, let's look at how we would add our final test case, which was:

> If we send it a document without an `id` element, it adds an `id` element for us with some sort of identifier; otherwise, the result document is indiscernible from the original.

We again modify our `id:insert` function to add the final test case (see the file *id-3.xqm*), as shown in Example 16-5.

Example 16-5. id.xqm with three test cases (id-3.xqm)

```
declare
  %test:args("<record><id>existing</id></record>")
    %test:assertError("id:ERR-PRESENT")

  %test:args("<record/>")
    %test:assertXPath("$result/exists(id)")
    %test:assertXPath("not($result/empty(id))")

  %test:args("<record a='1'><child1>text1</child1></record>") ❶
    %test:assertXPath("$result/exists(id)") ❷
    %test:assertXPath("$result/@a eq '1'") ❸
    %test:assertXPath("local-name(($result/child::element())[1]) eq 'id'") ❹
    %test:assertXPath("local-name(($result/child::element())[2]) eq 'child1'") ❺
    %test:assertXPath("$result/child1/text() eq 'text1'") ❻
function id:insert($record as element(record)) as element(record) {
  if($record/id)then
    fn:error($id:ERR-PRESENT, "<id> is already present in record!", $record/id)
  else
    <record>
    {
      $record/@*,
      <id>{id:generate()}</id>,
      $record/node()
    }
    </record>
};
```

❶ We set the argument for the function for our test case to be an XML element that contains both attributes and descendant nodes, as we want to make sure the result is properly constructed.

❷ This is the same assertion as from our last test case, to ensure that an `id` element is added to the record.

❸❹❺❻ These are several assertions to ensure that the `record` element returned by the function contains all of the nodes in the same order that the `record` element provided them as the argument to the function.

Executing our test runner again (*test-runner.xq*) now produces an xUnit result document similar to the following:

```
<testsuites>
  <testsuite package="http://example.com/record/id"
   timestamp="2014-07-02T18:26:29.784+01:00"
   failures="0" tests="3" time="PT0.064S"> ❶
    <testcase name="insert" class="id:insert"/> ❷
    <testcase name="insert" class="id:insert"/> ❸
    <testcase name="insert" class="id:insert"/> ❹
  </testsuite>
</testsuites>
```

❶ We can now see that all three of our test cases are executing.

❷❸❹ Unfortunately, at present XQSuite shows the same `name` for all of our test cases.

We have now written our three test cases for our `id:insert` function, but what about the `id:generate` and `id:random` functions? Well, of course, we could write some further test cases for these using XQSuite annotations—but wait a minute, maybe there are some difficulties in testing these functions! These two functions have in fact been written to illustrate problems that can arise in writing test cases.

First, let's consider the issues with the `id:random` function. This function, as its name implies, will return something *random* each time it is called (in this instance, an eight-character alphanumeric string). This function exhibits a trait known as *nondeterministic behavior*, which means that each time the function is called, it may produce a different result. This makes testing harder, as in this case we cannot make assertions about the exact return value of the function using the XQSuite annotation `%test:assertEquals`. Instead, we can only make generalizations such as "The result must be a string of 8 characters" and "The result can only contain the characters `a` to `z` and `0` to `9`." Unfortunately, this nondeterminism is spread throughout our identifier's module, as each function eventually calls `id:random`. We could solve this in some test frameworks in other languages by introducing *mocks* that would effectively replace the underlying call to `util:random` with some static value, which we could then use as the basis for assertions about the deterministic results of our tests—that is, asserting that our algorithm for encoding from numbers into a charset is entirely correct. Regrettably, XQSuite does not yet support function mocks.

So, what about testing our `id:generate` function? Well, again, we have the nondeterminism problems of `id:random`, but here they are amplified by the contents of a database collection because the database itself is in a *mutable state*, which can lead to even worse nondeterministic results. Again, if only function mocking were supported, we could provide a deterministic substitute for the database collection.

> There are also other potential problems with how `id:generate` is implemented and would be used in a multiuser system, but we will leave that as a mental exercise for the reader.

However, all is not lost! If we were to change our `id:generate` function to take a path to a collection as an argument rather than the currently hardcoded one, perhaps we could create a *test collection* and prime it with deterministic data for our test case, before we run it. Afterward, when we are done with the test data, we would be courteous and remove our test collection, as is good practice. Indeed, we can do such a thing by providing custom `setup` and `teardown` functions that run *before* and *after* our test functions, respectively. To declare a function as a `setup` or `teardown` function, you can use the `%test:setUp` and `%test:tearDown` annotations. So let's look briefly at a refactored `id:generate` function with test cases (see the file *id-4.xqm*), as shown in Example 16-6.

Example 16-6. Refactored id.xqm to inject collection path (id-4.xqm)

```
declare
  %test:setUp ❶
function id:_test-setup() {
  xmldb:create-collection("/db", "test-records"),
  xmldb:store("/db/test-records", (), <record><id>12345678</id></record>),
  xmldb:store("/db/test-records", (), <record><id>abcdefgh</id></record>)
};

declare
  %test:tearDown ❷
function id:_test-teardown() {
  xmldb:remove("/db/test-records")
};


declare
  %test:args("/db/test-records") ❸
    %test:assertXPath("$result ne '12345678'") ❹
    %test:assertXPath("$result ne 'abcdefgh'") ❺
function id:generate($records-collection as xs:string) as xs:string { ❻
  let $id := id:random()
  return
    if(exists(collection($records-collection)/record/id[. eq $id]))then
      id:generate($records-collection)
    else
      $id
};
```

❶ The `%test:setUp` annotation will cause the `id:_test-setup` function to be executed once *before* each function under test.

In this instance, we create the test data collection */db/test-records* and place two records in it containing the ids `12345678` and `abcdefgh`.

❷ The `%test:tearDown` annotation will cause the `id:_test-teardown` function to be executed once *after* each function under test.

In this instance, we clean up the test data we created in the *before* step by removing the collection */db/test-records*.

❸ We inject the test data collection */db/test-records* into our function under test as an argument.

❹❺ We make assertions that the function should never return the `ids` of the records in the test data collection.

Note that this is quite a brittle set of test assertions, as the `ids` are generated randomly, so we may never hit the edge cases. Hopefully, though, it shows you what is possible.

❻ Our newly refactored function now accepts the path to the records collection as a parameter, which allows us to use our test data collection when testing, and the real collection otherwise.

We have really only scratched the surface of both the larger topic of testing and the specifics of XQSuite here, but hopefully we've provided enough information to start you thinking about testing and XQuery code quality. XQSuite provides several other annotations that allow you to pass parameters in different ways and make different types of assertions about the results of a function. For further information, consult the XQSuite documentation.

Versioning

eXist provides two simple versioning mechanisms, which, while not applicable to all use cases, can be useful to those who wish to track the revision history of certain types of documents. Both mechanisms can be configured on a per-collection hierarchy basis, which enables you to switch versioning on and off for various document collections within your database.

Historical Archiving

The historical archival facility will make an archival copy of any document before it is deleted or overwritten. The archival copy will be placed into a mirror of the collection tree under the archival collection */db/history* by default.

While this is not versioning in the strictest sense, it can be valid for many applications. Arguably, you could also achieve basic versioning, if all updates to a document are performed by the user as document replacements. However, the real purpose of this facility is to effectively make documents immutable for archival purposes.

Historical archiving is implemented in a document trigger written in Java called `org.exist.collections.triggers.HistoryTrigger`. For more information on database triggers and configuring them, see "Database Triggers" on page 449. If you wish to configure the `HistoryTrigger` for a collection in your database, you need to add the following to the triggers section of your collection's configuration document (*collection.xconf*):

```
<triggers>
    <trigger class="org.exist.collections.triggers.HistoryTrigger">

        <!--
            Collection to store the archival copies in;
            if omitted, then the collection /db/history
            is used.
        -->
        <parameter name="root" value="/db/system/archival"/>
    </trigger>
</triggers>
```

Archival copies will be stored into a mirror of their original collection path within the archival collection. In addition, a collection is created for each document using its name, and the archival copy is stored within this collection using a timestamp for its name that reflects the previous last-modified date of the document.

Document Revisions

The document revision versioning mechanism in eXist is much more comprehensive than the historical archiving mechanism (described in the previous section).

However, you should note that it is not well suited to large collections of data-centric XML or binary documents; rather, it is designed with human editors in mind and for use with modest collections of XML documents. For example, if you are working as part of a team with published articles or humanities texts, then the versioning mechanism may be useful for you, but if you are streaming gigabytes of server log data into eXist it may not scale to your needs.

The versioning mechanism in eXist works well enough for many cases, but while eXist knows the version history of your documents, there is little support from XML editor tools that connect to eXist to expose this information to the end user. eXist does provide some basic tools for you to visually examine the version history of documents and also to interrogate this from XQuery, but these tools are rudimentary at best. If you wish to use the versioning mechanism in a larger enterprise, you might want to consider building your own tools atop the XQuery functions that eXist exposes.

The document revision versioning mechanism has two main parts:

- The versioning actions are implemented in a document trigger written in Java called `org.exist.collections.triggers.VersioningTrigger`. For more information on database triggers and configuring them, see "Database Triggers" on page 449.

- Version write conflict avoidance is implemented in a serialization filter written in Java called `org.exist.versioning.VersioningFilter`. The use of the filter is optional, but it potentially stops two users from editing the same document outside of eXist and overwriting each other's changes.

When the `VersioningTrigger` is configured on a collection, it will store information about revisions to documents within that collection hierarchy in a mirrored collection hierarchy, under */db/system/versions*. Specifically, the version trigger performs the following operations on various document events:

- The first time a document is changed:
  - Before it is changed, a copy will be made and stored into the appropriate mirrored collection within */db/system/versions*; it will have the same filename as the original but with the additional suffix *.base*. This is known as the *document base revision*.
  - After it is changed, a document describing the differences from the document base revision to the new revision is also stored into the appropriate mirrored collection within */db/system/versions*; it will have the same filename as the original but with an additional ordinal suffix indicating the revision number. This is known as a *diff document*. These documents use an XML format that is specific to eXist. For binary documents we can only say that one document replaced another, so we also store a copy of the new binary document with the ordinal suffix and an additional *.binary* suffix. For XML documents the diff document also describes node-level *insertion*, *deletion*, and *append* operations; a *change* is modeled as a `delete` and `insert`.
- The 1+*n* time a document is changed:
  - After it is changed, like the first time it is changed, a document describing the differences from the document base revision to the new revision is again stored into the versions collection, with an ordinal suffix (and if it is a binary document, then that is likewise stored again with the *.<ordinal>.binary* suffix). Just as before, this is known as a diff document; however, it is worth noting that every diff is between the current document and its document base revision, not the most recent revision.

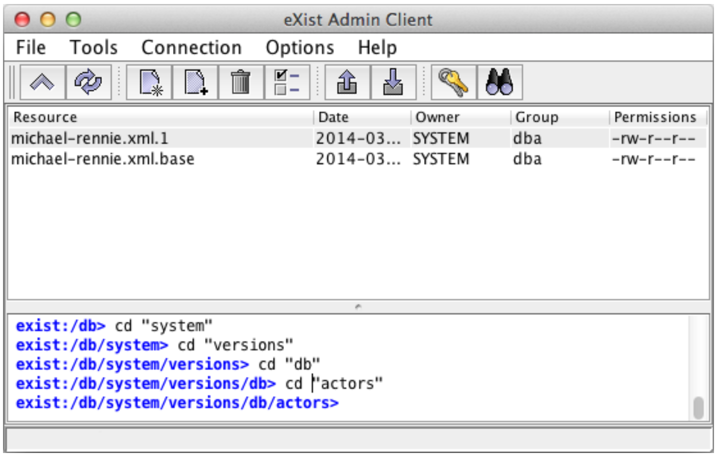
Diff documents are absolute rather than incremental. The advantage of this is that it is very simple to understand the changes from the original document to the current document. The downside, however, is that diff documents become increasingly large over time, and you have to replay each diff independently to see the changes over time (which repeats many operations).

- When a document is copied or moved, the behavior just described is applied to the destination document (assuming it is also in a versioned collection).

For example, say we have the collection */db/actors* for which we have enabled the `VersioningTrigger`, and we have a document stored in that collection called *michael-rennie.xml*, which looks similar to:

```
<actor>
    <name>
        <given>Michael</given>
        <family>Rennie</family>
    </name>
    <born>1909</born>
    <desceased>1971</desceased>
    <abstract>The British actor Michael Rennie worked as a car salesman and factory
      manager before he turned to acting.</abstract>
</actor>
```

Ah, but then we realize that we have spelled *deceased* wrong in our element, and decide to update the document to fix this. Subsequently, we then see the documents shown in Figure 16-1 in the collection */db/system/versions/db/actors*.



```
Resource                  Date          Owner    Group   Permissions
michael-rennie.xml.1      2014-03...    SYSTEM   dba     -rw-r--r--
michael-rennie.xml.base   2014-03...    SYSTEM   dba     -rw-r--r--
```

```
exist:/db> cd "system"
exist:/db/system> cd "versions"
exist:/db/system/versions> cd "db"
exist:/db/system/versions/db> cd "actors"
exist:/db/system/versions/db/actors>
```

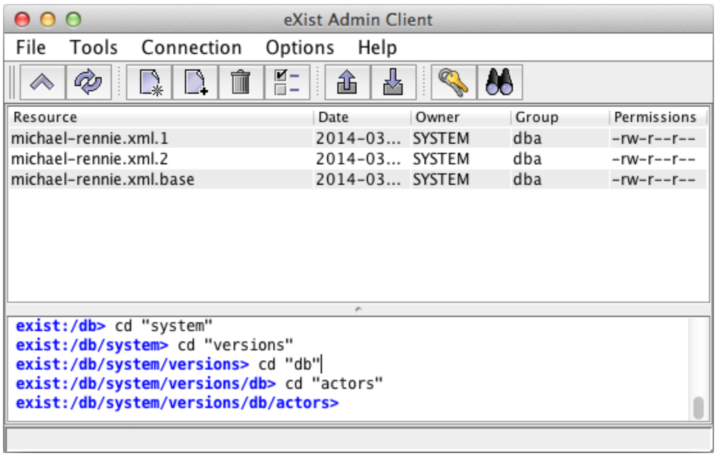
Figure 16-1. Versioning trigger, first revision documents

While the *michael-rennie.xml.base* document is just a copy of the original XML document, the *michael-rennie.xml.1* document is the diff document, whose ordinal suffix immediately indicates to you that it is the first revision. The content of the document looks like:

```
<v:version xmlns:v="http://exist-db.org/versioning">
    <v:properties>
        <v:document>michael-rennie.xml</v:document>
        <v:user>admin</v:user>
        <v:date>2014-03-09T14:59:18.954Z</v:date>
        <v:revision>1</v:revision>
    </v:properties>
    <v:diff>
        <v:delete event="start" ref="1.3"/>
        <v:append ref="1.3">
            <v:end name="deceased"/>
        </v:append>
        <v:delete event="end" ref="1.3"/>
        <v:insert ref="1.3">
            <v:start name="deceased"/>
        </v:insert>
    </v:diff>
</v:version>
```

As expected, the diff document clearly shows the rename as a process of deleting the element with the incorrect name and then replacing it with a new one with the correct name.

Finally, let's update the document by adding some extra information to the `abstract` element. Following the update, we then see the documents shown in Figure 16-2 in the collection */db/system/versions/db/actors*.



Resource	Date	Owner	Group	Permissions
michael-rennie.xml.1	2014-03...	SYSTEM	dba	-rw-r--r--
michael-rennie.xml.2	2014-03...	SYSTEM	dba	-rw-r--r--
michael-rennie.xml.base	2014-03...	SYSTEM	dba	-rw-r--r--

```
exist:/db> cd "system"
exist:/db/system> cd "versions"
exist:/db/system/versions> cd "db"
exist:/db/system/versions/db> cd "actors"
exist:/db/system/versions/db/actors>
```

Figure 16-2. Versioning trigger, second revision documents

There is now a second diff document (*michael-rennie.xml.2*), and if we examine it, its content should look something like this:

```
<v:version xmlns:v="http://exist-db.org/versioning">
    <v:properties>
        <v:document>michael-rennie.xml</v:document>
        <v:user>admin</v:user>
        <v:date>2014-03-09T15:15:28.342Z</v:date>
        <v:revision>2</v:revision>
    </v:properties>
    <v:diff>
        <v:delete event="start" ref="1.3"/>
        <v:append ref="1.3">
            <v:end name="deceased"/>
        </v:append>
        <v:delete event="end" ref="1.3"/>
        <v:delete ref="1.4.1"/>
        <v:insert ref="1.3">
            <v:start name="deceased"/>
        </v:insert>
        <v:insert ref="1.4.1">The British actor Michael Rennie worked as a car
  salesman and factory manager before he turned to acting. A meeting
  with a Gaumont-British Studios casting director led to Rennie's first
  acting job - that of stand-in for Robert Young in Secret Agent
  (1936), directed by Alfred Hitchcock.</v:insert>
    </v:diff>
</v:version>
```

We can see that the second diff document contains both the changes we made in this revision (i.e., updated text in the abstract element) as well as all changes from previous revisions. This is because, as previously mentioned, each diff document is between the current document and the document base revision.

If you wish to configure the VersioningTrigger for a collection in your database, you need to add the following to the triggers element within your collection's configuration document (*collection.xconf*):

```
    <triggers>
      <trigger class="org.exist.versioning.VersioningTrigger">

        <!--
        Whether to try and avoid
        write conflicts on versioned
        documents.

        Set to 'false' to avoid write conflicts,
        or 'true' to ignore write conflicts and
        overwrite the later revision.
        -->
        <parameter name="overwrite" value="false"/>

      </trigger>

    </triggers>
```

It is worth noting that you can also interrogate document revisions from XQuery by using the versioning XQuery module, as discussed in versioning.

Write conflict avoidance

Being able to edit a document and have a history of revisions created for you is all well and good, but what happens if multiple people start editing the document at the same time, and all wish to save their changes independently?

When two users open the same document—which is, say, at revision 1—and then make changes, when the first user saves the document he is effectively creating revision 2. What should happen when the second user attempts to save her update of revision 1? Mitigating the issue of revision 2 being overwritten with a new version based on revision 1 is known in eXist as *write conflict avoidance*.

As most XML editor clients that connect to eXist have no awareness of resource versioning within eXist, if you enable write conflict avoidance, eXist solves that problem by prohibiting later revisions from being overwritten by updates to earlier revisions. Figure 16-3 attempts to show how eXist solves this.

As shown in Figure 16-3, write conflict avoidance comes in two parts:

1. The versioning trigger can be configured to avoid changes being overwritten by changes to an earlier revision. However, the versioning trigger can respond only to changes that it knows about. As the changes are potentially coming from a third-party client application, we need some mechanism to identify revisions within the documents that are being accessed; this is where the versioning filter comes in.
2. The versioning filter can be configured to add versioning attributes to the root element of any document that is retrieved from eXist and is from a collection that has the versioning trigger enabled on it. When this document is sent back to eXist, the versioning trigger will see the versioning attributes on the root element, act on them, and, if there is no conflict, remove them before storing the document into the database. If there is a conflict, it will reject the operation and prohibit the document from being stored.

If you wish to enable write conflict avoidance, you need to set the `overwrite` parameter on the versioning trigger to `false` in the collection configuration document (*collection.xconf*), enable the versioning filter in eXist's configuration file (*$EXIST_HOME/conf.xml*), and then restart eXist. To enable the versioning filter, make sure the following definition is present and uncommented in the `serializer` element of the configuration file:

```
<custom-filter class="org.exist.versioning.VersioningFilter"/>
```

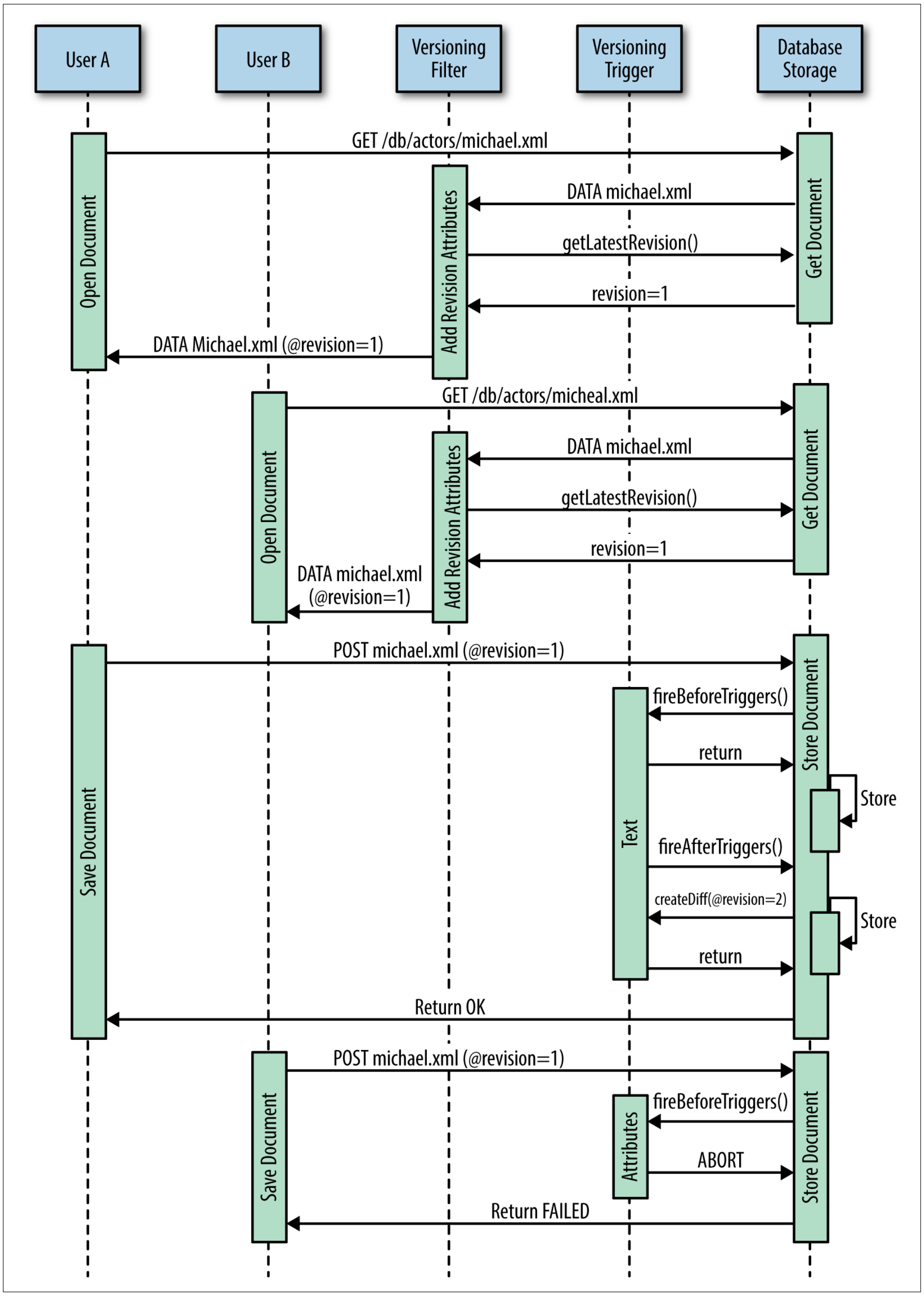


Figure 16-3. Write conflict avoidance between two users operating on the same versioned document

Once the versioning filter is enabled, if you request a versioned document from the database you will see three additional attributes on its root element: `v:revision`, `v:key`, and `v:path`. For example:

```
<actor xmlns:v="http://exist-db.org/versioning" v:revision="2"
  v:key="144a903cec62" v:path="/db/actors/michael-rennie.xml">
    <name>
        <given>Michael</given>
        <family>Rennie</family>
    </name>
    <born>1909</born>
    <deceased>1971</deceased>
    <abstract>The British actor Michael Rennie worked as a car salesman
  and factory manager before he turned to acting. A meeting with a
  Gaumont-British Studios casting director led to Rennie's first acting
  job - that of stand-in for Robert Young in Secret Agent (1936),
  directed by Alfred Hitchcock.</abstract>
</actor>
```

While adding the versioning attributes to the root element of the document is not ideal, it is really the only way that eXist can attempt to ensure that this information is preserved during a round-trip of the document out of the database, into a client editor, and back out again. The versioning attributes, however, are in their own namespace and so hopefully will not interfere with your document.

If you are making use of the versioning filter, you must not remove the versioning attributes from the document if you plan to store it back into eXist in the same location, or you risk losing previous changes from the revision history!

Scheduled Jobs

eXist has a scheduler built into its core that enables you to schedule jobs to be executed at some point(s) in the future. Internally eXist wraps the Quartz scheduler, but it exposes a much simpler interface to the user and allows jobs to interact with the database. You can write your jobs in either XQuery or Java. XQuery jobs are simpler to implement ,while Java jobs give you more control over how the job is executed. There are two types of job that can be executed by the scheduler:

User jobs
: This is the standard job type that users will typically implement in either XQuery or Java. User jobs may execute concurrently, and the same job may overlap with a previously scheduled execution if it is long-running.

System task jobs
: System task jobs are solely for executing system tasks and may only be implemented in Java. System tasks execute when the database is switched into *pro-*

tected mode; no other transactions are permitted. System task jobs do not execute concurrently due to the restrictions on system tasks, and they cannot overlap with a previously scheduled execution. eXist makes use of the system task job both to schedule its synchronization task, which flushes the database journal to persistent storage, and to execute scheduled backups.

Scheduling Jobs

You can schedule jobs by setting up their configuration with the scheduler in eXist's configuration file (*$EXIST_HOME/conf.xml*): you add jobs to the `scheduler` element indicated by the XPath */exist/scheduler*. Remember that changes to the configuration file are only read when eXist is started.

So, what can you do if you want to schedule a job to run immediately, without restarting eXist?

Simply, you must add it to *$EXIST_HOME/conf.xml* so that it is persisted across restarts, and also submit it using the `scheduler` XQuery extension module (see `scheduler`) so that it is scheduled immediately without needing to restart.

Enabling the scheduler XQuery Extension Module

The `scheduler` XQuery extension module (see `scheduler`) in eXist is not compiled or enabled by default. To enable it, you need to:

1. Stop eXist.
2. Set `include.module.scheduler` to `true` in *$EXIST_HOME/extensions/build.properties*.
3. Recompile eXist's extension modules in place (see "Building eXist from Source" on page 485). Make sure to use the `extension-modules` Ant target!
4. Enable the module in eXist's configuration file (*$EXIST_HOME/conf.xml*) by uncommenting the line `<module uri="http://exist-db.org/xquery/scheduler" class="org.exist.xquery.modules.scheduler.SchedulerModule"/>`.
5. Start eXist.

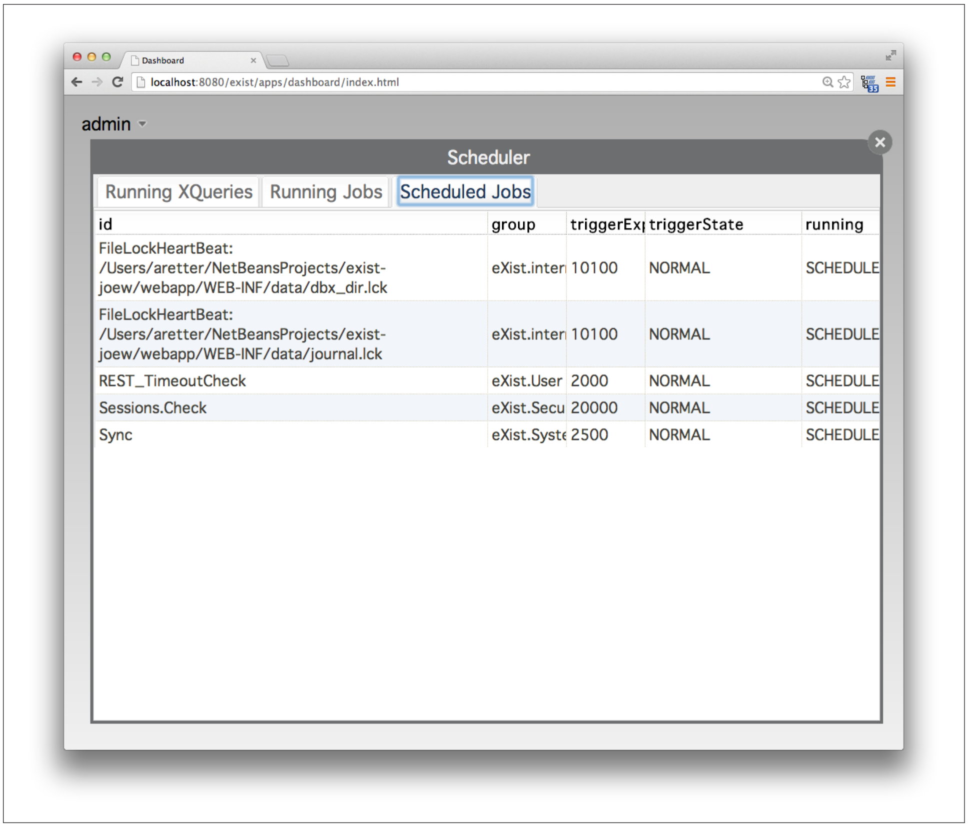
Table 16-1 lists the scheduled job arguments.

Table 16-1. Scheduled job arguments

Argument	Description	Mandatory/optional
`type`	The type of the job to schedule. Either `system` for system task jobs or `user` for user jobs.	Mandatory.
`class`	Used for specifying the fully qualified class name of your Java class that implements either `org.exist.scheduler.UserJavaJob` or `org.exist.storage.SystemTask`.[a]	Mandatory if job is implemented in Java.
`xquery`	Used for specifying the database path to an XQuery if you have implemented your scheduled job in XQuery. The syntax uses the simple database path (e.g., */db/my-collection/my-job.xq*).	Mandatory if job is implemented in XQuery.
`name`	When scheduling an XQuery job, you can provide a friendly name for the scheduled job, but it must be unique across all scheduled jobs. Java jobs implement their own name function.	Optional if XQuery job, and otherwise ignored. The default is the name of the XQuery file.
`unschedule-on-exception`	When you're executing an XQuery job, if the job causes some sort of exception it can be unscheduled so that it does not run again in the future. Must be either `yes` or `no`.	Optional if XQuery job, and otherwise ignored. The default is `yes`.
`cron-trigger`	A description of when the scheduled job is run, using a cron-like syntax. For the exact syntax, see *http://www.quartz-scheduler.org/documentation/quartz-1.x/tutorials/crontrigger*.	Mandatory, or use `period` instead.
`period`	A period in milliseconds defining the frequency after any `delay` with which the job is run.	Mandatory, or use `cron-trigger` instead.
`delay`	A startup delay in milliseconds after which the scheduled job is first run. If unspecified, the job will be executed immediately after eXist has initialized.	Optional; use with `period`.
`repeat`	The number of times to repeat execution of the job. If unspecified, the job will be executed periodically indefinitely.	Optional; use with `period`.

[a]Your Java class must be available on eXist's classpath, which you typically accomplish by placing your JAR and any JAR dependencies into *$EXIST_HOME/lib/user*.

You may view the currently scheduled and executing jobs either by using the Scheduler dashboard application (Figure 16-4) or by executing the `scheduler:get-scheduled-jobs` XQuery extension function.



id	group	triggerEx	triggerState	running
FileLockHeartBeat: /Users/aretter/NetBeansProjects/exist-joew/webapp/WEB-INF/data/dbx_dir.lck	eXist.inter	10100	NORMAL	SCHEDULE
FileLockHeartBeat: /Users/aretter/NetBeansProjects/exist-joew/webapp/WEB-INF/data/journal.lck	eXist.inter	10100	NORMAL	SCHEDULE
REST_TimeoutCheck	eXist.User	2000	NORMAL	SCHEDULE
Sessions.Check	eXist.Secu	20000	NORMAL	SCHEDULE
Sync	eXist.Syste	2500	NORMAL	SCHEDULE

Figure 16-4. Dashboard Scheduler application, showing scheduled jobs

XQuery Jobs

Writing user jobs in XQuery that are executed periodically by the scheduler is not much different from writing normal XQuery. The scheduler can execute only XQuery main modules—that is, it cannot directly call a specific XQuery function—but this should not be a problem as you can always write a one-line XQuery main module that acts as a wrapper for your library function.

Sometimes you need to be able to parameterize an XQuery that the scheduler will execute, perhaps with some configuration settings. The scheduler fits well with the standard mechanism for passing external configuration into XQuery, which is to use variable declarations with external binding. In the configuration of the XQuery scheduled job, you may set parameters, each of which will attempt to bind to the equivalently named external variable declared in your XQuery. By default, the external variables are expected to be bound to the local namespace of the XQuery and use the binding prefix `local`. If you wish to change this, you can use the special parameter `bindingPrefix` in the configuration of the XQuery scheduled job.

Scheduled weather retrieval (XQuery)

Supplied alongside this chapter is an XQuery file called *weather.xq*, in the *chapters/advanced-topics* folder of the *book-code* Git repository (see "Getting the Source Code" on page 15). This XQuery has been designed as a simple example of what you can achieve with a scheduled XQuery job. The XQuery simply connects to a public web service and downloads the current weather for a particular city, parses the results, and stores them into the database. By scheduling this XQuery, you can build up a dataset of weather over time, which you can then later query to understand how the weather changed.

To use the example, you must store the query into the database (for example, at */db/weather.xq*), set it as executable by the `guest` user, create a collection for storing weather data, and make that writable by the `guest` user. You then need to add the scheduled job configuration shown in Example 16-7 to *$EXIST_HOME/conf.xml* and restart eXist.

Example 16-7. Scheduled configuration for the weather example

```
<job type="user" xquery="/db/weather.xq" name="hourly-weather"
  cron-trigger="0 0 0/1 * * ?">
  <parameter name="city" value="Exeter"/>
  <parameter name="country" value="United Kingdom"/>
  <parameter name="weather-collection" value="/db/weather"/>
</job>
```

This scheduler configuration will cause the XQuery */db/weather.xq* to be executed every hour.

Alternatively, you could schedule it using the `scheduler:schedule-xquery-cron-job` XQuery extension function, as shown in Example 16-8.

Example 16-8. Immediate scheduling for the weather example

```
scheduler:schedule-xquery-cron-job(
    "/db/weather.xq",
    "0 0 0/1 * * ?",
    "hourly-weather",
    <parameters>
        <param name="city" value="Exeter"/>
        <param name="country" value="United Kingdom"/>
        <param name="weather-collection" value="/db/weather"/>
    </parameters>
)
```

The code for the *weather.xq* scheduled weather web service query is listed in Example 16-9.

Example 16-9. The weather web service scheduled query (weather.xq)

```
xquery version "3.0";

import module namespace http = "http://expath.org/ns/http-client";
import module namespace util = "http://exist-db.org/xquery/util";
import module namespace xmldb = "http://exist-db.org/xquery/xmldb";

declare namespace wsx = "http://www.webserviceX.NET";

(: Configuration :)
declare variable $local:city external; ❶
declare variable $local:country external; ❷
declare variable $local:weather-collection external; ❸


let $webservice := "http://www.webservicex.net/globalweather.asmx/GetWeather",
$url := $webservice || "?CityName=" || encode-for-uri($local:city) ||
    "&CountryName=" || encode-for-uri($local:country) , ❹
$result := http:send-request(<http:request href="{$url}" method="get"/>) ❺
return

    let $doc := if($result[1]/@status eq "200" and $result[2]/wsx:string) then ❻
            (: reconstruct XML, the webservice provides it as a string
            for some reason! :)

        util:parse($result[2]/wsx:string/text()) ❼
        else
            (: record failure :)

        <failed at="{current-dateTime()}">{$result}</failed> ❽
    return
        let $stored := xmldb:store($local:weather-collection, (), $doc) ❾

        return
            (: log that we ran! :)
            util:log("debug", "Stored hourly weather to: " || $stored) ❿
```

❶❷❸ These externally bound variables are filled by the parameters to the scheduled job configuration.

❹ We prepare the full URL for the weather web service call.

❺ We make a call to the weather web service using the EXPath HTTP Client extension module (see `http`).

❻ We check whether the web service call succeeded or failed.

❼ If the web service call succeeded, we extract the weather data.

❽ If the web service call failed, we prepare some failure data.

❾ We store the result into the database.

❿ We write a message to the logfile indicating that the task ran to completion.

While you are typically interested in the final computed result of your XQuery, when your query is running within the scheduler there is nowhere for the result to be implicitly sent to. If you wish to understand or retain the result of your query, you are responsible for either explicitly storing it into the database from the query using the `xmldb:store` XQuery extension function or writing it to a logfile using the `fn:trace` or `util:log` XQuery functions.

Java Jobs

As noted previously, if you wish to write a scheduled job in Java, you can implement either a *user job* or a *system task job*. For each option, eXist provides appropriate interfaces and abstract classes to assist in your implementation. Once you have implemented the appropriate class, you need to place your compiled class onto eXist's classpath, which you typically do by placing a JAR file of your code and any dependent JAR files into *$EXIST_HOME/lib/user*. Remember that you may have to restart eXist for the JVM class loader to see your new JAR files!

When writing a Java job, you must be careful to return any brokers that you borrow from the broker pool you are provided with and ensure that you release any locks that you have taken on collections or resources. Failure to do so can reduce the connections available to the database and leave resources inaccessible until eXist is restarted. It is strongly recommended to use the Java `try/catch/finally` pattern to release any acquired resources in the `finally` block. In addition, you should be aware that you cannot keep state in member variables of your `Job` class across invocations; instead, you should keep any nontransient state in a singleton (remember to synchronize access where needed) or store it into the database.

Java user job

Most often, developers will want to implement user jobs, which you do by extending the abstract class `org.exist.scheduler.UserJavaJob`. When extending this class, you must implement the following job naming, defined in `org.exist.scheduler.JobDescription`:

```
/**
 * Set the name of the job.
 * After being set, the job should return this name.
 *
 * @param name  The job's new name
 */
public String getName();

/**
 * Get the name of the job.
 * If a name has not yet been set, you must create one!
 *
 * @return The job's name
 */
public void setName(final String name);
```

These functions simply round-trip the *name* of the job. The job's name itself must be unique across all job instances. For Java jobs, you simply need to create an initial name and then return that each time `get` is called, or store the incoming name when `set` is called and then return that.

You also need to provide the actual work of the job in the form of implementing the abstract function defined in `org.exist.scheduler.UserJavaJob`:

```
/**
 * The actual work/task of the scheduled job.
 * This function is called each time the job is executed by the scheduler
 * according to its schedule.
 *
 * @param brokerpool The database BrokerPool.
 * @param params Any parameters passed to the job, or null otherwise.
 *
 * @throws JobException You may throw a JobException to control the
 *   cleanup of the job and affect rescheduling in case of a problem.
 */
  public abstract void execute(BrokerPool brokerpool, Map<String, ?> params)
  throws JobException;
```

When your code is executing, should you encounter a problem or exception, you must catch it, and if you wish to end processing you may throw an `org.exist.scheduler.JobException`. The `JobException` class takes an enumerated value of type `org.exist.scheduler.JobException.JobExceptionAction`, which controls how the job is aborted by the scheduler and whether its schedule should be adjusted. The available *job exception actions* when aborting a running job are outlined in Table 16-2.

Table 16-2. Job exception actions

Action	Description
JOB_ABORT	Instructs the scheduler to stop the current job. It may still be executed in the future according to its schedule.
JOB_ABORT_THIS	Instructs the scheduler to stop the current job. It also removes from the scheduler the schedule that triggered this job.
JOB_ABORT_ALL	Instructs the scheduler to stop the current job. It also removes from the scheduler the schedule that triggered this job and all other schedules that refer to this job.
JOB_REFIRE	Instructs the scheduler to stop the current job. It also reschedules the job for immediate execution. The schedule that triggered this job remains scheduled.

Scheduled weather retrieval (Java). An example class implementing `UserJavaJob` called `exist.book.example.scheduler.user.WeatherJob` is supplied with this chapter in the folder *chapters/advanced-topics/scheduler-java-job/weather-user-java-job-example* of the *book-code* Git repository (see "Getting the Source Code" on page 15). The example is an indirect port of the XQuery code in Example 16-9, and hopefully serves as a good comparison while showing that it is much simpler to implement a user job for the scheduler in XQuery than in Java (when possible). The example uses the Jersey client library for talking to the public weather web service.

To compile the example, enter the *scheduler-java-job* folder and run `mvn package`; the Java user job example can be found in the subfolder *weather-user-java-job-example*.

To deploy the `WeatherJob` to eXist, you need to create the collection */db/weather* (with write access by the `guest` user), stop eXist, and copy all of the files from *scheduler-java-job/weather-user-java-job-example/target/weather-user-java-job-example-1.0-assembly* to *$EXIST_HOME/lib/user*. You can then schedule the `WeatherJob` class to run hourly in *$EXIST_HOME/conf.xml* by adding the following job definition to the scheduler configuration before restarting eXist:

```
<job type="user" class="exist.book.example.scheduler.user.WeatherJob"
  name="hourly-weather" cron-trigger="0 0 0/1 * * ?">
  <parameter name="city" value="Exeter"/>
  <parameter name="country" value="United Kingdom"/>
  <parameter name="weather-collection" value="/db/weather"/>
</job>
```

As an alternative to adding the `WeatherJob` to eXist's configuration file, you could schedule nonpersistently by executing the following XQuery:

```
scheduler:schedule-java-cron-job(
    "exist.book.example.scheduler.user.WeatherJob",
    "0 0 0/1 * * ?",
    "hourly-weather",
    <parameters>
        <param name="city" value="Exeter"/>
        <param name="country" value="United Kingdom"/>
        <param name="weather-collection" value="/db/weather"/>
    </parameters>
)
```

Java system task job

It is unlikely that most developers will ever implement system task jobs, as you can meet the vast majority of use cases by instead implementing a user job. However, should you want to implement a scheduled system task job in Java, you do not actually need to implement the scheduler job aspect. Instead, you can just implement `org.exist.storage.SystemTask`, as eXist provides a generic system task job `org.exist.scheduler.impl.SystemTaskJobImpl` that enables any system task to be scheduled.

Remember that when your system task runs, you have exclusive access to the database, as the database will be in protected mode. All other operations will be *blocked until your task completes*. Accordingly, you should ensure that your task executes quickly and efficiently!

When implementing `org.exist.storage.SystemTask`, you must implement the following configuration functions:

```
/**
 * Called to configure the system task.
 * Enables you to configure your system task!
 * Note: If the system task is managed by the scheduler,
 * this happens before scheduling.
 *
 * @param config A reference to the parsed in-memory
 *   representation of eXist's configuration file $EXIST_HOME/conf.xml.
 * @param properties A property set containing any parameters passed
 *   to the scheduled job.
 *
 * @throws EXistException You may throw this to abort configuration of
 *   the system task. If the task is managed by the scheduler,
 *   it will not be scheduled.
 */
void configure(Configuration config, Properties properties)
  throws EXistException;
/**
 * Indicates whether a checkpoint should be generated
 * before the execute function is called. A checkpoint
 * guarantees that any outstanding changes are flushed to
```

```
     * persistent storage.
     *
     * @return true if a checkpoint should be generated, false otherwise
     */
    boolean afterCheckpoint();
```

You also need to provide the actual work of the system task job in the form of implementing the function:

```
    /**
     * Called when the system task is executed in protected mode.
     * Constitutes the work unit of the system task.
     *
     * @param broker A database broker that can be used to access the database.
     *
     * @throws ExistException You may throw this to indicate an error and abruptly
    *   abort executing of the system task. If the task is managed by the
    *   scheduler, this will not affect its future schedule.
     */
    void execute(DBBroker broker) throws EXistException;
```

System tasks do not need to acquire locks on collections and documents, as they are operating in protected mode—in fact, they should not attempt to do so! Invalid management of locks in a system task can lead to deadlock situations.

If you wish to acquire collections and documents without locks, you can use the `getCollection(XmldbURI)` method on `org.exist.storage.DBBroker` and then subsequently the `collectionIteratorNoLock(DBBroker)` and `iteratorNoLock(DBBroker)` methods on `org.exist.collections.Collection`.

Database stats scheduled system task. An example class implementing `SystemTask` called `exist.book.example.scheduler.system.StatsSystemTask` is supplied in the folder *chapters/advanced-topics/scheduler-java-job/stats-system-task-example* of the *book-code* Git repository (see "Getting the Source Code" on page 15). The example simply generates statistics about the current content of the database and stores them into a new document in a configured database collection. When the system task is scheduled, this allows you to build a collection of statistics about the content of the database over time, which could potentially be used by other XQueries to generate reports and/or graphs about database usage. The use of a system task here allows us to guarantee that our statistics represent the *exact state of the database at a particular point in time*, due to execution happening while the database is in protected mode.

To compile the example, enter the *scheduler-java-job* folder and run `mvn package`; the system task job example can be found in the subfolder *stats-system-task-example*.

To deploy the `StatsSystemTask` to eXist, you need to create the collection */db/stats* (the permissions are not important, as each system task will be executed as the `SYSTEM` user), stop eXist, and copy all of the files from *scheduler-java-job/stats-system-task-example/target/stats-system-task-example-1.0-assembly* to *$EXIST_HOME/lib/user*. You can then schedule the `StatsSystemTask` class to run hourly in *$EXIST_HOME/conf.xml* by adding the following job definition to the scheduler configuration before restarting eXist:

```
<job type="system" class="exist.book.example.scheduler.system.StatsSystemTask"
  name="hourly-stats" cron-trigger="0 0 0/1 * * ?">
    <parameter name="stats-collection" value="/db/stats"/>
</job>
```

As an alternative to adding the `StatsSystemTask` to eXist's configuration file, while you cannot schedule immediately using the XQuery `scheduler` extension module, you can instead trigger the system task for almost immediate execution by using the `system:trigger-system-task` function from the XQuery `system` extension module (see `system`):

```
system:trigger-system-task(
    "exist.book.example.scheduler.system.StatsSystemTask",
    <parameters>
        <param name="stats-collection" value="/db/stats"/>
    </parameters>
)
```

While you cannot directly schedule a system task for execution from XQuery, a possible workaround is to create a stub XQuery in the database that simply calls `system:trigger-task`, and then schedule the execution of that XQuery using either `scheduler:schedule-xquery-cron-job` or `schedule-xquery-periodic-job`.

Startup Triggers

Startup triggers are a very simple mechanism that enable you to implement a Java class that has *exclusive* access to the database during the database's startup process. A startup trigger is executed as the final phase, after the database server has initialized but before it is made available for general use.

You may be wondering what you would use a startup trigger for. Typically, they are used for performing computed configuration or adjustments to the database when it is started. To illustrate their use, let's briefly look at the three startup triggers that eXist provides for use during normal database startup:

Autodeployment
The trigger `org.exist.repo.AutoDeploymentTrigger` is used to install any new EXPath packages into the database that have been placed into *$EXIST_HOME/autodeploy*. This ensures that the latest packages are available when the database is started.

Message receiver
For the emerging database replication support in eXist, the trigger `org.exist.replication.jms.subscribe.MessageReceiverStartupTrigger` starts a JMS listener to listen to any incoming replication requests.

RESTXQ
The RESTXQ implementation in eXist uses the trigger `org.exist.extensions.exquery.restxq.impl.RestXqStartupTrigger` to load its resource function registry from the file *$EXIST_HOME/webapp/WEB-INF/data/restxq.registry*. This is done so that the RESTXQ implementation can reregister those resource functions that were previously registered before eXist was restarted. This ensures that resource functions remain available across database restarts. For further details, see "Configuring RESTXQ" on page 216.

Note that startup triggers are executed synchronously, and thus their executing will block the database startup. Depending on your use of eXist, you might want to avoid delaying the database startup any more than necessary. Startup triggers are effectively executed in protected mode, as there is only a single database broker available and it is provided to your startup trigger; hence, no other transactions will be taking place against the database while the triggers are executing. Just as with system tasks (see "Java system task job" on page 444), you need not worry about locking collections or resources.

When creating your own startup trigger, you must implement the `execute` method defined in `org.exist.storage.StartupTrigger`:

```
/**
 * Synchronously execute a task at database startup before the database
 * is made available to connections.
 *
 * Remember, your code within the execute function will block the database
 * startup until it completes!
 *
 * Any RuntimeExceptions thrown will be ignored and database startup
 * will continue. Database startup cannot be aborted by this trigger!
 *
 * Note: If you want an asynchronous trigger, you could use a future in your
 * implementation to start a new thread; however, you cannot access the
 * sysBroker from that thread as it may have been returned to the broker
 * pool. Instead, if you need a broker, you may be able to do something
 * clever by checking the database status and then acquiring a new broker
 * from the broker pool. If you wish to work with the broker pool you must
```

```
     * obtain this before starting your asynchronous execution by calling
     * sysBroker.getBrokerPool().
     *
     * @param sysBroker The single broker available during database startup.
     * @param params A parameter map of keys/values that provide any parameters
     *   given to the startup trigger configuration in $EXIST_HOME/conf.xml.
     */
    public void execute(final DBBroker sysBroker,
      final Map<String, List<? extends Object>> params);
```

Once you have your implementation, you need to place your compiled class onto eXist's classpath; this is typically done by placing a JAR file of your code and any dependent JAR files into *$EXIST_HOME/lib/user*. Remember that you may have to restart eXist for the JVM class loader to see your new JAR files!

Configured Modules Example Startup Trigger

An example class implementing `StartupTrigger` called `exist.book.example.startuptrigger.ConfiguredModulesStartupTrigger` is supplied in the folder *chapters/advanced-topics/startup-trigger* of the *book-code* Git repository (see "Getting the Source Code" on page 15). The example examines all of the available XQuery extension modules written in Java that are configured in eXist (via *$EXIST_HOME/conf.xml*), extracts some details about each module, and stores all of the details into a document in the database at a configured location. The example is rather contrived and may be of little practical use; however, it clearly shows how to implement a startup trigger that both examines eXist's configuration and writes a document into the database, which are useful and common tasks in their own right!

While the code inside a startup trigger executes in a manner akin to protected mode (i.e., there are no other transactions happening at the same time), the task performed by the `ConfiguredModulesStartupTrigger` need not necessarily be executed in this manner. This is because the available configured modules never change while eXist is running; they only change when we adjust eXist's configuration file and restart eXist.

To compile the example, enter the *startup-trigger* folder and run `mvn package`.

To deploy the `ConfiguredModulesStartupTrigger` to eXist, you need to stop eXist and copy all of the files from *startup-trigger/target/configured-modules-startup-trigger-1.0-assembly* to *$EXIST_HOME/lib/user*. You can then configure the `ConfiguredModulesStartupTrigger` class in *$EXIST_HOME/conf.xml* by adding the following job definition to the startup trigger's configuration before restarting eXist:

```
    <trigger
      class="exist.book.example.startuptrigger.ConfiguredModulesStartupTrigger">
        <parameter name="target" value="/db/modules-summary.xml"/>
    </trigger>
```

Database Triggers

Database triggers in eXist are considerably different from the startup triggers we looked at in the previous section, and in many respects are similar to triggers found in relational database systems. Database triggers (or just *triggers*, as we will refer to them going forward) in eXist allow actions to be carried out in response to events *before*, *during*, or *after* various document and collection operations. You can implement triggers that can intercept and either reject, change, or take additional steps when an action is performed on the database.

For example, say you have two collections of documents, and that in one collection you wish to store (and update) documents that meet some criteria set out in documents in the other collection. If the documents do not meet these criteria, they should not be stored or updated. Such cross-document and even cross-collection validation can easily be achieved with a trigger. In this example, you would implement a trigger and configure it on the target collection. Your trigger would perform checks against the criteria collection before a new document was stored or an existing document updated. If those criteria were not met, it could return an error or throw an exception to abort the store or update operation.

Database triggers in eXist offer a huge amount of power to the developer, but remember that the trigger is called once for each operation that it is configured to listen for and is a blocking operation. Any trigger will have an impact on the time it takes to complete the requested database operation, so developers should try to avoid performing lengthy operations in triggers.

So what can you do if you have a lengthy operation (perhaps because it is computationally complex, or you need to talk to several external systems) that you wish to perform as a trigger, but you cannot afford the performance hit to that database operation?

Well, you could instead consider performing your database operations on a *staging* collection, using a scheduled job (see "Scheduled Jobs" on page 435) to carry out the task *asynchronously* in the background, and moving resources to a *live* collection periodically.

While you are most likely interested in creating your own triggers, eXist provides several triggers out of the box. It is useful to mention these, as you may wish to study them as examples:

XML CSV extraction

The trigger `src.org.exist.collections.triggers.CSVExtractingTrigger` offers the facility to split the text node of an element into multiple subelements during document storage. For example, you could transform the following element:

```
<value key="product_model">SomeName|SomeCode12345</value>
```

into:

```
<value key="product_model">
  <product_name>SomeName</product_name>
  <product_code>SomeCode12345</product_code>
</value>
```

The trigger takes two parameters:

`separator`
: The character or string that separates the text values that you wish to split into multiple element text values.

`path`
: An expression similar to a simple XPath expression, and a list of extractions to perform. You may provide as many `path` parameters as you wish.

For example, for you to achieve the preceding example split, the collection configuration for the CSV extracting trigger would look like:

```
<collection xmlns="http://exist-db.org/collection-config/1.0">
  <triggers>
    <trigger class="org.exist.collections.triggers.CSVExtractingTrigger">
      <parameter name="separator" value="|"/>
      <parameter name="path">
        <xpath>/content/properties/value[@key eq "product_model"]</xpath>
        <extract index="0" element-name="product_name"/>
        <extract index="1" element-name="product_code"/>
      </parameter>
    </trigger>
  </triggers>
</collection>
```

History
: The trigger `org.exist.collections.triggers.HistoryTrigger` can be used to create an archive copy of a resource before it is deleted or overwritten. For details, see "Versioning" on page 427.

Replication
: For the emerging database replication support in eXist, the trigger `org.exist.replication.jms.publish.ReplicationTrigger` publishes details of all database operations to a JMS topic.

RESTXQ
: The RESTXQ implementation in eXist uses the trigger `org.exist.extensions.exquery.restxq.impl.RestXqTrigger` to compile XQuery modules after they are stored into the database. It then examines the compiled XQuery for resource functions, and any it finds are registered with the RESTXQ resource

registry so that they may respond to incoming HTTP requests. By default this is configured for the entire database by means of being present in the collection configuration for */db*. For further details, see "Configuring RESTXQ" on page 216.

Streaming Transformations for XML (STX)
: STX is an alternative mechanism to XSLT that was designed specifically for performing transformations on a stream of XML events.

The trigger `org.exist.collections.triggers.STXTransformerTrigger` allows you to transform documents with *STX transformation sheets* when they are stored or updated. For details of the STX transformation sheet language, see the STX specification.

The STX transformer trigger relies on the Joost implementation of STX, so to use STX you need to download Joost 0.9.1 and place the *joost.jar* file into *$EXIST_HOME/lib/user* before restarting eXist. The trigger takes a single parameter, `src`, which points to your STX transformation sheet. This may be either a path to a document in the database, or a URI from which the STX transformation sheet can be downloaded.

For example, the collection configuration for the STX transformer trigger may look like:

```
<collection xmlns="http://;exist-db.org/collection-config/1.0">
  <triggers>
    <trigger class="org.exist.collections.triggers.STXTransformerTrigger">
      <parameter name="src" value="/db/my-transformation.stx"/>
    </trigger>
  </triggers>
</collection>
```

Versioning
: The resource versioning facility in eXist uses the trigger `org.exist.versioning.VersioningTrigger` to support creating *diffs* of resources and tracking the revision history of documents. See "Versioning" on page 427.

XQuery trigger
: The XQuery trigger, while itself written in Java, acts as a bridge to XQuery and is implemented in the `org.exist.collections.triggers.XQueryTrigger` class.

Upon receiving an event, it calls an appropriate function in a configured XQuery. Its purpose is to allow developers to implement their own triggers in XQuery as an easier alternative to Java. It is described in detail in "XQuery Triggers" on page 453.

Triggers in eXist may be implemented in either XQuery or Java. Implementing the triggers in XQuery is much simpler, but there are some operations that can currently only be implemented in Java. You can implement trigger actions *before* or *after* database operations performed for both documents and collections in either XQuery or Java, but you can only modify documents or examine the content of documents *during* store or update operations by using Java triggers.

Triggers that take action on database operations for documents are called *document triggers*; likewise, those that act on collection actions are called *collection triggers*. Triggers that modify documents or examine them as they are being stored are called *document filtering triggers*. Triggers are configured on a per-collection basis in the collection's configuration document (*collection.xconf*), and like all aspects of collection configuration (see "Configuring Indexes" on page 275 for related detail), triggers are inherited or overridden from their parent collection.

As triggers are configured in collection configuration documents, and these are inherited downward or overridden, if you define triggers in a collection configuration document—for example, on the collection */db/a/b*—then any triggers defined in collection configuration documents on */db/a* or */db* will not be inherited by */db/a/b*. If you define your triggers in a collection configuration document, you must also include any other triggers that you wish to use.

For example, the RESTXQ trigger is defined in the collection configuration for */db*. Thus, if you define your own triggers in any other collection configuration documents and wish to use RESTXQ in those collections, you will also need to include a definition for the RESTXQ trigger in your lower-level collection configuration documents.

Whether implementing a trigger in XQuery or Java, you are required to implement a function for each event that you wish to act *before* or *after*. You may only reject changes to the database during *before* actions. Your function, which must follow a naming convention (see Table 16-3 and Table 16-4 for documents and collections, respectively), will be called each time that event occurs within the database.

Table 16-3. Naming convention for document trigger events

Event	XQuery function name	Java function name
Create document	`before-create-document`	`beforeCreateDocument`
	`after-create-document`	`afterCreateDocument`
Update document	`before-update-document`	`beforeUpdateDocument`
	`after-update-document`	`afterUpdateDocument`
Update document metadata	*n/a*	`beforeUpdateDocumentMetadata`
	n/a	`afterUpdateDocumentMetadata`
Copy document	`before-copy-document`	`beforeCopyDocument`

Event	XQuery function name	Java function name
	`after-copy-document`	`afterCopyDocument`
Move document	`before-move-document`	`beforeMoveDocument`
	`after-move-document`	`afterMoveDocument`
Delete document	`before-delete-document`	`beforeDeleteDocument`
	`after-delete-document`	`afterDeleteDocument`

Table 16-4. Naming convention for collection trigger events

Event	XQuery function name	Java function name
Create collection	`before-create-collection`	`beforeCreateCollection`
	`after-create-collection`	`afterCreateCollection`
Copy collection	`before-copy-collection`	`beforeCopyCollection`
	`after-copy-collection`	`afterCopyCollection`
Move collection	`before-move-collection`	`beforeMoveCollection`
	`after-move-collection`	`afterMoveCollection`
Delete collection	`before-delete-collection`	`beforeDeleteCollection`
	`after-delete-collection`	`afterDeleteCollection`

XQuery Triggers

When implementing an XQuery Trigger in eXist, you have two main options:

- Store the XQuery into an XQuery library module in the database and reference it by URI in a parameter to the `XQueryTrigger` called `url` in the collection configuration.
- Write the XQuery code into a parameter to the `XQueryTrigger` called `query` in the collection configuration.

While both options are available to the developer, we would advise taking the first approach and storing your XQuery code for your trigger into the database. This is the approach we will cover in this chapter. This enables you to store your code separately from your configuration, which means that you can reuse this trigger in multiple collection configurations. In addition, it means that you can test your trigger code independently by writing an XQuery that imports your trigger module and calls your functions.

Storing an invalid XQuery trigger library module into the database may cause any collections for which it is configured to reject all database operations. This is because eXist will attempt to execute the query for each database operation, and an exception will be raised if the query is invalid, which will reject the operation.

Consequently, it is often better not to store your XQuery trigger into the same collection as that on which you are configuring the trigger. Otherwise, should you make a mistake in your trigger code, you may be unable to save your updated trigger. This problem occurs when you store an invalid XQuery trigger that is listening to the `before-update-document` event; it will attempt to execute the invalid query when you try to save your fixed trigger code, and as the existing query is invalid, it will reject your update. To resolve this you need to deconfigure the trigger in the appropriate collection configuration document, fix the trigger XQuery code, and then re-enable the trigger in the collection's configuration.

To implement an XQuery trigger you need to simply implement one or more of the functions named in Tables 16-3 and 16-4. Your implementation of each of these functions must currently reside in the namespace `http://exist-db.org/xquery/trigger`.

If you are placing your XQuery trigger in a stored XQuery library module, it is recommended that your module have its own namespace, with the trigger functions just calling your own functions (see Example 16-10). The majority of these functions take a single argument, `$uri` (of type `xs:anyURI`), which provides you with the URI of the document or collection that has caused the trigger event to fire. The exceptions are the functions for copying and moving collection; these instead take two parameters, `$src` and `$dst` (both of type `xs:anyURI`), which describe the source and destination URIs, respectively, of the database operation on the collection. Table 16-5 lists the parameters for the XQuery trigger configuration.

Table 16-5. Parameters for the XQuery trigger configuration

Parameter	Description	Mandatory/optional
`query`	You can directly place your XQuery code in this parameter.	Mandatory, unless using `url`
`url`	You can provide a URI to your XQuery library module in the database that implements one or more functions from the `http://exist-db.org/xquery/trigger` namespace. This approach is preferred over using the `query` parameter.	Mandatory, unless using `query`

Parameter	Description	Mandatory/optional
bindingPrefix	If you wish to pass any parameters as external variables into your XQuery, you need to declare the prefix of the namespace (declared in your XQuery) that they should be bound to.	Mandatory if passing parameters to external variables
anything	You may pass any other parameter to your XQuery and it will be bound to the equivalently named external variable (in the namespace indicated by bindingPrefix), which is declared in your XQuery.	Optional

You will find an example of a simple XQuery trigger implemented in an XQuery library module in the file *chapters/advanced-topics/simple-example-trigger.xqm* of the *book-code* Git repository (see "Getting the Source Code" on page 15), and in Example 16-10. The trigger simply writes an entry to eXist's logfile to record the fact that the trigger function was called when a new document was stored into a collection in the database.

Example 16-10. Simple XQuery trigger implemented in an XQuery library module

```
module namespace et = "http://example/trigger"; ❶

declare namespace trigger = "http://exist-db.org/xquery/trigger"; ❷
import module namespace util = "http://exist-db.org/xquery/util";

declare variable $et:log-level external; ❸

declare function trigger:after-create-document($uri as xs:anyURI) { ❹
    et:log(("XQuery Trigger called after document '", $uri, "' created.")) ❺
};


declare function et:log($msgs as xs:string+) ❻ {
    util:log($et:log-level, $msgs)
};
```

❶ Namespace binding prefix of the XQuery library module.

❷ Import of the XQuery trigger namespace, so that you may implement trigger functions in this namespace.

❸ External variable will be bound to a parameter named `log-level` declared in the trigger configuration within the collection configuration document.

❹ Your implementation of the `after-create-document` function. This must be in the trigger namespace!

❺ Call to your module's functions that provide the actual processing.

❻ Implementation of your business logic.

Example 16-10 shows how to implement a simple XQuery library module that implements the `after-create-document` function that eXist will call when a document is stored into a collection on which the trigger has been configured. The collection configuration for that example might look like:

```
<collection xmlns="http://exist-db.org/collection-config/1.0">
    <triggers>
        <trigger class="org.exist.collections.triggers.XQueryTrigger">
            <parameter name="url"
              value="xmldb:exist:///db/simple-example-trigger.xqm"/> ❶
            <parameter name="bindingPrefix"
              value="et"/> ❷

            <parameter name="log-level" value="INFO"/> ❸
        </trigger>
    </triggers>
</collection>
```

❶ The URI to the XQuery library module in the database that provides the XQuery trigger

❷ The binding prefix, which must be the same as the XQuery library module's namespace prefix

❸ A parameter that is passed into the XQuery library module by binding to the external variable named `$example:log-level`

To install the example:

1. Store the XQuery trigger into the database in a document located at */db/simple-example-trigger.xqm*.
2. Create the collection */db/test-trigger*.
3. Create the collection */db/system/config/db/test-trigger*.
4. Store the collection configuration document (see the preceding code block) into the database in a document located at */db/system/config/db/test-trigger/collection.xconf*. This configures the trigger on the */db/test-trigger* collection.

To test the example:

1. Store *any* document into */db/test-trigger*.
2. Check the logfile *$EXIST_HOME/webapp/WEB-INF/logs/exist.log*. If everything succeeded you should see a message in the log that looks similar to:

```
2014-01-14 11:10:44,561 [eXistThread-30] INFO (LogFunction.java [eval]:150) -
(Line: 14 /db/simple-example-trigger.xqm) XQuery Trigger called after
document '/db/test- trigger/blah3.xml' created.
```

A more complicated example that shows how to react to multiple trigger events and send email notifications is provided in the file *chapters/advanced-topics/journal-notification-trigger.xqm* of the *book-code* Git repository.

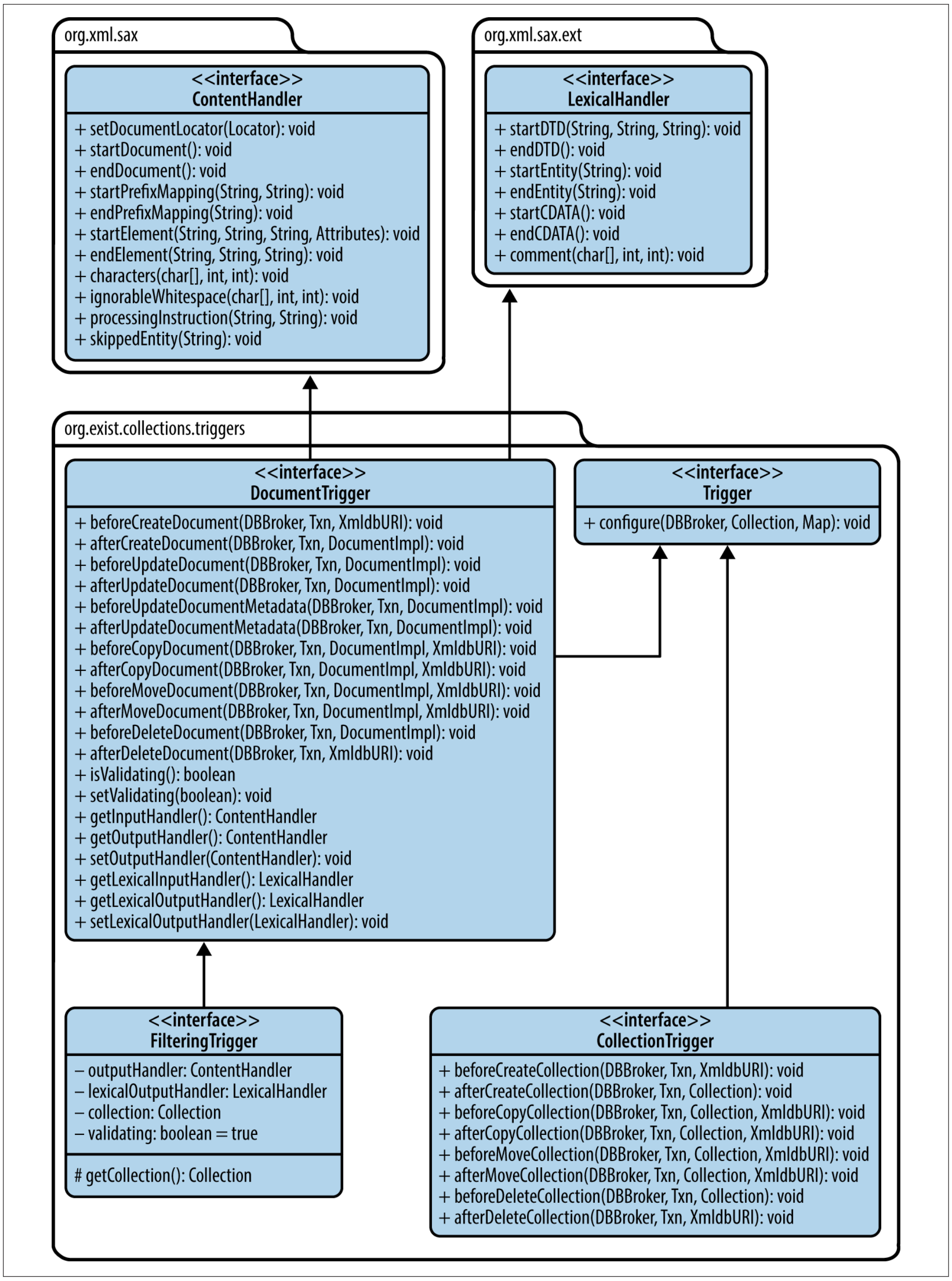
Java Triggers

When implementing database triggers in Java for eXist, you need to implement the appropriate Java interfaces for the events that you wish to handle (see Figure 16-5). The events are split between document events and collection events, as described in Tables 16-3 and 16-4. The interface for document events is `org.exist.collections.triggers.DocumentTrigger`, while the interface for collection events is `org.exist.collections.triggers.CollectionTrigger`. It is perfectly possible to implement both interfaces in a single class if you wish to respond to both types of events.

Omitted from the interfaces `DocumentTrigger` and `CollectionTrigger` are two methods, `prepare` and `finish`. These functions are deprecated and can safely be ignored; in fact, they were removed after the eXist 2.1 release. They provide the old infrastructure for triggers and have been replaced by the `before*` and `after*` functions.

Once you have implemented the appropriate class you need to place your compiled class onto eXist's classpath, which you typically do by placing a JAR file of your code and any dependent JAR files into *$EXIST_HOME/lib/user*. Remember that you may have to restart eXist for the JVM class loader to see your new JAR files!

When writing triggers in Java, you can assume that any `DocumentImpl` or `Collection` objects that you are given are already locked. However, if you open any other documents or collections you must be sure to lock them correctly and, most importantly, to release those locks when you are done with them. Failure to do so could lead to deadlocks in eXist. It is strongly recommended to use the Java `try/catch/finally` pattern to release any acquired locks in the `finally` block.



org.xml.sax
<<interface>>
ContentHandler
+ setDocumentLocator(Locator): void
+ startDocument(): void
+ endDocument(): void
+ startPrefixMapping(String, String): void
+ endPrefixMapping(String): void
+ startElement(String, String, String, Attributes): void
+ endElement(String, String, String): void
+ characters(char[], int, int): void
+ ignorableWhitespace(char[], int, int): void
+ processingInstruction(String, String): void
+ skippedEntity(String): void
org.xml.sax.ext
<<interface>>
LexicalHandler
+ startDTD(String, String, String): void
+ endDTD(): void
+ startEntity(String): void
+ endEntity(String): void
+ startCDATA(): void
+ endCDATA(): void
+ comment(char[], int, int): void
org.exist.collections.triggers
<<interface>>
DocumentTrigger
+ beforeCreateDocument(DBBroker, Txn, XmldbURI): void
+ afterCreateDocument(DBBroker, Txn, DocumentImpl): void
+ beforeUpdateDocument(DBBroker, Txn, DocumentImpl): void
+ afterUpdateDocument(DBBroker, Txn, DocumentImpl): void
+ beforeUpdateDocumentMetadata(DBBroker, Txn, DocumentImpl): void
+ afterUpdateDocumentMetadata(DBBroker, Txn, DocumentImpl): void
+ beforeCopyDocument(DBBroker, Txn, DocumentImpl, XmldbURI): void
+ afterCopyDocument(DBBroker, Txn, DocumentImpl, XmldbURI): void
+ beforeMoveDocument(DBBroker, Txn, DocumentImpl, XmldbURI): void
+ afterMoveDocument(DBBroker, Txn, DocumentImpl, XmldbURI): void
+ beforeDeleteDocument(DBBroker, Txn, DocumentImpl): void
+ afterDeleteDocument(DBBroker, Txn, XmldbURI): void
+ isValidating(): boolean
+ setValidating(boolean): void
+ getInputHandler(): ContentHandler
+ getOutputHandler(): ContentHandler
+ setOutputHandler(ContentHandler): void
+ getLexicalInputHandler(): LexicalHandler
+ getLexicalOutputHandler(): LexicalHandler
+ setLexicalOutputHandler(LexicalHandler): void
<<interface>>
Trigger
+ configure(DBBroker, Collection, Map): void
<<interface>>
FilteringTrigger
– outputHandler: ContentHandler
– lexicalOutputHandler: LexicalHandler
– collection: Collection
– validating: boolean = true
# getCollection(): Collection
<<interface>>
CollectionTrigger
+ beforeCreateCollection(DBBroker, Txn, XmldbURI): void
+ afterCreateCollection(DBBroker, Txn, Collection): void
+ beforeCopyCollection(DBBroker, Txn, Collection, XmldbURI): void
+ afterCopyCollection(DBBroker, Txn, Collection, XmldbURI): void
+ beforeMoveCollection(DBBroker, Txn, Collection, XmldbURI): void
+ afterMoveCollection(DBBroker, Txn, Collection, XmldbURI): void
+ beforeDeleteCollection(DBBroker, Txn, Collection): void
+ afterDeleteCollection(DBBroker, Txn, XmldbURI): void

Figure 16-5. UML diagram showing Java trigger classes

An instance of your trigger class will be lazily instantiated for each collection it is configured for. When your trigger class is instantiated, your implementation of the configure method from org.exist.collections.triggers.Trigger will be called; you may use this function to read any parameters from the trigger's configuration and set up any initial state in a thread-safe manner. Should you decide to store some state in member variables of your class, remember that the class instance is per collection, so the values of these variables will not be globally consistent.

Also be aware that the calls to the trigger methods (e.g., beforeStoreDocument) are not thread-safe. This means that more than one thread may be in any, or even all, of the event functions defined in your trigger class! If you wish to keep state, it is up to you to manage concurrent access to that state appropriately.

As they are simpler to implement than document triggers, we will look first at collection triggers, which will show you how to implement *event* functions. We will then look at document triggers, which have similar event functions and a whole lot more!

Java collection triggers

When you're implementing Java collection triggers your class must provide implementations for all of the methods defined in org.exist.collections.triggers.Trigger and org.exist.collections.triggers.CollectionTrigger to compile. However, triggers were designed in such a way that you really need only fill out those methods that you wish to act upon.

The simplest collection trigger would only provide code for a single method, as shown in Example 16-11, where we log the username of a user creating a collection.

Example 16-11. Simplest collection trigger

```
package example;

import org.apache.log4j.Logger;
import org.exist.collections.triggers.CollectionTrigger;
import org.exist.collections.triggers.TriggerException;
import org.exist.storage.DBBroker;
import org.exist.storage.txn.Txn;
import org.exist.xmldb.XmldbURI;
import java.util.List;
import java.util.Map;

public class SimplestCollectionTrigger implements CollectionTrigger { ❶
    private final static Logger LOG =
        Logger.getLogger(SimplestCollectionTrigger.class);

    @Override
    public void beforeCreateCollection(DBBroker broker, Txn txn, XmldbURI uri) ❷
      throws TriggerException {
        LOG.info("User '" + broker.getSubject().getName() + ❸
          "' is creating the Collection '" + uri + "'...");
    }

    // Omitted: other empty function implementations here...
    ❹
}
```

❶ We must implement the `CollectionTrigger` interface.

❷ We provide an implementation of the event method `beforeCreateCollection`.

❸ Before the collection indicated by `uri` is created, we record the event in eXist's logfile, attributing it to a specific user.

❹ We must provide implementations of all the other functions from `CollectionTrigger` here for the code to compile; however, as we are not using them and they are easily generated by an IDE, we have omitted them for brevity.

Just like in previous examples, to have the trigger fired on the database events that you are interested in, you must add it to the collection configuration documents for those collections that you wish your trigger to act upon. An example configuration for the `SimplestCollectionTrigger` would look like:

```
<collection xmlns="http://exist-db.org/collection-config/1.0">
    <triggers>
        <trigger class="example.SimplestCollectionTrigger"/>
    </triggers>
</collection>
```

Remember that you must configure your collection triggers on the *parent* of the collection for those collection events that you wish them to be triggered upon. For example, if you want to be made aware when a new collection is created in */db/myapp/data*, then you must add your trigger to the collection configuration document for the collection */db/myapp/data*. In this way, if a user were to create, copy, move, or delete the collection */db/myapp/data/some-collection*, your trigger would receive the event. Likewise, it would receive the events for any descendant collections—for example, */db/myapp/data/some-collection/subcollection*—as collection configuration is inherited!

No delete example collection trigger

An example class implementing `CollectionTrigger` called `exist.book.example.trigger.collection.NoDeleteCollectionTrigger` is supplied in the folder *chapters/advanced-topics/java-database-trigger/nodelete-collection-trigger-example* of the *book-code* Git repository (see "Getting the Source Code" on page 15).

The example is designed to show how database operations can be aborted by Java triggers. The trigger takes a blacklist of collection URIs as a parameter and prevents those collections from being deleted and optionally from being moved (indicated by another parameter). The trigger is able to prevent these collections from being deleted or moved by throwing a `TriggerException` during the *before* phase of the delete or move events, which causes eXist to abort the operation and report the exception.

To compile the example, enter the *java-database-trigger* folder and run `mvn package`.

To deploy the `NoDeleteCollectionTrigger` to eXist, you need to:

1. Compile the code as described previously, and then copy all of the files from *java-database-trigger/nodelete-collection-trigger-example/target/nodelete-collection-trigger-example-1.0-assembly* to *$EXIST_HOME/lib/user*.
2. Restart eXist so that it picks up the new JAR files.
3. Create the collections */db/data*, */db/data/private*, and */db/data/private/subcollection*.
4. Create the configuration collection */db/system/config/db/data*.
5. Configure the trigger in a collection configuration document in the database, which you should locate at */db/system/config/db/data/collection.xconf*:

```
<collection xmlns="http://exist-db.org/collection-config/1.0">
  <triggers>
    <trigger
  class="exist.book.example.trigger.collection.NoDeleteCollectionTrigger">

      <!-- whether to also prevent collection moves
        for your blacklisted collections -->
      <parameter name="treatMoveAsDelete" value="true"/>

      <!-- your blacklist: -->
      <parameter name="blacklist" value="/db/data/super-secret"/>
      <parameter name="blacklist" value="/db/data/private"/>
    </trigger>
  </triggers>
</collection>
```

To test the example:

1. Attempt to delete or move the collection */db/data/private/subcollection*, */db/data/private*, or */db/data/super-secret*; you should find that it is now impossible.

2. Check the logfile *$EXIST_HOME/webapp/WEB-INF/logs/exist.log*. You should see the NoDeleteCollectionTrigger log messages that look similar to:

   ```
   2014-01-16 14:18:52,918 [eXistThread-32] INFO
   (NoDeleteCollectionTrigger.java [beforeDeleteCollection]:97) -
   Preventing deletion of blacklisted collection '/db/ data/private'.
   ```

Java document triggers

While our discussion in "Java collection triggers" on page 459 focused on handling the *before* and *after* events for various database operations on collections, here we'll look at handling events *during* the storage of XML documents. This allows us to modify a document dynamically as it is being stored. Of course, document triggers still have all of the before and after events that you would expect for database operations on documents, but implementing them is similar enough to implementing collection triggers that we need not discuss them further here. Document triggers also give you the ability to perform streaming validation and transformations on documents.

When you're implementing Java document triggers, your class must provide implementations for all of the methods defined in `org.exist.collections.triggers.Trigger` and `org.exist.collections.triggers.DocumentTrigger` to compile.

The `DocumentTrigger` interface mainly varies from `CollectionTrigger` in that it also acts as a SAX (Simple API for XML) event handler by extending `org.xml.sax.ContentHandler` and `org.xml.sax.ext.LexicalHandler`. With SAX events, your trigger effectively sits in the middle of a pipeline, receiving SAX events from the parser (which is reading the document to store) and sending them on to the database for validation or storage (see Figure 16-6). If you choose to discard some of these events or generate new events, you are effectively modifying the incoming document for validation or storage.

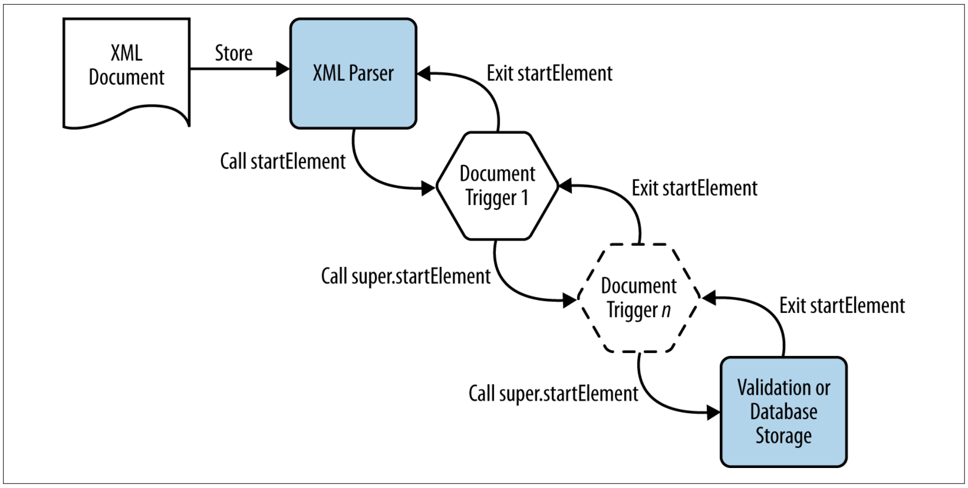


Figure 16-6. Document trigger SAX pipeline

Having to implement `ContentHandler` and `LexicalHandler` brings some complexity to document triggers, so for convenience eXist offers the abstract class `org.exist.collections.triggers.FilteringTrigger` to reduce this. It is recommended that you always extend `FilteringTrigger` and never directly implement `DocumentTrigger` in your own triggers. `FilteringTrigger` provides default implementations of both `ContentHandler` and `LexicalHandler` by simply forwarding the SAX events to either validation or storage. If you are only interested in working with the *before* and *after* document events, by extending `FilteringTrigger` you need not ever worry about the SAX events. Conversely, if you are interested in the SAX events for the purpose of modifying the document, additional validation, or some other reason, by extending `FilteringTrigger` you can just override those SAX methods of interest, while the remainder will be handled correctly.

If you choose to override the SAX event methods in `FilteringTrigger` and you still want the event to be passed on to the database for validation or storage, then you must remember to call the equivalent method on the `super` class. If you do not call the method on the `super` class, your trigger is actually discarding those events and they will never reach the database!

When dealing with SAX events in a document trigger you must recognize that the trigger is called *twice*, once during each of eXist's two phases of storing a document:

Validation
: As the entire document is being parsed, the generated SAX events are sent to your trigger, which is responsible for swallowing or forwarding them (with or

without modifications). When the events are forwarded, they are sent to the *validator*, which ensures the resultant document is well formed and valid according to any configured schemas. If you choose to throw a `SAXException` (or any `RuntimeException`) from one of your SAX event handler methods, then you are effectively indicating to eXist that you consider the document to be invalid, which stops the validation phase and aborts the store process.

Storage
: If the validation phase completes, eXist then enters the storage phase. The entire document is parsed for a second time, and the generated SAX events are again sent to your trigger, which is responsible for forwarding them (with or without modifications). When the events are forwarded, they are sent to the database storage engine, which is responsible for writing the document into the database. If you choose to throw a `SAXException` (or any `RuntimeException`) from one of your SAX event handler methods, you are signaling to eXist that there was a problem with the store process, and eXist will abort the store and roll back the transaction.

You can determine which phase your SAX event handler methods are being called in by calling the function `isValidating` from the `super FilteringTrigger` class. An interesting effect of having different validation and storage phases is that you can modify the document stream in a different manner in each phase. This offers some interesting possibilities, such as allowing the validation phase to pass, and then rewriting the document into another form before it is stored.

Consider the `startElement` method from a fictional implementation of `FilteringTrigger` shown in Example 16-12. This example shows you how you can *drop* elements, *rename* elements, and *create* new elements as the document is being stored.

Example 16-12. Event handling in a filtertrigger

```
@Override
public void startElement(final String namespaceURI, final String localName,
  final String qname, final Attributes attributes) throws SAXException {

    if(localName.equals("author")) {
       //drop an element ❶

    } else if(localName.equals("color")) {
       //rename an element
       super.startElement(namespaceURI, "colour", ❷
         qname.replace("color", "colour"), attributes);

    } else if(localName.equals("isbn")) {
       //encapsulate an element
       super.startElement(namespaceURI, "reference", "reference", null); ❸
       super.startElement(namespaceURI, localName, qname, attributes); ❹
    } else {
        //keep other elements
        super.startElement(namespaceURI, localName, qname, attributes); ❺
    }
}
```

❶ We drop any element that is named `author`. We achieve the drop simply by not calling `super.startElement`.

❷ We rename any element that is named `color` to `colour`. We do so by calling `super.startElement` but *replacing* the element name.

❸❹ We encapsulate any element that is named `isbn` (and its following siblings) inside an element named `reference`. We do this by calling `super.startElement` to start a new element, then calling `super.startElement` for the current element.

❺ We keep any other element by simply calling `super.startElement` for the current element.

For the trigger to actually work, though, we must also have a matching `endElement` method that *balances* the start and end of elements; otherwise, we will end up with a document that is not *well formed*. Such a matching `endElement` method implementation would look like:

```
@Override
public void endElement(final String namespaceURI, final String localName,
  final String qname) throws SAXException {

    if(localName.equals("author")) {
        //drop an element

    } else if(localName.equals("color")) {
        //rename an element
        super.endElement(namespaceURI, "colour",
          qname.replace("color", "colour"));

    } else if(localName.equals("isbn")) {
        //encapsulate an element
        ❶super.endElement(namespaceURI, localName, qname);
        ❷super.endElement(namespaceURI, "reference", "reference");

    } else {
        //keep other elements
        super.endElement(namespaceURI, localName, qname);
    }
}
```

❶❷ Note that when you encapsulate an element in another, the order of the generated events is reversed in the `endElement` method as compared to the `startElement` method. In other words, the `isbn` element has to be ended *before* the `reference` element, as the `isbn` element was started *after* the `reference` element.

Document triggers are incredibly powerful, and here we have barely scratched the surface of what is possible. However, to further assist you we have included a more complete example with this book.

Example filtering trigger

An example class extending `FilteringTrigger` called `exist.book.example.trigger.document.ExampleFilteringTrigger` is supplied in the folder *chapters/advanced-topics/java-database-trigger/filtering-trigger-example* of the *book-code* Git repository (see "Getting the Source Code" on page 15).

The example is designed to show how you can build a path to the current element even though you are processing a stream, and you can then use this path to make decisions about whether to remove an element. The example also shows how to pass in a significantly more complicated set of configuration parameters to the trigger from the collection configuration document. These parameters are then used for keeping a map of elements that should be renamed.

To compile the example, enter the *java-database-trigger* folder and run `mvn package`.

To deploy the `ExampleFilteringTrigger` to eXist, you need to:

1. Compile the code as described previously, and then copy all of the files from *java-database-trigger/filtering-trigger-example/target/filtering-trigger-example-1.0-assembly* to *$EXIST_HOME/lib/user*.
2. Restart eXist so that it picks up the new JAR files.
3. Create the collection */db/test-data*.
4. Create the configuration collection */db/system/config/db/test-data*.
5. Configure the trigger in a collection configuration document in the database, which you should locate at */db/system/config/db/test-data/collection.xconf*:

```
<collection xmlns="http://exist-db.org/collection-config/1.0">
    <triggers>
        <trigger
  class="exist.book.example.trigger.document.ExampleFilteringTrigger">

            <!-- paths to elements that should be dropped -->
            <parameter name="drop" value="/a/b/c"/>

            <!-- map of elements that should be renamed -->
```

```
                <parameter name="elements">
                    <rename from="d" to="e"/>
                    <rename from="e" to="d"/>
                </parameter>
            </trigger>
        </triggers>
    </collection>
```

To test the example:

1. Store the following document into */db/test-data*:

    ```
    <a>
        <b>
            <c>should be removed</c>
            <c>should also be removed</c>
        </b>
        <c>should not be removed</c>
        <d>should be renamed to e</d>
        <e>should be renamed to d</e>
    </a>
    ```

2. Open the actual document stored into */db/test-data*. You should see that it instead contains something like the following:

    ```
    <a>
        <b/>
        <c>should not be removed</c>
        <e>should be renamed to e</e>
        <d>should be renamed to d</d>
    </a>
    ```

Internal XQuery Library Modules

As you know by now, XQuery modules come in two varieties, *main modules* and *library modules*. Main modules can be directly invoked, and execution begins at the *query body*. Main modules may import other library modules, but a complete XQuery may contain only a single main module. Library modules reside in a specific namespace and contain function and variable declarations grouped by that namespace. Library modules do not have a query body, and thus there is no way to directly execute a library module, but frameworks like RESTXQ (see "Building Applications with RESTXQ" on page 215) and eXist's SOAP Server (see "SOAP Server" on page 362) are able to map HTTP requests onto specific library module function invocations.

eXist provides two types of library modules:

External
: These are library modules written in XQuery. They follow the W3C XQuery specification for library modules and allow users to easily write modules in XQuery. For further information, see *http://www.w3.org/TR/xquery/#dt-library-module*.

Internal
: These are library modules written in Java. In these modules, the internals of XQuery functions and variables are written in eXist's host programming language (Java) but are callable from XQuery as though they were any other XQuery function or variable. This arguably follows the W3C XQuery specification for library modules, but deviates slightly as eXist does not require you to explicitly declare the functions as *external functions* in XQuery because it is able to perform the required static type analysis regardless. For further information, see *http://www.w3.org/TR/xquery/#dt-external-function*.

In this chapter we do not look at external modules, as they are not specific to eXist and there is already a great wealth of material on them, both in this book and other XQuery learning resources. Instead, we focus here on internal modules and how you can easily build your own using Java.

So perhaps first we should ask: why would we write a library module in Java as opposed to XQuery?

There is really only one valid reason to consider:

It cannot be done in XQuery!
: You'll need to turn to Java when it is impossible to solve your problem by putting other XQuery functions together. You most likely want to introduce a new and unique function. For example, W3C XQuery 1.0 has no functional capability for sending email, so you may wish to create a function that allows this (in fact, such an extension function is already included in eXist, as covered in `mail`).

Conversely, there are many reasons why you should *not* write a library module in Java as opposed to XQuery. Here are a few of the important ones:

Not understanding XQuery
: It may be tempting to implement something in Java because you have not had as much experience with XQuery. Generally speaking, this is a bad plan, as you will be calling this Java from XQuery regardless—your time would most likely be better invested in learning more about XQuery. XQuery is really the thing that makes eXist so powerful, so you would be well advised to get to grips with it.

Performance
: When you write an XQuery function to implement a specific piece of business logic or use case, it may call many other XQuery functions. Often people misun-

derstand how XQuery is executed in eXist and fear that this long function call chain is affecting performance, so they produce a single function in Java that can be called from XQuery, which does it all. In fact, eXist compiles all XQuery code down to Java function calls and caches the compiled form. It is much more likely that performance issues are caused by misconfigured indexes or collections. Even if you do find that some eXist XQuery functions are slow, it would be better to work on optimizing those so that all XQuery code benefits!

Deadlocks, memory leaks, and death
If you wish to interact with eXist from your own code, doing so from within Java is much harder than from within XQuery. From within Java, your internal module code is running directly inside eXist, so to talk to eXist you need to use its internal APIs. While this is absolutely possible, you must take great care to lock and unlock resources appropriately, and to free up any memory that you allocate. Failure to do so can quickly lock up the database and potentially crash eXist, and if you mismanage resources and transactions you may even corrupt your database!

If you still wish to implement an internal module, your implementation needs to implement the interface `org.exist.xquery.InternalModule`. As a convenience, eXist provides the abstract class `org.exist.xquery.AbstractModule` as a starting point; this greatly eases implementation.

XQuery is a functional programming language, so its functions—including those that you implement in your internal modules—should really not cause side effects. However, sometimes with XQuery you have to allow side effects to be able to achieve the desired outcome. For example, the eXist `xmldb` module's side effects allow you to change the state of the database. If you can avoid it, it is good practice to write your functions as transformations from their input to their output, without causing side effects!

Implementing a library module typically involves implementing at least two classes. The first, the module class, contains information about all the functions that the module provides. The others (one or more) are classes for each function that the module provides (although functions may also be grouped into classes if desired). Perhaps the easiest way to explain this is for us to dive straight in at the deep end and look at some code for our first internal module, a very simple "Hello World" module that provides a single function to XQuery (see Example 16-13).

Example 16-13. HelloWorld Java XQuery module

```
public class HelloModule extends AbstractInternalModule { ❶
```

```
    protected final static String NS = "http://hello"; ❷
    protected final static String NS_PREFIX = "h"; ❸

    private final static FunctionDef functions[] = { ❹
      new FunctionDef(HelloFunctions.FNS_HELLO_WORLD, HelloFunctions.class) ❺
    };

    public HelloModule(Map<String, List<? extends Object>> parameters) {
        super(functions, parameters); ❻
    }

    @Override
    public String getNamespaceURI() {
        return NS;
    }

    @Override
    public String getDefaultPrefix() {
        return NS_PREFIX;
    }

    @Override
    public String getDescription() {
        return "Simple Hello World module";
    }

    @Override
    public String getReleaseVersion() {
        return "2.1"; ❼
    }
}
```

❶ Our class implements `InternalModule` by extending `AbstractInternalModule`.

❷❸ We define a namespace and namespace prefix for our module.

❹ We define the functions that will form a part of this module.

❺ We reference a single function, which will be a function of this module. This shows how modules and their functions are linked together: basically, the module has a static array of references to the functions that it provides, and it declares these by calling the constructor of the `super` class.

❻ We call the constructor of the `super` class, passing in the array of functions that make up this module.

7. We have to return a string that describes which version of eXist this module became available in. This is just for documentation purposes and is not further processed.

To implement the actual "Hello World" function, which will be named `hello-world`, we need to extend the abstract class class `org.exist.xquery.Function`. To assist in this, eXist provides the abstract subclass `org.exist.xquery.BasicFunction`, which makes life much easier by dealing with the necessary mechanics for the *XQuery profiler* and extracting argument values that are passed to our function from the *XQuery context*. Almost all of the internal module functions already implemented in eXist extend `BasicFunction`, and we would recommend that you do the same unless you need more control over processing (which is unlikely in most use cases). So now that we have seen the preceding internal module implementation, which tells eXist about our `hello-world` function, let's see how we actually implement the function in Example 16-14.

Example 16-14. HelloWorld Java XQuery function

```
public class HelloFunctions extends BasicFunction { (1)

    private final static QName qnHelloWorld = (2)
      new QName("hello-world", HelloModule.NS, HelloModule.NS_PREFIX);

    //signature of our XQuery h:hello-world() function
    public final static FunctionSignature FNS_HELLO_WORLD = (3)
      new FunctionSignature(
        qnHelloWorld (4),
        "Say \"hello world\"!" (5),
        null (6),
        new FunctionReturnSequenceType(
          Type.DOCUMENT, Cardinality.ONE, "The hello!"
          ) (7)
    );

    //standard constructor, which allows multiple functions to be
    //implemented in one class
    public HelloFunctions(final XQueryContext context,
      final FunctionSignature signature) {
      super(context, signature);
    }

    //called when the xquery function is executed
    @Override
    public Sequence eval(final Sequence[] args, (8)
      final Sequence contextSequence) throws XPathException {

        final Sequence result;
        //act on the invoked function name
        if(isCalledAs(qnHelloWorld.getLocalName())) { ❾
            result = sayHelloWorld(); ❿
        } else {
            throw new XPathException("Unknown function call: " ⓫
              + this.getName().toString());
        }

        return result; ⓬
    }

    //Constructs the in-memory XML document:
    //* <hello>world</hello>
    //@return The in-memory XML document
    private Sequence sayHelloWorld() { ⓭
        final MemTreeBuilder builder = new MemTreeBuilder(); ⓮
        builder.startDocument();
        builder.startElement(new QName("hello", HelloModule.NS, ⓯
          HelloModule.NS_PREFIX), null);
          builder.characters("world");
        builder.endElement();
        builder.endDocument();

        return builder.getDocument(); ⓰
    }
}
```

❶ Our class implements `org.exist.xquery.Function` by extending `BasicFunction`.

❷ We define the name of our XQuery function to be `hello-world`. You should always define this within the namespace of the internal module.

> The standard way of naming functions and variables in XQuery is to use all lowercase letters and separate terms with a hyphen.

❸ Here we define the function signature of our XQuery function, which includes the name, description (for documentation), any expected parameters, and the return type. Our function will have the signature `h:hello-world() as xs:string`.

❹ The function signature includes the name of our function. We define the name as a separate variable for the purposes or referencing it later as a constant—for example, within the `eval` function.

❺ The textual description of our function.

❻ Any expected parameters for our function. Our `hello-world` function does not take any parameters, so we can use `null` here.

❼ The return type and cardinality of our function. Our `hello-world` function will return a single XML document node.

❽ Any parameters that our functions expect will be passed into the `eval` function as an array of `Sequence` objects.

❾ As it is possible to encode more than one XQuery function in a single Java class that extends `BasicFunction`, we switch on the name of the function that was called from XQuery.

❿ We call our business logic, which generates the "Hello World" XML document.

⓫ If we do not recognize which function was called, we throw an `org.exist.xquery.XPathException`. This is really just for completeness and should not ever be invoked, as eXist should not route unexpected XQuery function calls to us.

⓬ We return the results of our processing to the XQuery.

⓭ This is our isolated business logic, which will generate our "Hello World" XML.

⓮ We use eXist's `MemTreeBuilder` to construct an XML document dynamically in memory.

⓯ We define the namespace of the XML document that we are producing.

If you're constructing a custom XML document and there is otherwise no defined namespace to use, it is considered good practice to place the nodes of the document into the namespace of the module as opposed to the default namespace.

⓰ We return the XML document node of our constructed document.

Using the Hello World Module

As mentioned previously, the Java source code implementing the internal module called `exist.book.example.module.internal.HelloModule` is supplied in the folder *chapters/advanced-topics/internal-module/hello-world-module-example* of the *book-code* Git repository (see "Getting the Source Code" on page 15). The example covers both the simple `hello-world` function just discussed and a more complex `say-hello` function, which is described next. It is designed to show how relatively little bespoke code is required to implement a simple internal module.

To compile the example, enter the *internal-module* folder and run `mvn package`.

To deploy the `HelloModule` to eXist, you need to:

1. Compile the code as previously described, and then copy all of the files from *internal-module/hello-world-module-example/target/hello-world-module-example-1.0-assembly* to *$EXIST_HOME/lib/user*.
2. Add the following module definition to *$EXIST_HOME/conf.xml*, in the `xquery/builtin-modules` section:

   ```
   <module uri="http://hello"
     class="exist.book.example.module.internal.HelloModule"/>
   ```
3. Restart eXist so that it picks up the new JAR files.

To test the example:

1. Execute the following XQuery from either eXist's Java Admin Client (see "Java Admin Client" on page 373) or eXide (see "eXide" on page 374):

   ```
   xquery version "1.0";

   declare namespace h = "http://hello";

   h:hello-world()
   ```
2. You should see a result that looks similar to:

   ```
   <h:hello xmlns:h="http://hello">world</h:hello>
   ```

While this example shows you how to write an extension function for XQuery in Java, it is very basic. To expand on this, we will look next at eXist's Java model of the XDM types used in XQuery, and then study an example where we create a function that takes several parameters using these types and acts upon them.

Types and Cardinality

Where functions take arguments and return values as a result of their computation (and they should in the functional world), these arguments and return values have both a *type* and a *cardinality*. The type defines the variety of data that can be held (for example, a string or number), and the cardinality defines how many values of that type may be present. Types and cardinalities are defined in the W3C XQuery 1.0 and

XPath 2.0 Data Model (XDM) specification. If you are an experienced XQuery developer, you're most likely already familiar with this document; if not, as an implementer of an internal module you should have at least a basic understanding of these subjects. A very useful summary diagram of the type hierarchy in XQuery is available in the specification.

When implementing an internal module, you will be working with the XDM types in Java as opposed to XQuery. eXist has a Java class to model each of those XDM types. An understanding of how to map from an XDM type as used in XQuery to eXist's Java type is essential to enable you to create functions that accept parameters and return values. All of eXist's XDM Java types for atomic types are in the package `org.exist.xquery.value`. The type mappings are listed in Table 16-6.

Table 16-6. XDM atomic value type mappings

XDM atomic value type	eXist's Java class	Notes
`item`	`Item`	An interface.
`xs:anyAtomicType`	`AtomicValue`	An abstract class.
`xs:untypedAtomic`	`UntypedAtomicValue`	Internally represented using `java.lang.String`.
`xs:anyURI`	`AnyURIValue`	Internally represented using `java.lang.String`. Provides utility methods for converting to/from `org.exist.xmldb.XmldbURI`.
`xs:base64Binary`	`BinaryValue`	Internally represented using `java.io.InputStream` and `java.io.OutputStream`. Actual encoding/decoding is lazy, and uses either `Base64BinaryValueType` or `HexBinaryType`, respectively.
`xs:hexBinary`		
`xs:boolean`	`BooleanValue`	Internally represented using `boolean`.
`xs:dateTime`	`DateTimeValue`	Internally represented using `javax.xml.datatype.XMLGregorianCalendar`.
`xs:date`	`DateValue`	
`xs:time`	`TimeValue`	
`xs:gDay`	`GDayValue`	
`xs:gMonth`	`GMonthValue`	
`xs:gMonthDay`	`GMonthDay`	
`xs:gYear`	`GYearValue`	
`xs:gYearMonth`	`GYearMonthValue`	
`xs:duration`	`DurationValue`	Internally represented using `javax.xml.datatype.Duration`.
`xs:dayTimeDuration`	`DayTimeDurationValue`	

XDM atomic value type	eXist's Java class	Notes
`xs:yearMonthDuration`	`YearMonthDuration Value`	
`xs:string`	`StringValue`	Internally represented using a composition of `java.lang.String`, `int`, and `boolean`.
`xs:normalizedString`		
`xs:language`		
`xs:NMTOKEN`		
`xs:Name`		
`xs:NCName`		
`xs:ID`		
`xs:IDREF`		
`xs:ENTITY`		
`xs:QName`	`QNameValue`	Internally represented using `org.exist.dom.QName`.
`xs:float`	`FloatValue`	Internally represented using `float`.
`xs:double`	`DoubleValue`	Internally represented using `double`.
`xs:decimal`	`DecimalValue`	Internally represented using `java.math.BigDecimal`.
`xs:integer`	`IntegerValue`	Internally represented using a composition of `java.lang.BigInteger` and `int`.
`xs:nonPositiveInteger`		
`xs:negativeInteger`		
`xs:long`		
`xs:int`		
`xs:short`		
`xs:byte`		
`xs:nonNegativeInteger`		
`xs:unsignedLong`		
`xs:unsignedInt`		
`xs:unsignedShort`		
`xs:unsignedByte`		
`xs:positiveInteger`		

eXist has two Java implementations of each XDM *node type*. One is an in-memory implementation, called Memtree, that solely retains the nodes in memory and is useful for computed node construction. The other is a persistent Document Object Model (DOM) implementation that represents nodes that are stored in the database.

The classes of each have mostly the same names but are maintained in different packages. The Java classes for the in-memory implementation of XDM node types are in the package `org.exist.memtree`, while the persistent DOM implementations are in the package `org.exist.dom`. See Table 16-7.

Table 16-7. XDM node type mappings

XDM node type	eXist's Java class
`node`	`NodeImpl`
`attribute`	`AttrImpl` (DOM) / `AttributeImpl` (memtree)
`comment`	`CommentImpl`
`document`	`DocumentImpl`
`element`	`ElementImpl`
`processing-instruction`	`ProcessingInstructionImpl`
`text`	`TextImpl`

While all function parameters in eXist are *sequences*, the cardinality of these parameters is constrained in the definition of the function signature. eXist provides cardinality constants that model the occurrence indicators used for function parameters in the XQuery specification. These cardinality constants are defined in the class `org.exist.xquery.Cardinality` (see Table 16-8).

Table 16-8. XQuery occurrence mappings

XQuery occurrence indicator	eXist's cardinality constant
	`Cardinality.EXACTLY_ONE` (when explicitly typed)/`Cardinality.ZERO_OR_MORE` (when not explicitly typed)
?	`Cardinality.ZERO_OR_ONE`
*	`Cardinality.ZERO_OR_MORE`
+	`Cardinality.ONE_OR_MORE`

Function Parameters and Return Types

Now that we have an understanding of how XDM types are implemented by eXist in Java, we can consider how we might use these to accept parameters to our functions or return certain result types. When you're implementing a function for an internal module, it helps to think of the function as a transformation from an *array of sequences* to a *sequence*. For example, the `eval` function of your `BasicFunction` will be passed a Java array of `org.exist.xquery.value.Sequence` objects and must either throw an `org.exist.xquery.XPathException` or return a `Sequence`. Sequences are also described in the XDM specification, and can basically be thought of as collections of zero or more atomic values and/or nodes. Each item in the Java array of Sequence objects represents an individual parameter that was passed to your XQuery function; even though these are Sequence objects, you will have already declared the type and cardinality of the parameters for your function in its FunctionSignature for the internal module.

Now let's look at defining a function signature for a function that allows one person to say hello to many other people. This function potentially needs to know:

- The name of the person who is saying hello.
- The names of the people she is saying hello to.
- We could also optionally allow the greeting to be customized so that instead of saying "Hello," she could say "Bonjour" or use any other desired form of greeting.

If we imagine the function signature for such an XQuery function, it might look something like:

```
h:say-hello($greeter as xs:string, $greeting as xs:string?,
  $visitors as xs:string+) as xs:string+
```

Such a function could be called like so:

```
xquery version "1.0";

declare namespace h = "http://hello";

h:say-hello("adam", "Hi" ("Erik", "Simon"))
```

and we might expect to see a result similar to:

```
("Adam says Hi to Erik", "Adam says Hi to Simon")
```

Now that we know what we want our XQuery function to do and we know what the signature should look like, we need to implement this in our internal module just as we did before (in Example 16-14) by defining another FunctionSignature:

```
private final static QName qnSayHello =
  new QName("say-hello", HelloModule.NS, HelloModule.NS_PREFIX);

public final static FunctionSignature FNS_SAY_HELLO = new FunctionSignature(
    qnSayHello,
    "Say \"hello world\"!",
    new SequenceType[] { ❶
        new FunctionParameterSequenceType("greeter",
          Type.STRING,
          Cardinality.EXACTLY_ONE,
          "The greeter, i.e. the name of the person that is saying 'hello'."
          ),
        new FunctionParameterSequenceType("greeting",
          Type.STRING,
          Cardinality.ZERO_OR_ONE,
          "An optional greeting, if omitted then 'hello' is used."
          ),
        new FunctionParameterSequenceType("visitors",
          Type.STRING,
          Cardinality.ONE_OR_MORE,
          "The visitors, i.e. the names of the people that the greeter is "
          + "saying 'hello' to."
          ),
    },
    new FunctionReturnSequenceType(Type.DOCUMENT,
      Cardinality.ONE,
      "The hello!"
      )
);
```

❶ As our new function takes parameters, we define an array of `org.exist.xquery.FunctionParameterSequenceType` objects in our `FunctionSignature`. You can see each of the parameters named, its type and cardinality defined, and a description provided for documentation purposes.

Now that we have written a signature for our function, we can implement the actual processing of the function. We can do this within the same class of our `h:hello-world` function from Example 16-14 by simply checking for a different calling signature within our `eval` function, handling the parameters we are interested in, and then executing our business logic. For example:

```
    @Override
    public Sequence eval(final Sequence[] args,
      final Sequence contextSequence) throws XPathException {

        final Sequence result;

        //act on the invoked function name
        if(isCalledAs(qnHelloWorld.getLocalName())) {
            result = sayHelloWorld();

        } else if(isCalledAs(qnSayHello.getLocalName())) { ❶

            final String greeter = args[0].itemAt(0).getStringValue(); ❷

            final String greeting;
            if(args[1].hasOne()) { ❸
                greeting = args[1].itemAt(0).getStringValue(); ❹
            } else {
                greeting = "hello"; ❺
            }

            final List<String> visitors =
               new ArrayList<String>(args[2].getItemCount());
            final SequenceIterator itVisitors = args[2].iterate(); ❻
            while(itVisitors.hasNext()) {
                final String visitor = itVisitors.nextItem().getStringValue(); ❼
                visitors.add(visitor);
            }

            result = sayHello(greeter, greeting, visitors); ❽

        } else {
            throw new XPathException("Unknown function call: " +
              this.getName().toString());
        }

        return result;
    }

    /**
     * Says a greeting to many people
     *
     * @param greeter The name of the person saying the greeting
     * @param greeting The greeting to use
     * @param visitors The visitors to say the greeting to
     *
     * @return A sequence of greetings, one for each visitor
     */
    private Sequence sayHello(final String greeter, final String greeting, ❾
      final List<String> visitors) throws XPathException {
        final Sequence results = new ValueSequence(); ❿

        for(final String visitor : visitors) {
            final StringValue result =
              new StringValue(greeter + " says " + greeting + " to " + visitor); ⓫
            results.add(result); ⓬
        }

        return results; ⓭
    }
```

❶ We add a switch on the name of our new function.

❷ From the first `Sequence` in the array, we extract the first item and get its string value. This is the value of our `greeter` parameter. Note that while indexes in sequences in XQuery start at `1`, in Java they start at `0`.

❸ As our second parameter, `greeting`, is optional, we first check whether an `xs:string` value or an empty sequence was given.

❹ If a value for the `greeting` parameter was given, we extract it.

❺ If an empty sequence was used for the `greeting` parameter, we fall back to the default greeting of `hello`.

❻ As our third parameter, `visitors`, is a sequence of one or more values, we obtain an iterator over the values.

❼ We iterate over each visitor name from `visitors` and add it to our list of visitors.

❽ We call our business logic, which generates a greeting for each visitor.

❾ This is our isolated business logic, which will generate our greetings.

❿ We create a new `ValueSequence` to hold each of the greetings that we wish to return to the XQuery.

⓫ We create a new `StringValue`, which represents an `xs:string` value to hold each of our greetings.

⓬ We add our greeting to the value sequence.

⓭ We return the value sequence, which now contains each of our greetings as strings. This is then returned to the XQuery by the `eval` method.

The source code for this example is included in the `HelloModule` code, as discussed earlier. To compile and deploy the module, see "Using the Hello World Module" on page 473. To test the example, execute the following XQuery from either eXist's Java Admin Client (see "Java Admin Client" on page 373) or eXide (see "eXide" on page 374):

```
xquery version "1.0";

declare namespace h = "http://hello";

h:say-hello("Adam", (), ("Elisabeth", "David"))
```

The result of the query should look similar to:

```
Adam says hello to Elisabeth
Adam says hello to David
```

You can experiment with providing different values for the second argument to the function and observe how the results change.

We have now built an XQuery extension function in Java for our internal module that can both accept multiple parameters of varying cardinality and return a sequence of results. Every internal module extension function that is written for XQuery in eXist follows this same pattern.

There are many, many internal modules of extension functions already provided with eXist, the vast majority of which are described at a high level in Appendix A. When you are developing your own modules, these are excellent examples from which to learn. You can find their source code in the folders *$EXIST_HOME/src/org/exist/xquery/functions* and *$EXIST_HOME/extensions/modules/src/org/exist/xquery/modules*.

If you do choose to write your own internal module for eXist, we strongly recommend reading both "Developing eXist" on page 483 and "Debugging eXist" on page 488, which will assist you with developing and debugging the Java code of your module running inside eXist.

Variable Declarations

While we have so far focused on defining functions within an internal module, you can also declare variables that live within the namespace of the internal module. This is most useful when your module wishes to expose a number of variables to XQuery that either represent some static constants or confer some configuration information.

You can define variable declarations in the constructor of your module class that extends `AbstractInternalModule` by calling the `declareVariable` method of the `super` class. For example, consider a module that provides mathematical constants:

```
public class MathConstantsModule extends AbstractInternalModule {

    protected final static String NS = "https://math/constants";
    protected final static String NS_PREFIX = "mc";

    public MathConstantsModule(
      final Map<String, List<? extends Object>> parameters) {

        super(functions, parameters);

        final Variable piApprox = ❶
          new VariableImpl(new QName("pi", NS, NS_PREFIX)); ❷
        piApprox.setValue(new FloatValue(22f / 7f)); ❸
        declareVariable(piApprox); ❹
        ❺
        final Variable speedOfLightApprox =
          new VariableImpl(new QName("speed-of-light", NS, NS_PREFIX));
          speedOfLightApprox.setValue(new FloatValue(1f / 299792458f));
        declareVariable(speedOfLightApprox);

        //...further variable declarations omitted for brevity
    }

    //...module body omitted for brevity

}
```

❶ We create a new variable by instantiating an instance of `org.exist.xquery.VariableImpl`.

❷ The variable must, of course, be named, but that name must reside within the namespace of the internal module.

❸ We set the value of the variable.

❹ We declare the variable within the internal module.

❺ We again create a variable, set its value, and declare it.

Module Configuration

You have probably noticed by now the `parameters` argument that is given to your internal module class's constructor. So far we have largely ignored this, and we simply pass it on to the `super` class's constructor as required. When you configure eXist to use your internal module in *$EXIST_HOME/conf.xml*, you can also specify configuration parameters inside the module declaration, and these parameters will be parsed, extracted, and passed in the `parameters` argument of your module's constructor.

This is a rather simple configuration facility, and is the same used elsewhere for scheduled tasks (see "Java Jobs" on page 441, "Startup Triggers" on page 446, and "Java Triggers" on page 457). As well as those other mechanisms, which share the same configuration semantics, the `xslfo` module (see `xslfo`) makes use of such parameters and serves well as an example of how to do this for your own internal modules.

Developing eXist

The eXist development community is always open to new contributors, beginners or experts, from those who just want to fix a typo in the documentation to those who want to reengineer the core storage of the database. Whatever your level of expertise, all contributions are treated equally and follow the same process to reach acceptance. eXist makes use of the fork and pull GitHub model of collaborative development. Simply put, all contributors follow the same three steps:

1. *Fork* the eXist Git repository that you are interested in contributing to from *https://github.com/eXist-db* to your own GitHub user/organization.
2. Make your changes within your fork (preferably using git-flow).

   If you are modifying Java code, you must run eXist's test suite (see the `test` entry in Table 16-9) and check that there are no regressions.

3. When you are happy with your completed changes, you send a *pull request* via GitHub.

When you are adding new source code folders or JAR files to eXist, it is important to make sure that each of the IDE project files is updated to support your changes before sending a pull request.

Now, to be clear, *all* pull requests to eXist are evaluated by at least one member of the eXist CDT (Core Development Team), each of whom has a responsibility to evaluate and merge pull requests in a timely fashion. Even the members of the CDT are not exempt from this process; they too must submit their changes by pull request and have them merged by a (different) member of the CDT. Nothing is ever merged into eXist without at least two people knowing about it and agreeing that it improves the status quo. For full details of the development process employed by eXist, see *https://github.com/eXist-db/eXist#contributing-to-exist*.

While simple bug fixes and updates are often obvious and easy to develop, more complicated bug fixes or features should be openly discussed on the *eXist-development mailing-list*. There are two main advantages to doing this:

Avoiding duplication
: With each contributor communicating his intentions clearly, we can hopefully avoid any duplication of work, as it is possible otherwise that two people may be attempting to solve the same problem simultaneously!

Continuity
: This ensures that proposed new features complement the community vision for eXist. It is very unlikely that a new feature would be rejected outright; however, there are often many ways to approach the same problem, and an open discussion between peers can often bring new insights!

Should you have any pressing development concerns, or need to chase a pull request, at the time of writing, the eXist CDT comprises Tobi Krebs, Wolfgang Meier, Leif-Jöran Olsson, Adam Retter, Dmitriy Shabanov, Joern Turner, Dannes Wessels, and Lars Windauer, all of whom should be contactable through the *eXist-open* and *eXist-development* mailing lists. The members of the CDT are not considered special in any way; they are simply people who have made many contributions to eXist over time and have a feeling for what eXist *means*. Anyone is very welcome to join the CDT if they are willing to invest time to review and merge pull requests in the longer term.

On a more technical level, it is perhaps pertinent to mention here that eXist is written almost entirely in Java, its XQuery parser is written in ANTLR v2, and it uses the Apache Ant build system (although there is an embryonic effort underway to migrate to Apache Maven). In addition, many add-ons for eXist (such as the dashboard and demo applications) are written in HTML, JavaScript, and XQuery. The documentation for eXist is entirely authored in DocBook v5.

You can use any IDE or other text editor that you wish to when developing eXist, but for convenience IDE project files can be found for NetBeans, IntelliJ, and Eclipse inside *$EXIST_HOME*.

Each of the IDE projects is configured to build eXist, but note that the IntelliJ build configuration does not compile in the AspectJ aspects that eXist uses for database and XQuery execution security enforcement! Therefore, when eXist is run from IntelliJ it will run with very few security constraints and will not be suitable for testing database operations.

Building eXist from Source

As eXist is an open source project, it is fundamentally important that anyone should be able to download the source code and compile their own version of it. The developers of eXist have gone to great lengths to ensure that the build process is simple for all to use. Anyone can download the source code, compile it, and compare it with a released version of eXist to make sure they are the same and that some nefarious person or organization has not interfered with the software, which enables transparency. Another nice outcome of having an easy-to-use build system is that any user can compile eXist, for the purpose of either having the latest and greatest version in advance of the next release, or contributing fixes or features back as a developer.

The eXist source code repository was recently moved from its previous home on SourceForge to GitHub. It is laid out using the git-flow scheme; thus, all of the development for the next release of eXist takes place in the `develop` branch, which is the *default branch for eXist*. The `master` branch represents the latest stable release of eXist; however, to be certain which version of eXist you will be working with, it is simpler to use the correct *tag*. At the time of writing, the latest tagged release of eXist was `eXist-2.1`. You will need to have Git installed if you wish to pull the latest source code directly from GitHub; see *http://www.git-scm.com* to get an installer for your platform. From the same website, there are also various GUI clients, such as SourceTree.app and GitHub Client, available if you prefer a graphical interface. If you are interested in reading further details on how eXist is developed and even potentially contributing, see "Developing eXist" on page 483.

Not all aspects of the eXist project are on GitHub, only the source code and issue tracker. The mailing lists and downloads of compiled releases remain at SourceForge for the time being. Whatever the infrastructure of the project, links to the latest locations will always be available from the eXist website.

The eXist source code is built with the Apache Ant build tool. eXist includes a copy of the Ant runtime in its *$EXIST_HOME/lib/tools/ant* folder so that you do not need to separately install it. eXist's Ant build scripts are just a series of XML files and can be found in *$EXIST_HOME/build.xml* and *$EXIST_HOME/build/scripts*. However, rather than having you use them directly, eXist provides two executable scripts that run Ant with the appropriate build scripts: *$EXIST_HOME/build.sh* (for Unix/Linux/Mac platforms) and *$EXIST_HOME/build.bat* (for Windows platforms).

The settings for the build are configurable: take a look at *$EXIST_HOME/build.properties*. This is particularly useful when you are working in a corporate environment behind a proxy server, as eXist may attempt to download some resources as part of the build. You can configure this using the proxy settings in *build.properties*.

When executing the build script, you can provide one or more *targets* that describe which build action(s) you wish to take. There are many build targets available, some of the most useful of which are described in Table 16-9. The table is followed by Example 16-15, which demonstrates the typical sequence of building eXist from source code.

Table 16-9. Useful eXist Ant build targets

Target	Description
`clean`	Removes all compiled code. Useful when you wish to do a clean recompile.
`clean-all`	Similar to `clean`, but also deletes the database. *Use with care!*
`jar`	Compiles just the eXist source code into JAR files.
`extension-modules`	Builds any extension modules that are defined and enabled in *$EXIST_HOME/extensions/build.properties*. Can be used by itself to compile in new extension modules to an existing installation.
`wrapper`	Builds the Java Service Wrapper for eXist. See "Windows Linux and Other Unix" on page 407.

Target	Description
`sign`	Signs the built JAR files so that they may be used in a security-restricted environment such as Java Web Start or a restricted application server.
`all`	The default build target. Builds the eXist source code, the Java Service Wrapper, the betterFORM XForms extension, the extension modules, and EXPath package support.
`rebuild`	A useful shortcut when making changes and rebuilding; calls the targets `clean` and then `all`.
`dist`	Builds a distribution of eXist. The result is a folder in *$EXIST_HOME/dist* that can be distributed to other machines and installed. There are also `dist-zip` and `dist-tgz` targets, which will create a ZIP file or tarball, respectively, under *$EXIST_HOME/dist* for you to distribute to other machines and install.
`dist-war`	Creates a WAR file in *$EXIST_HOME/dist* that can be deployed to any Java application server, such as Apache Tomcat.
`installer`	You can create installers for eXist, just like the binary releases provided on SourceForge. On completion the installers can be found in *$EXIST_HOME/installer*. However, this takes a little more effort, as you need to install the supporting tools IzPack 4.3.5 and Launch4j. The paths to the supporting tools then need to be configured in *$EXIST_HOME/build.properties*.
`app`	Similar to `installer`, but specific to Mac OS X. Creates a self-contained application and packages it in an Apple disk image (as a *.dmg* file). On completion, the disk image can be found in *$EXIST_HOME/dist*. There is also an `app-signed` target, which does the same as `app` but also signs the application. You will, however, need a valid Apple developer certificate installed for this to work. For further details, see *https://developer.apple.com/support/technical/certificates/*.
`test`	If you are making modifications to the eXist source code, it is essential to execute the test suite to ensure that you have not introduced any regressions. The output of the test suite can be found in the web page report located at*$EXIST_HOME/test/junit/html/index.html*.

Example 16-15. Typical sequence of building eXist from source code

```
git clone https://github.com/eXist-db/exist.git ❶
git checkout tags/eXist-2.1 ❷

./build.sh ❸
```

❶ Clone the eXist source code from GitHub. If you are planning to contribute, you should first fork the repository and then clone your own fork.

❷ Check out the `eXist-2.1` release tag from the repository. You can view all available tags by running `git tag -l`. If you wish to track the latest version of eXist, you need not check out a tag; instead, running `git branch -a` should show that you are on the `develop` branch.

❸ Build the eXist source code. By default, this builds the `all` target.

Remember that while it is, of course, possible to make distributions and installers for eXist from the source code, you can also work with eXist *in place*. You do so by checking out the source code, building it, and then running it directly by using *$EXIST_HOME/bin/startup.sh* (or *$EXIST_HOME/bin/startup.bat* on Windows), or even installing eXist as a service (see "Installing eXist as a Service" on page 405). A major advantage of this approach is that you can easily update to a newer version of eXist by using Git to pull changes if you are tracking the `develop` branch, or by checking out a newer release tag when it becomes available.

Make sure to back up your config and database before switching branches with Git!

Debugging eXist

If you are using one of the IDEs for which eXist provides project files (NetBeans, IntelliJ, and Eclipse), then these projects are already set up to enable you to debug either the eXist Java Admin Client or the eXist server. It is also worth remembering that you can debug the Java Admin Client in embedded mode, which can sometimes provide a simple mechanism for debugging the database core without your needing to run the full server.

However, if you are not using one of the supported IDEs or wish to debug eXist code that is running on a remote server, then your only real option is to use the Java Debugging Wire Protocol (JDWP). It is also worth mentioning that each of the supported IDEs functions as an excellent debugger when you're debugging eXist remotely. JDWP supports using a TCP/IP socket to communicate between the application you are debugging and the debugger on all platforms; depending on how it's configured, you can also run this across a network. Between the application that you wish to debug and the debugger, JDWP can work in either direction. That is, you can start up the JVM running your Java application that you wish to debug as either:

- A JDWP server that will listen for connection requests from a debugger
- A JDWP client that will connect to a remote debugger

To enable JDWP, where your Java application offers a JDWP server, pass the following options to the JVM when you start your Java application:

```
-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=127.0.0.1:4000
```

To enable JDWP, where your Java application connects using JDWP to a debugger, pass the following options to the JVM when you start your Java application:

```
-agentlib:jdwp=transport=dt_socket,server=n,suspend=y,address=127.0.0.1:4005
```

The `suspend` parameter may be set to either `y` or `n`. When set to `y` (yes), it will cause your Java application not to start running within the JVM immediately, but to wait until a debugger connects to the application in server mode, or the Java application connects to the debugger in client mode. This can be very useful with applications like eXist where you may wish to debug the database startup process.

The `address` parameter will cause the JVM to listen on a specific IP address and TCP port for JDWP requests in server mode, or to connect to a debugger listening on that specific address and port in client mode. If you use the *localhost* address of `127.0.0.1`, then the debugger must be running on the same machine. If you wish to debug across the network, you need to specify the IP address of the server's network interface in server mode (or you can omit the IP address entirely to listen on all server addresses), or the client's IP address in client mode.

We have discussed the JDWP settings in the context of any Java application, as eXist can be set up to run in many different ways, and you may need to add these options to whichever mechanism you are using to start eXist on your local machine or server. For further details, see *http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/conninv.html#Invocation*. However, if you are using the *$EXIST_HOME/client.sh* and/or *$EXIST_HOME/startup.sh* scripts to start eXist, then you can simply uncomment the line that starts `DEBUG_OPTS` near the top of those files to have JDWP enabled in server mode. Remember that you will have to restart eXist for these changes to take effect.

Remote debugging with the NetBeans IDE

While any IDE or client that supports JDWP can be used as a debugger against the Java code that makes up eXist, here we show how you can use the NetBeans IDE to connect to eXist running as a JDWP server. Before continuing you must start eXist running as a JDWP server, as discussed earlier.

eXist ships with project files for NetBeans, so you can simply go to the File→Open Project menu item in NetBeans and select your *$EXIST_HOME* folder. Once the project has loaded, you need to attach the NetBeans debugger to the eXist JDWP server by choosing the Debug→Attach Debugger menu item (see Figure 16-7).

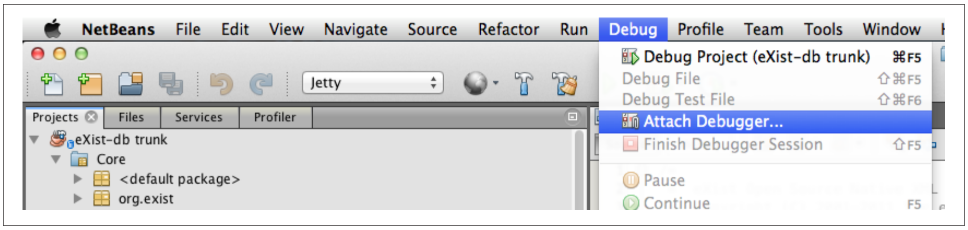


Figure 16-7. NetBeans: opening the Attach dialog

This will bring up the Attach dialog box. Assuming that eXist is running on the same machine as NetBeans and that you have used the default TCP port that eXist's JDWP settings are configured for (4000), you should set the following options in the dialog (see Figure 16-8):

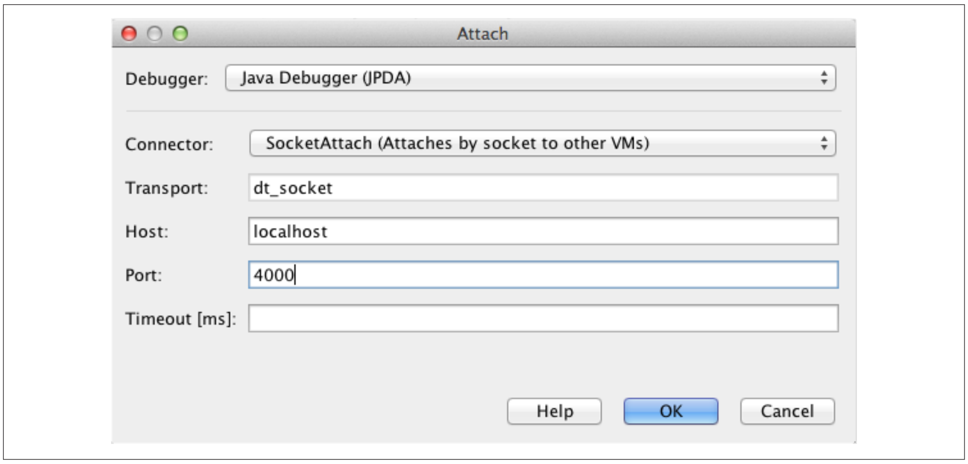
Debugger
  Java Debugger (JPDA)

Connector
  SocketAttach (Attaches by socket to other VMs)

Transport
  dt_socket

Host
  localhost

Port
  4000


Attach

Debugger: Java Debugger (JPDA)

Connector: SocketAttach (Attaches by socket to other VMs)

Transport: dt_socket

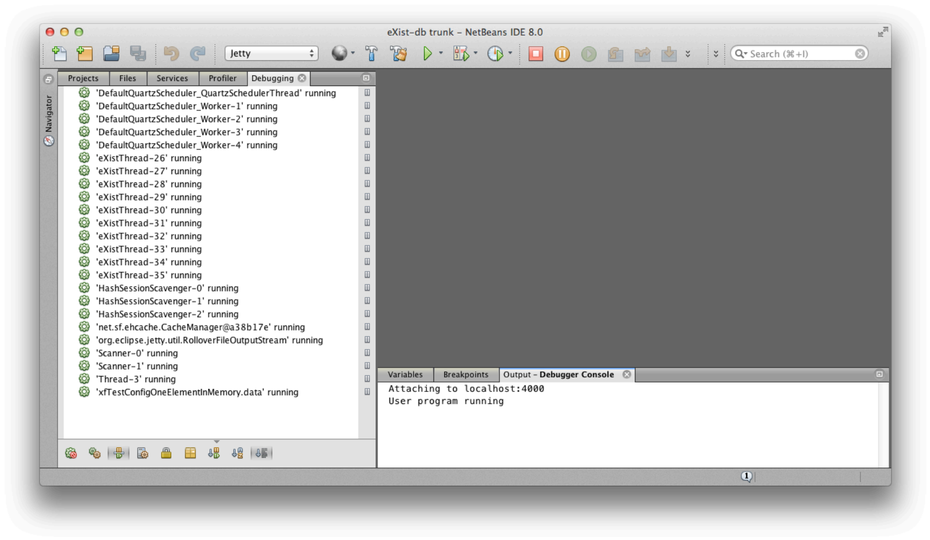
Host: localhost

Port: 4000

Timeout [ms]:

Figure 16-8. NetBeans: the Attach dialog

After you click the OK button, NetBeans will attempt to connect its debugger to eXist. All being well, you should initially see a confirmation that the debugger has connected to eXist and a list of the running threads that make up eXist, as shown in Figure 16-9.



```
'DefaultQuartzScheduler_QuartzSchedulerThread' running
'DefaultQuartzScheduler_Worker-1' running
'DefaultQuartzScheduler_Worker-2' running
'DefaultQuartzScheduler_Worker-3' running
'DefaultQuartzScheduler_Worker-4' running
'eXistThread-26' running
'eXistThread-27' running
'eXistThread-28' running
'eXistThread-29' running
'eXistThread-30' running
'eXistThread-31' running
'eXistThread-32' running
'eXistThread-33' running
'eXistThread-34' running
'eXistThread-35' running
'HashSessionScavenger-0' running
'HashSessionScavenger-1' running
'HashSessionScavenger-2' running
'net.sf.ehcache.CacheManager@a38b17e' running
'org.eclipse.jetty.util.RolloverFileOutputStream' running
'Scanner-0' running
'Scanner-1' running
'Thread-3' running
'xfTestConfigOneElementInMemory.data' running
```

```
Attaching to localhost:4000
User program running
```

Figure 16-9. NetBeans: confirmation of attachment and list of running threads

From here you can perform all the typical Java debugging steps, such as setting break-points in the eXist code, showing variable values, getting stack dumps, and stepping through the running code, stack frame by stack frame. Java debugging in itself is a huge and advanced topic ,and we do not presume to teach it here; however, hopefully if you are an aspiring or advanced Java developer, we have provided you with the information that you need to get started with debugging eXist's code base.

APPENDIX A

XQuery Extension Modules

This appendix provides an overview of the available XQuery extension modules in eXist.

You might have noticed that in other chapters we've often mentioned and described extension modules. For instance, "Controlling the Database from Code" on page 107 explains in part how to manipulate the database using the `xmldb` extension module. This is because we think it is better to describe an extension module within its context as much as possible. However, there are a lot of extension modules we haven't covered elsewhere, so we provide an overview here to assist you.

We won't handle all the extension modules that come with eXist. As is often the case with open source products, experiments creep in and are not removed. Some of the modules seem to hold interesting functionality, but documentation is missing. And some are very specialized and of interest only to a very small group of users. We therefore took the liberty to describe only those modules that we thought would be of general interest and for which at least some kind of documentation was available (often just the information in eXist's function documentation browser). Apologies to all module creators whose modules we left out. If you think something we left out deserves a place in this appendix, please drop us a line and we'll try to rectify this in a future edition of the book.

What we also won't do here is describe all functions in detail. eXist provides an online function documentation browser, and we saw no reason to duplicate this information; it would have made the book too heavy and the information would probably soon be out of date. What we *will* do is provide you with a list of modules available in the version of eXist (2.1) that this book was written for, tell you a little bit about them, and provide examples when we think it is appropriate.

Extension Modules by Category

This section contains a list of the extension modules discussed here, grouped by category. Categorizing is always a bit arbitrary, but hopefully this will you help find the module/functionality you're looking for more quickly. The brief descriptions provided for each module give an indication of what they are used for or contain.

Core

These modules form a core of useful extension functionality that you would not want to do without when working with XQuery in eXist:

Module	Description
`file`	Managing files and directories on the filesystem
`http`	Performing HTTP requests as a client (EXPath)
`httpclient`	Performing HTTP requests as a client (eXist native)
`map`	`map` data type functions
`request`	Handling HTTP requests
`response`	Controlling the HTTP response
`restxq`	Module for eXist's RESTXQ interface
`restxqex`	Module for eXist's RESTQX resource function registry
`session`	Functions for working with the HTTP session
`sm`	Security Manager functions
`system`	Information about eXist and the system environment
`transform`	XSLT-related functions
`util`	Utility functions
`validation`	Validating XML
`xmldb`	Core module for working with the database

Data Handling

Modules in this category provide additional data handling capabilities to eXist, such as reading and writing of ZIP files, conversion to JSON, and image handling:

Module	Description
`compression`	Handling of ZIP, GZIP, and TAR files
`exi`	Working with EXI (XML binary)
`image`	Performing operations on images stored in eXist
`jfreechart`	Generating charts using the JFreeChart library
`json`	Transforming XML into JSON

Module	Description
`jsonp`	Transforming XML into JSONP
`zip`	Accessing resources in ZIP files
`xqjson`	Serializing XML into JSON, and parsing JSON into XML

Data Type Extensions

These modules supply additional functionality for existing data types:

Module	Description
`datetime`	Date and time operations
`exiftool`	Extracting EXIF data from binary files
`math`	Mathematical functions
`sequences`	Working with sequences
`text`	Text-searching extension functions
`xmldiff`	Comparing XML documents

Database Functionality

This category of modules enables additional database-related functionality, like caching, scheduling, and the repository manager:

Module	Description
`cache`	Systemwide global cache
`counter`	Persistent counters
`metadata`	Document key/value metadata functions
`repo`	Working with the EXPath repository manager
`scheduler`	Scheduling jobs
`versioning`	Access to versioning extensions

Indexing

These are modules that work with indexes:

Module	Description
`contentextraction`	Indexing binary content
`ft`	Querying Lucene's full-text index
`kwic`	Keywords in context
`ngram`	NGram index–related functions
`sort`	Indexes for efficient sorting

Protocols/Interfaces

These modules enable access to various communication protocols and interfaces from XQuery code:

Module	Description
`ftpclient`	Performing FTP requests as a client
`jndi`	eXist JNDI interface
`mail`	Email-related functions
`sql`	Database access using JDBC
`xmpp`	Access to the XMPP instant messaging protocol

XML Technologies

These modules provide access to additional XML technologies:

Module	Description
`xmlcalabash`	Access to the XML Calabash XProc module (experimental)
`xslfo`	Rendering with XSL-FO

XQuery

These are modules that provide additional XQuery-related functionality:

Module	Description
`inspect`	Retrieving xqDoc information from XQuery modules
`xqdm`	Retrieving xqDoc information from XQuery modules (XQuery 1.0 only)

Extension Module Descriptions

This section lists all the extension modules and tells you what they're useful for, sometimes providing a small example.

Every module starts with a table that has the following information:

Module name

Description

A (very) short description of the module.

Namespace

The namespace for the module, including the preferred prefix. You can, of course, decide to use a different prefix.

Type/default status

The type of the module (Java or XQuery), and whether it is enabled by default and (for Java modules) built into eXist. Information about what this means exactly and how to change it can be found in "Enabling Extension Modules" on page 128.

Class

The Java class name (for Java-based modules) or the location (for XQuery-based modules) of the module.

cache

Description

Systemwide global cache

Namespace

`cache="http://exist-db.org/xquery/cache"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*; enabled in *$EXIST_HOME/extensions/build.properties*

Class

`org.exist.xquery.modules.cache.CacheModule`

The `cache` extension module adds a systemwide global cache to eXist. It allows you to create data that is not only sessionwide (for which you would use the `session` module) but also systemwide. It adds true global variables to eXist, with all the benefits but also the ugly dangers of global data.

The following example creates a cache (or gets a reference to it if it already exists) and stores the current date/time in it. The `cache:put` function returns the previous value, which we display alongside the new value:

```
let $cache := cache:cache('test')
let $previous-value := cache:put('test', 'KEY', current-dateTime())
return
    <cache previous="{$previous-value}" now="{cache:get('test', 'KEY')}"/>
```

If you run this example from different sessions (run it, close your browser, run it again), you'll find that the value stored is retained: the `now` value from the first session is the `previous` value in the second session.

compression

Description

Handling of ZIP, GZIP, and TAR files

Namespace

`compression="http://exist-db.org/xquery/compression"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*; enabled in *$EXIST_HOME/extensions/build.properties*

Class

`org.exist.xquery.modules.compression.CompressionModule`

This module allows you to create and read ZIP, GZIP, and TAR compressed files.

Let's take a ZIP file as example (GZIP and TAR work likewise). You can create a ZIP file by calling `compression:zip` and feeding it a sequence of items to compress:

- If such an item is of type `xs:anyURI`, it is supposed to be the URI of a resource or a collection. A collection is zipped in its entirety and its directory structure is mimicked in the ZIP file.
- If it's an `entry` element, its content is zipped. It takes three attributes:
  - — The `name` attribute is used as the path of the resource in the ZIP file.
  - — The `type` attribute tells the module how to interpret the contents of the `entry` element. Possible values are `collection`, `uri`, `binary`, `xml`, and `text`.
  - — The optional `method` attribute tells the module whether the resource should be compressed before being added to the ZIP file (the default) or simply stored as is. (The latter is useful when, for instance, creating ebook files in ePub format, which requires a noncompressed Internet media type file.)

Here is an example that zips a part of the book's examples and adds an extra XML file:

```
let $stuff-to-compress as item()+ := (
    xs:anyURI('/db/eXist-book/getting-started'),
    <entry name="EXTRA/x.xml" type="xml">
        <Extra>Some extra stuff...</Extra>
    </entry>
)
let $file-uri := '/some/path/on/your/disk/out.zip'
let $zipfile := compression:zip($stuff-to-compress, true())
return
    file:serialize-binary($zipfile, $file-uri)
```

contentextraction

Description

Indexing binary content

Namespace

`contentextraction="http://exist-db.org/xquery/contentextraction"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*; enabled in *$EXIST_HOME/extensions/build.properties*

Class

`org.exist.contentextraction.xquery.ContentExtractionModule`

The `contentextraction` module is an absolute wonder! It allows you to extract text content from binary resources like Word or PDF files, based on the Apache Tika toolkit. Everyone who has ever coped with this problem knows how difficult, frustrating, and time-consuming it can be.

The Tika toolkit supports many formats. For instance, you can feed it an HTML page and it will output well-formed and valid XHTML. Feed it a text file (in any character encoding) and it will neatly create paragraphs from it. Other formats supported include Microsoft Office, Open Document, PDF, ePub, RTF, mbox, and even the textual/metadata parts of audio, image, and video files (see the Tika website for details).

To use the `contentextraction` module, insert the following `import module` statement in the prolog of your XQuery script:

```
import module namespace content="http://exist-db.org/xquery/contentextraction"
  at "java:org.exist.contentextraction.xquery.ContentExtractionModule";
```

The module has three functions (use the XQuery Function Documentation app from the dashboard to inspect them). The following code will return the metadata and contents of a (recognized) binary file:

```
let $file := 'some/path/to/a/binary/file'
return
    content:get-metadata-and-content(util:binary-doc($file))
```

An interesting use case of `contentextraction` is, of course, indexing using the full-text index capabilities of eXist and allowing the user to search binary documents stored in the database (read more about this in "Manual Full-Text Indexing" on page 301). For indexing large documents the third, somewhat complicated, function, `content:stream-content`, comes in handy. There is an interesting content extraction example in the eXist-db demo apps (available through the dashboard) that uses this.

counter

Description

Persistent counters

Namespace

`counter="http://exist-db.org/xquery/counter"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*; enabled in *$EXIST_HOME/extensions/build.properties*

Class

`org.exist.xquery.modules.counter.CounterModule`

This module adds eXist global persistent counters. It allows you to count things and keep the results across sessions and database restarts.

datetime

Description

Date and time operations

Namespace

`datetime="http://exist-db.org/xquery/datetime"`

Type/default status

Java; disabled in *$EXIST_HOME/conf.xml*; enabled in *$EXIST_HOME/extensions/build.properties*

Class

`org.exist.xquery.modules.datetime.DateTimeModule`

This module has functions for working with dates and times, like parsing strings into dates, counting days in a month, and creating date/time ranges. It can also format dates/times into more user-friendly strings. Much of its functionality has already been superceded by newer (more portable) modules.

exi

Description

Working with EXI (XML binary)

Namespace

`exi="http://exist-db.org/xquery/exi"`

Type/default status

Java; disabled in *$EXIST_HOME/conf.xml*; enabled in *$EXIST_HOME/extensions/build.properties*

Class

`org.exist.xquery.modules.exi.ExiModule`

This module allows encoding and decoding XML into/from the Efficient XML Interchange (EXI) format. EXI is a binary format for XML data.

exiftool

Description

Extracting EXIF data from binary files

Namespace

`exiftool="http://exist-db.org/xquery/exiftool"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*; enabled in *$EXIST_HOME/extensions/build.properties*

Class

`org.exist.exiftool.xquery.ExiftoolModule`

This module extracts EXIF data from binary files—for instance, photos shot with a modern digital camera. It relies, however, on the presence of Perl at */usr/bin/perl*.

file

Description

Managing files and directories on the filesystem

Namespace

`file="http://exist-db.org/xquery/file"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*; enabled in *$EXIST_HOME/extensions/build.properties*

Class

`org.exist.xquery.modules.file.FileModule`

This module allows you to work with files and directories on the filesystem (outside the database). There are functions like `file:mkdir`, `file:move`, and `file:serialize` (in several variants). You need DBA privileges to use it.

For Windows, passing file and directory names to functions in the `file` module can lead to some odd behavior. This only seems to work when you do *not* specify the *file://* prefix, *only* use backslashes as the path separator, and *always* start the name with a backslash (e.g., *\C:\test\erik\test.xml* instead of *file://c:/test/erik/test.xml*).

ft

Description

Querying Lucene's full-text index

Namespace

`ft="http://exist-db.org/xquery/lucene"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*

Class

`org.exist.xquery.modules.lucene.LuceneModule`

This module provides access to Lucene's full-text index results. How to do this is described in "Searching with the Full-Text Index" on page 292.

ftpclient

Description

Performing FTP requests as a client

Namespace

`ftpclient="http://exist-db.org/xquery/ftpclient"`

Type/default status

Java; disabled in *$EXIST_HOME/conf.xml*; enabled in *$EXIST_HOME/extensions/build.properties*

Class

`org.exist.xquery.modules.ftpclient.FTPClientModule`

This module allows you to use eXist as a client against an FTP server. You can read and write (binary) files and get directory listings.

http

Description

Performing HTTP requests as a client (EXPath)

Namespace

`http="http://expath.org/ns/http-client"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*

Class

`org.expath.exist.HttpClientModule`

This is a module that allows you to send out HTTP requests according to the EXPath HTTP Client module specifications (refer to *http://www.expath.org/modules/http-client/* and *http://www.expath.org/spec/http-client*). The functions expect an XML fragment that contains the specification of the request and returns an XML fragment with its results.

For instance, the following example fires an HTTP `GET` request at the local eXist REST interface (see "Querying the Database Using REST" on page 94). The result is a directory listing of */db*:

```
let $http-request-data := <request xmlns="http://expath.org/ns/http-client"
  method="GET" href="http://localhost:8080/exist/rest/db"/>
return
    http:send-request($http-request-data)
```

The result is another XML fragment that provides you with the result and all the HTTP properties surrounding this:

```
<http:response xmlns:http="http://expath.org/ns/http-client"
  status="200"
  message="OK">
    <http:header name="date" value="Tue, 20 Nov 2012 14:49:55 GMT"/>
    <http:header name="set-cookie"
         value="JSESSIONID=1jkifx0ip6jin6kswi0n96q54;Path=/exist"/>
    <http:header name="expires" value="Thu, 01 Jan 1970 00:00:00 GMT"/>
    <http:header name="content-type" value="application/xml; charset=UTF-8"/>
    <http:header name="last-modified" value="Tue, 25 Sep 2012 08:15:31 GMT"/>
    <http:header name="created" value="Tue, 25 Sep 2012 08:15:31 GMT"/>
    <http:header name="transfer-encoding" value="chunked"/>
    <http:header name="server" value="Jetty(8.1.3.v20120416)"/>
    <http:body media-type="application/xml"/>
</http:response>
<exist:result xmlns:exist="http://exist.sourceforge.net/NS/exist">
    <exist:collection name="/db" created="2012-09-25T10:15:30.686+02:00"
         owner="SYSTEM" group="dba" permissions="rwxr-xr-x">
```

```
        <exist:collection name="eXist-book" created="2012-09-05T10:03:28+02:00"
               owner="admin" group="dba" permissions="rwxr-xr-x"/>
        <!-- Etc. -->
    </exist:collection>
</exist:result>
```

eXist also contains an HTTP Client module of its own (see the next section). Both modules have roughly the same functionality, but if you're ever going to do cross-platform XQuery programming, the EXPath one is the better choice.

httpclient

Description

Performing HTTP requests as a client (eXist native)

Namespace

`httpclient="http://exist-db.org/xquery/httpclient"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*; enabled in *$EXIST_HOME/extensions/build.properties*

Class

`org.exist.xquery.modules.httpclient.HTTPClientModule`

This module allows you to send out HTTP requests as if eXist were a client. It has functions like `httpclient:put` and `httpclient:get`. All functions return an extensive XML fragment with details of the request's response. An example of using the `httpclient:post` function can be found in "POST Requests" on page 98.

eXist also contains an EXPath-conformant HTTP Client module (see `http`). Both modules have roughly the same functionality, but if you're ever going to do cross-platform XQuery programming, the EXPath one is the better choice.

image

Description

Performing operations on images stored in eXist

Namespace

`image="http://exist-db.org/xquery/image"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*; enabled in *$EXIST_HOME/extensions/build.properties*

Class

`org.exist.xquery.modules.image.ImageModule`

This module has functions for performing various operations on images stored in eXist, like requesting their metadata, width, and height. It can also crop and scale images and do bulk thumbnail conversions.

Most functions take the image as `xs:base64Binary` for input. Converting is easy: use the `util:binary-doc` function. The following example tells you the height of an image stored in eXist:

```
let $image-uri as xs:anyURI := xs:anyURI('/db/some/path/to/an/image/file')
return
    image:get-height(util:binary-doc($image-uri))
```

There is also an example of using this module in "RESTXQ" on page 353.

inspect

Description

Retrieving xqDoc information from XQuery modules

Namespace

`inspect="http://exist-db.org/xquery/inspection"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*

Class

`org.exist.xquery.functions.inspect.InspectionModule`

This module allows you to inspect XQuery modules and their xqDoc documentation. More about this can be found in "XQuery Documentation with xqDoc" on page 125.

jfreechart

Description

Generating charts using the JFreeChart library

Namespace

`jfreechart="http://exist-db.org/xquery/jfreechart"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*; enabled in *$EXIST_HOME/extensions/build.properties*

Class

`org.exist.xquery.modules.jfreechart.JFreeChartModule`

This module allows you to create all kinds of beautiful pie, bar, and other fancy charts using the JFreeChart library.

jndi

Description

eXist JNDI interface

Namespace

`jndi="http://exist-db.org/xquery/jndi"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*; enabled in *$EXIST_HOME/extensions/build.properties*

Class

`org.exist.xquery.modules.jndi.JNDIModule`

This module is an interface for XQuery to use the Java Naming and Directory Interface (JNDI).

json

Description

Transforming XML into JSON

Namespace

`json="http://www.json.org"`

Type/default status

XQuery; enabled in *$EXIST_HOME/conf.xml*

Location

resource:org/exist/xquery/lib/json.xq

This module transforms (data-centric) XML into straightforward JSON. The module contains a number of functions, but the main entry point is `json:xml-to-json`. The rules for the JSON conversion are the same as those used in the JSON Serializer (see "JSON serialization" on page 121). For an alternative, see also `xqjson`.

jsonp

Description

Transforming XML into JSONP

Namespace

`jsonp="http://www.jsonp.org"`

Type/default status

XQuery; enabled in *$EXIST_HOME/conf.xml*

Location

resource:org/exist/xquery/lib/jsonp.xq

This module transforms (data-centric) XML into JSON, which is wrapped inside a named JavaScript function. This usage pattern is known as JSONP. The module contains a number of functions, but the only really useful one is its main entry point, `jsonp:xml-to-jsonp`. The first argument is the XML node to convert to JSON, and the second is the name to use for the JavaScript function wrapper.

kwic

Description

Keywords in context

Namespace

`kwic="http://exist-db.org/xquery/kwic"`

Type/default status

XQuery; enabled in *$EXIST_HOME/conf.xml*

Location

resource:org/exist/xquery/lib/kwic.xql

This module allows you to easily show fragments of text found in full-text or NGram search operations. The module is described in "Using Keywords in Context" on page 297.

mail

Description

Email-related functions

Namespace

`mail="http://exist-db.org/xquery/mail"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*; enabled in *$EXIST_HOME/extensions/build.properties*

Class

`org.exist.xquery.modules.mail.MailModule`

This module allows you to send and receive email messages using the Java mail libraries. The following example shows sending a message through a straight SMTP server (without required authorization):

```
let $receiver-email := 'your@email.address'
let $smtp-server := 'your.smtp.server.address'
let $message :=
<mail>
    <from>E. Mailtester &lt;emailtester@dummy.org&gt;</from>
    <to>{$receiver-email}</to>
    <subject>Testing send-email()</subject>
    <message>
        <text>Test message, Testing 3, 2, 1 at {current-dateTime()}</text>
        <xhtml>
            <html>
                <head>
                    <title>Testing</title>
                </head>
                <body>
                    <h1>Testing</h1>
                    <p>Test message, Testing 3, 2, 1 at {current-dateTime()}</p>
                </body>
            </html>
        </xhtml>
    </message>
</mail>
let $props :=
    <properties>
        <property name="mail.smtp.auth" value="false"/>
        <property name="mail.smtp.port" value="25"/>
        <property name="mail.smtp.host" value="{$smtp-server}"/>
    </properties>

let $session := mail:get-mail-session( $props )
```

```
return
    mail:send-email($session, $message)
```

map

Description

`map` data type functions

Namespace

`map="http://www.w3.org/2005/xpath-functions/map"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*

Class

`org.exist.xquery.functions.map.MapModule`

This module supports the `map` data type, as described in "The map data type proposed for XQuery 3.1" on page 116.

math

Description

Mathematical functions

Namespace

`math="http://exist-db.org/xquery/math"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*; enabled in *$EXIST_HOME/extensions/build.properties*

Class

`org.exist.xquery.modules.math.MathModule`

This module includes various mathematical functions like `sin`, `pi`, and `exp`.

This module is now deprecated by the math functions from the namespace `http://www.w3.org/2005/xpath-functions/math` as defined by the XPath and XQuery Functions and Operators 3.0 specification.

metadata

Description

Document key/value metadata functions

Namespace

`md="http://exist-db.org/metadata"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*; enabled in *$EXIST_HOME/extensions/build.properties*

Class

`org.exist.storage.md.xquery.MetadataModule`

This module incudes functions for associating key/value metadata pairs with a document, rerieving key/value pairs associated with documents, and searching for documents based on key/value pairs.

While useful, the `metadata` module should be considered highly experimental and not yet ready for production use.

For example, say you wish to store a key/value pair of metadata for the document */db/my-docs/doc1.xml*. You can do this as follows:

```
xquery version "1.0";

import module namespace md = "http://exist-db.org/metadata";

md:set-value(doc("/db/my-docs/doc1.xml"), "roll-number", 12345)
```

Each time you set a metadata key/value pair, you are also returned a UUID (universally unique identifier) that represents the intersection of *document*, *key*, and *value*.

You can find all of the metadata keys set for a document like so:

```
xquery version "1.0";

import module namespace md = "http://exist-db.org/metadata";

md:keys(doc("/db/my-docs/doc1.xml"))
```

And you can retrieve the document(s) with the metadata key `roll-number` and the value `12345` like this:

```
xquery version "1.0";

import module namespace md = "http://exist-db.org/metadata";

md:document-by-pair("roll-number", "12345")
```

Other available functions in the `metadata` module include `md:set-value-by-url`, `md:uuid`, `md:uuid-by-url`, `md:document-by-uuid`, `md:keys-by-url`, `md:get-value`, `md:get-value-by-url`, `md:search`, and `md:delete`.

ngram

Description

NGram index–related functions

Namespace

`ngram="http://exist-db.org/xquery/ngram"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*

Class

`org.exist.xquery.modules.ngram.NGramModule`

This module supports using an NGram index in XPath queries. Use of this module is explained in "Using the NGram Indexes" on page 280.

repo

Description

Working with the EXPath repository manager

Namespace

`repo="http://exist-db.org/xquery/repo"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*

Class

`org.exist.xquery.modules.expathrepo.ExpathPackageModule`

This module allows you to manage the local EXPath package repository of your eXist installation. The package repository manages external packages (*.xar* archives), which can include everything from third-party XQuery libraries to full applications or other XML technology functionality.

The module distinguishes between installation and deployment steps. This is because although the installation process is standardized by the EXPath packaging specification, the deployment step is not. It is implementation-defined and specific to eXist-db. Installation will register a package with the EXPath packaging system, but will not copy anything into the database. Deployment will deploy the application into the database as specified by *repo.xml*.

The packaging mechanism and this module are described in "Packaging" on page 227. Background information is available at *http://exist-db.org/exist/apps/doc/repo.xml*.

request

Description

Handling HTTP requests

Namespace

`request="http://exist-db.org/xquery/request"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*; enabled in *$EXIST_HOME/extensions/build.properties*

Class

`org.exist.xquery.functions.request.RequestModule`

This module provides functions to handle an HTTP request in XQuery, like finding out the URI and the parameters. It is described in "The request Extension Module" on page 209.

response

Description

Controlling the HTTP response

Namespace

`response="http://exist-db.org/xquery/response"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*

Class

`org.exist.xquery.functions.response.ResponseModule`

This module allows you to control the HTTP response in XQuery—for instance, by setting headers or streaming binary data. It is described in "The response Extension Module" on page 211.

restxq

Description

Module for eXist's RESTXQ interface

Namespace

`restxq="http://exquery.org/ns/restxq"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*; enabled in *$EXIST_HOME/extensions/build.properties*

Class

`org.exist.extensions.exquery.restxq.impl.xquery.RestXqModule`

This module supports eXist's RESTXQ interface. For more information, see "Building Applications with RESTXQ" on page 215.

restxqex

Description

Module for eXist's RESTXQ resource function registry

Namespace

`restxqex="http://exquery.org/ns/restxq/exist"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*; enabled in *$EXIST_HOME/extensions/build.properties*

Class

`org.exist.extensions.exquery.restxq.impl.xquery.exist.ExistRestXqModule`

This module is specific to eXist and provides some reporting and control over the registration of RESTXQ resource functions with eXist's implementation of the RESTXQ resource function registry. For more information, see "Building Applications with RESTXQ" on page 215.

scheduler

Description

Scheduling jobs

Namespace

`scheduler="http://exist-db.org/xquery/scheduler"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*; enabled in *$EXIST_HOME/extensions/build.properties*

Class

`org.exist.xquery.modules.scheduler.SchedulerModule`

The `scheduler` module allows you to run a job (for instance, an XQuery script) at regular intervals. For instance, starting a script that will run five times at a 10-second interval is done with this call (for details, please refer to the module documentation):

```
    scheduler:schedule-xquery-periodic-job('/path/to/your/script.xq', 10000,
      'JobName', (), 0, 5 )
```

Jobs can be stopped, paused, and examined. Calling `scheduler:get-scheduled-jobs` may surprise you by showing that there are quite a few system jobs running that you weren't aware of.

Jobs scheduled with the `scheduler` module are not persisted across restarts of the database. If you wish to create persistent scheduled jobs, you need to add them to the scheduler configuration section of *$EXIST_HOME/conf.xml* as well.

If your job doesn't seem to run or does not behave as you expect, consult the *$EXIST_HOME/webapp/WEB-INF/logs/exist.log* file. For more information on eXist's scheduler, see "Scheduled Jobs" on page 435.

sequences

Description

Working with sequences

Namespace

`sequences="http://exist-db.org/xquery/sequences"`

Type/default status

XQuery; enabled in *$EXIST_HOME/conf.xml*

Location

resource:org/exist/xquery/lib/sequences.xq

This module includes utility functions for working with sequences like `filter` and `fold`.

session

Description

Functions for working with HTTP sessions

Namespace

`session="http://exist-db.org/xquery/session"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*

Class

`org.exist.xquery.functions.session.SessionModule`

The HTTP session is a piece of memory that is allocated by the Java application server underlying eXist and associated with a series of requests from a single client. So that the Java application server can identify each request from a client as belonging to the same session, a unique identifier is generated and given to the client. This is expected to be sent back to the server on each request. This identifier may take the form of an HTTP cookie or a parameter in the HTTP URL query string; either way, it will be named `JSESSIONID`.

A session can contain data that is kept alive between requests. Access to a session is provided through this module. It is described in "The session Extension Module" on page 211.

sm

Description

Security Manager functions

Namespace

`sm="http://exist-db.org/xquery/securitymanager"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*

Class

`org.exist.xquery.functions.securitymanager.SecurityManagerModule`

This module is for managing eXist's security settings. You can, for instance, check whether you have access to a resource by calling `sm:has-access`. More about security can be found in Chapter 8.

sort

Description

Indexes for efficient sorting

Namespace

`sort="http://exist-db.org/xquery/sort"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*

Class

`org.exist.xquery.modules.sort.SortModule`

The `sort` module lets you create (global, systemwide) indexes to speed up sorting (with `order by` clauses) on nodes in FLWOR expressions. This works only for nodes stored in the database, not nodes in temporary, in-memory XML fragments.

As an example, assume we have an XML document stored in our database that looks like this:

```
<NodeSet>
    <Node>a</Node>
    <Node>B</Node>
    <Node>c</Node>
    <Node>D</Node>
    <Node>e</Node>
    <Node>F</Node>
</NodeSet>
```

Not terribly interesting and not at all difficult or time-consuming to sort, but it serves the purpose of illustrating how a sort index works. Assume we want to sort the `Node` elements based on the uppercase value of their contents, using a sort index. To do this we first have to create a function that converts the node to sort into an atomic value. In this case:

```
declare function local:sort-callback($node as node()) as xs:string {
  upper-case(normalize-space($node))
};
```

Now we can create the index using this function:

```
let $node-set as element()+ := doc('/db/path/to/nodeset/document')/*/Node
let $index-id as xs:string := 'SORTINDEX'
let $sort-index := sort:create-index-callback(
   $index-id,
   $node-set,
   local:sort-callback#1,
   ()
)
```

With this we've created a sort index for our nodes with identifier `SORTINDEX`. The `sort:create-index-callback` always returns the empty sequence `()`. Its fourth argument can be an XML fragment for specifying the sort order and what to do with empty values:

```
<options order? = "ascending" | "descending"
         empty? = "least" | "greatest"
```

Using the created sort index in a FLWOR expression is easy:

```
for $node in $node-set
order by sort:index($index-id, $node)
return
   string($node)
```

This returns the expected:

```
a B c D e F
```

Once created, a sort index is available globally, systemwide. All queries, in any session, can access it. So, if you need sort indexes in your application, it would be best to create them in some kind of initialization script. If you want to check whether a sort index exists, use `sort:has-index`.

sql

Description

Database access using JDBC

Namespace

`sql="http://exist-db.org/xquery/sql"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*; enabled in *$EXIST_HOME/extensions/build.properties*

Class

`org.exist.xquery.modules.sql.SQLModule`

This module allows you to access SQL databases from eXist using the Java Database Connectivity (JDBC) API. Find more information at *http://atomic.exist-db.org/HowTo/SQLDatabases/*.

system

Description

Information about eXist and the system environment

Namespace

`system="http://exist-db.org/xquery/system"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*

Class

`org.exist.xquery.functions.system.SystemModule`

This module contains several utility functions for working with eXist and the system environment. For instance, you can use it to get the memory settings, the eXist version, and a list of running XQueries, and to shut down the database.

text

Description

Text-searching extension functions

Namespace

`text="http://exist-db.org/xquery/text"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*

Class

`org.exist.xquery.functions.text.TextModule`

This module contains several functions for searching in and working with text—for instance, finding all matches to a regular expression. Many of these functions work together with the full-text index.

transform

Description

XSLT-related functions

Namespace

`transform="http://exist-db.org/xquery/transform"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*

Class

`org.exist.xquery.functions.transform.TransformModule`

This module allows you to perform XSLT transformations within your XQuery code. It is described in "XSLT" on page 238.

util

Description

Utility functions

Namespace

`util="http://exist-db.org/xquery/util"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*

Class

`org.exist.xquery.functions.util.UtilModule`

This module comprises a miscellaneous collection of functions for performing all sorts of not easily categorized tasks; for instance, creating UUIDs, dynamically setting XQuery options, and various type conversions.

validation

Description

Validating XML

Namespace

`validation="http://exist-db.org/xquery/validation"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*

Class

`org.exist.xquery.functions.validation.ValidationModule`

This module is for validating XML against DTDs, XML schemas and/or RELAX NG schemas, and Schematron. It is described in "Validation" on page 246.

versioning

Description

Access to versioning extensions

Namespace

`versioning="http://exist-db.org/versioning"`

Type/default status

XQuery; enabled in *$EXIST_HOME/conf.xml*; enabled in *$EXIST_HOME/extensions/build.properties*

Location

resource:org/exist/versioning/xquery/versioning.xqm

This module provides access to eXist's versioning extension. This provides basic versioning capabilities for resources stored in the database. The versioning extension is described in Chapter 16.

xmlcalabash

Description

Access to the XML Calabash XProc module (experimental)

Namespace

`calabash="http://xmlcalabash.com"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*; enabled in *$EXIST_HOME/extensions/build.properties*

Class

`org.exist.xquery.modules.xmlcalabash.XMLCalabashModule`

This is an experimental module for running XProc pipelines with the XML Calabash implementation. It registers a single function called `process`.

This module doesn't follow the normal pattern for Java modules: it does not register a default namespace prefix. You'll have to add `import module namespace xmlc="http://xmlcalabash.com";` at the top of your script (which is good practice anyway) to be able to call `xmlc:process`.

xmldb

Description

Core module for working with the database

Namespace

`xmldb="http://exist-db.org/xquery/xmldb"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*

Class

`org.exist.xquery.functions.xmldb.XMLDBModule`

This module contains all the base functions for working with the database: exploring the database content, creating collections and resources, logging in, and so on. More information is available in "Controlling the Database from Code" on page 107.

xmldiff

Description

Comparing XML documents

Namespace

`xmldiff="http://exist-db.org/xquery/xmldiff"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*; enabled in *$EXIST_HOME/extensions/build.properties*

Class

`org.exist.xquery.modules.xmldiff.XmlDiffModule`

This module allows you to compare two XML documents with the `xmldiff:compare` function. It only returns whether the documents are equal (not a difference list).

xmpp

Description

Access to the XMPP instant messaging protocol

Namespace

`xmpp="http://exist-db.org/xquery/xmpp"`

Type/default status

Java; disabled in *$EXIST_HOME/conf.xml*; disabled in *$EXIST_HOME/extensions/build.properties*

Class

`org.exist.xquery.modules.xmpp.XMPPModule`

This module is an XQuery interface to the Extensible Messaging and Presence Protocol (XMPP) used for instant messaging, chatting, and so on.

xqdm

Description

Retrieving xqDoc information from XQuery modules (XQuery 1.0 only)

Namespace

`xqdm="http://exist-db.org/xquery/xqdoc"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*

Class

`org.exist.xqdoc.xquery.XQDocModule`

This is an older implementation of the XQuery documentation system xqDoc, as described in "XQuery Documentation with xqDoc" on page 125. It only works with pure XQuery 1.0 modules and does not support XQuery 3.0.

xqjson

Description

Module for serializing XML into JSON, and parsing JSON into XML.

Namespace

`xqjson="http://xqilla.sourceforge.net/lib/xqjson"`

Type/default status

XQuery; installable as an EXPath package.

Location

xmldb:///db/system/repo/xqjson-0.1.5/content/xqjson.xql

This module provides a different approach to conversion to/from JSON by introducing a custom XML grammar for representing a parsed JSON document. The two main functions of interest are `xqjson:serializejson` and `xqjson:parse-json`. You can find further details about the module and its XML grammar on the XQJSON GitHub page.

xslfo

Description

Rendering with XSL-FO

Namespace

`xslfo="http://exist-db.org/xquery/xslfo"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*; enabled in *$EXIST_HOME/extensions/build.properties*

Class

`org.exist.xquery.modules.xslfo.XSLFOModule`

This module provides access to eXist's XSL-FO capabilities. This is described in full in "XSL-FO" on page 252.

zip

Description

Accessing resources in ZIP files

Namespace

`zip="http://expath.org/ns/zip"`

Type/default status

Java; enabled in *$EXIST_HOME/conf.xml*

Class

`org.expath.exist.ZipModule`

This module allows you to read resources from ZIP files using the standard EXPath functions for this; for further information, see http://expath.org/spec/zip.

APPENDIX B

REST Server Processes

eXist's REST Server offers a great deal of functionality from a relatively simple API. This chapter tries to illustrate the various process (decision) flows evaluated by the REST Server when you make HTTP requests of it, so that you may better understand their actions. We do this through a series of diagrams linked to the different request types.

We also examine the optional and mandatory parameters that can be used with the REST Server to modify requests and responses.

GET Process Flow

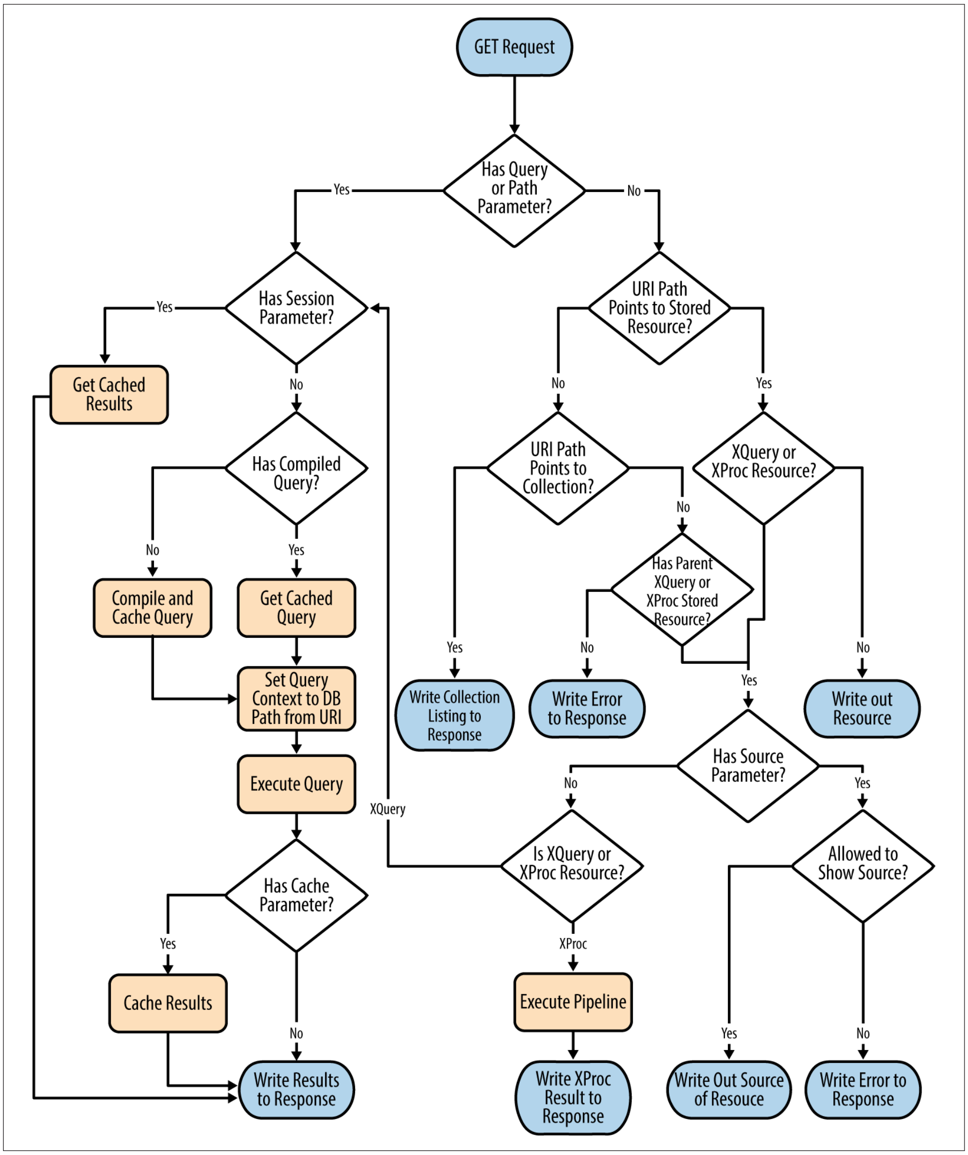

GET Request
Has Query or Path Parameter?
Yes
No
Has Session Parameter?
URI Path Points to Stored Resource?
Get Cached Results
Has Compiled Query?
URI Path Points to Collection?
XQuery or XProc Resource?
Has Parent XQuery or XProc Stored Resource?
Compile and Cache Query
Get Cached Query
Set Query Context to DB Path from URI
Execute Query
Write Collection Listing to Response
Write Error to Response
Write out Resource
Has Source Parameter?
XQuery
Is XQuery or XProc Resource?
Allowed to Show Source?
Has Cache Parameter?
XProc
Cache Results
Execute Pipeline
Write Results to Response
Write XProc Result to Response
Write Out Source of Resouce
Write Error to Response

Figure B-1. Process flow for HTTP GET requests in the REST Server

HEAD Process Flow

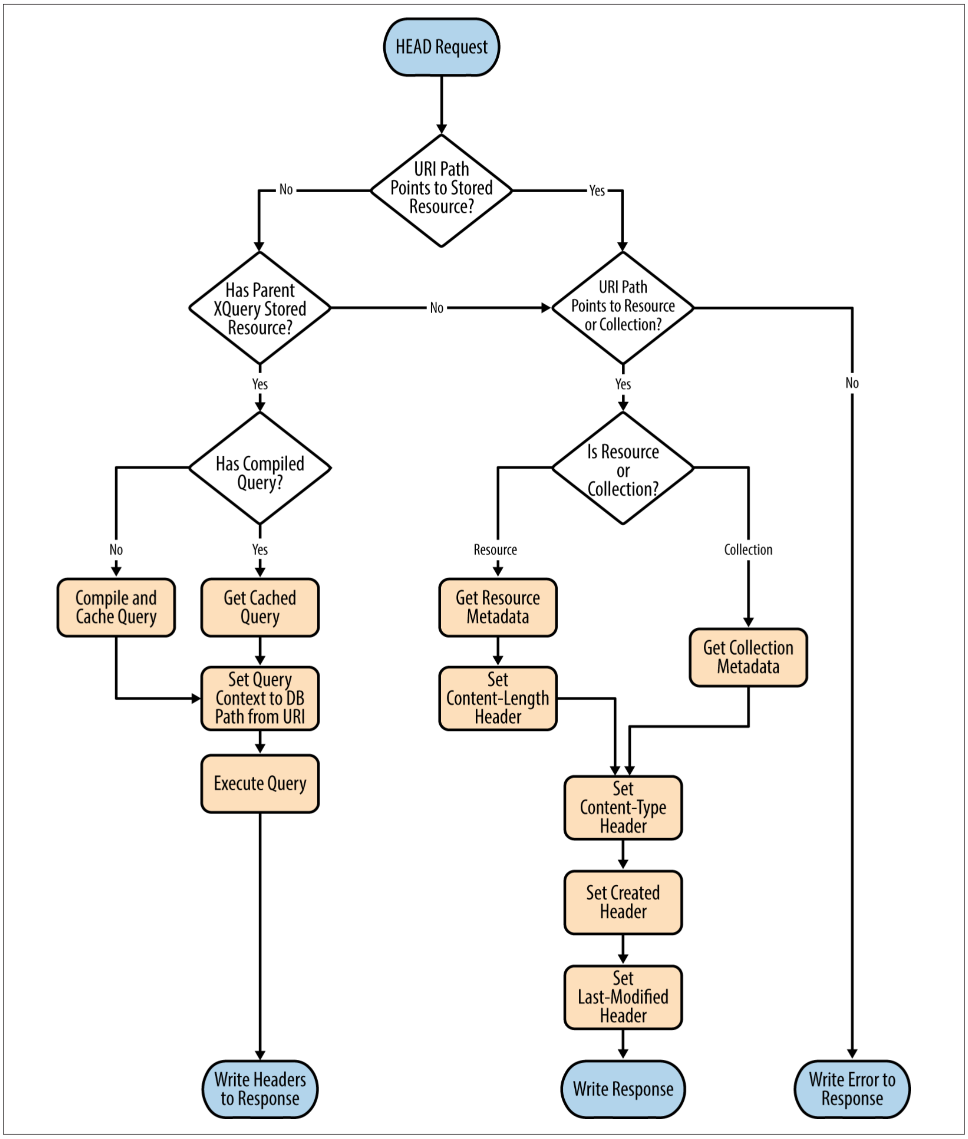


Figure B-2. Process flow for HTTP HEAD requests in the REST Server

PUT Process Flow

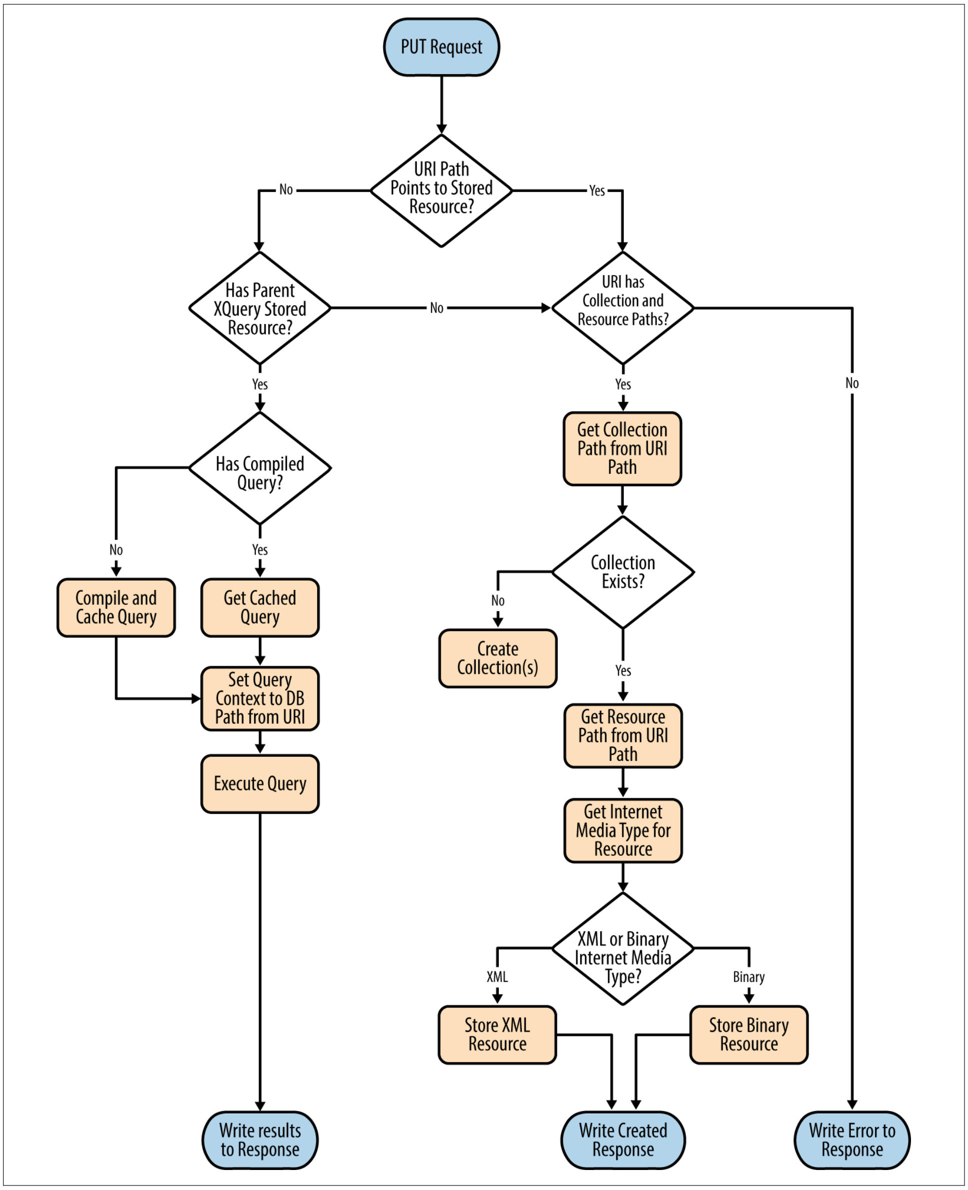

PUT Request
URI Path Points to Stored Resource?
No
Yes
Has Parent XQuery Stored Resource?
No
URI has Collection and Resource Paths?
Yes
Yes
No
Has Compiled Query?
Get Collection Path from URI Path
No
Yes
Collection Exists?
Compile and Cache Query
Get Cached Query
No
Create Collection(s)
Yes
Set Query Context to DB Path from URI
Get Resource Path from URI Path
Execute Query
Get Internet Media Type for Resource
XML or Binary Internet Media Type?
XML
Binary
Store XML Resource
Store Binary Resource
Write results to Response
Write Created Response
Write Error to Response

Figure B-3. Process flow for HTTP PUT requests in the REST Server

DELETE Process Flow

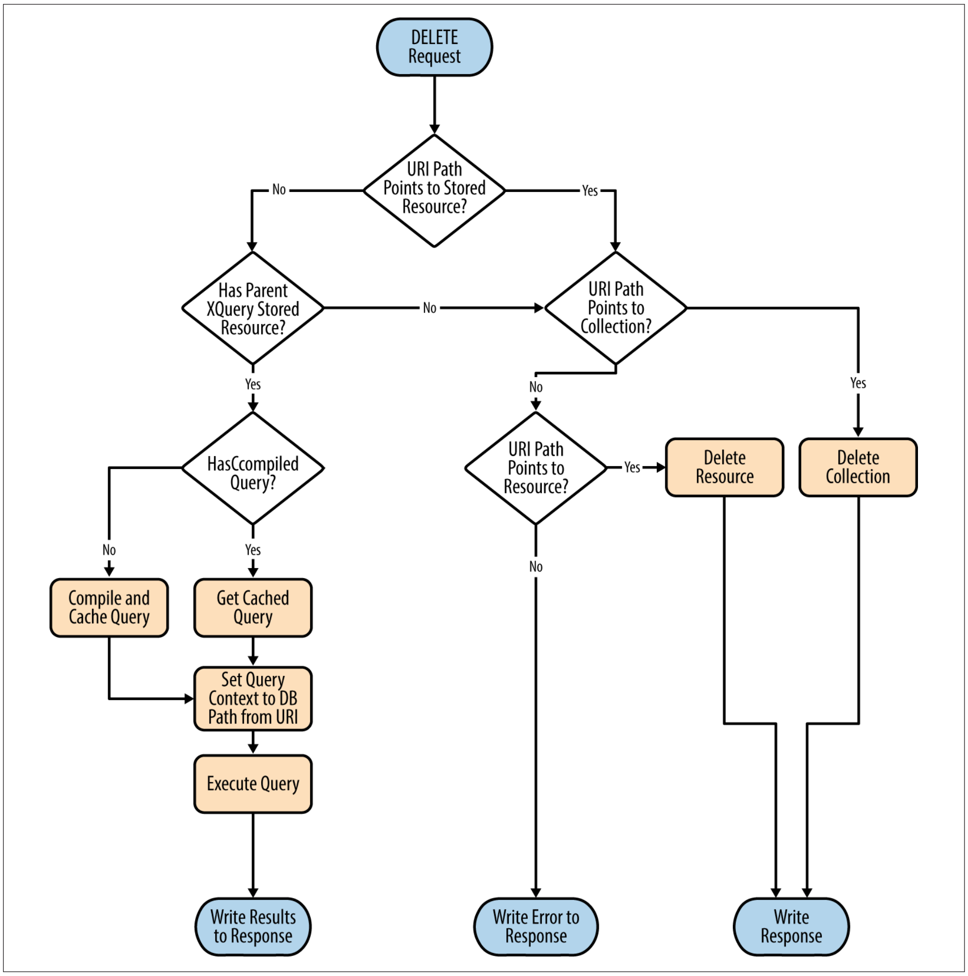

DELETE Request
URI Path Points to Stored Resource?
No
Yes
Has Parent XQuery Stored Resource?
No
URI Path Points to Collection?
Yes
No
Yes
HasCompiled Query?
URI Path Points to Resource?
Yes
Delete Resource
Delete Collection
No
Yes
No
Compile and Cache Query
Get Cached Query
Set Query Context to DB Path from URI
Execute Query
Write Results to Response
Write Error to Response
Write Response

Figure B-4. Process flow for HTTP DELETE requests in the REST Server

POST Process Flow

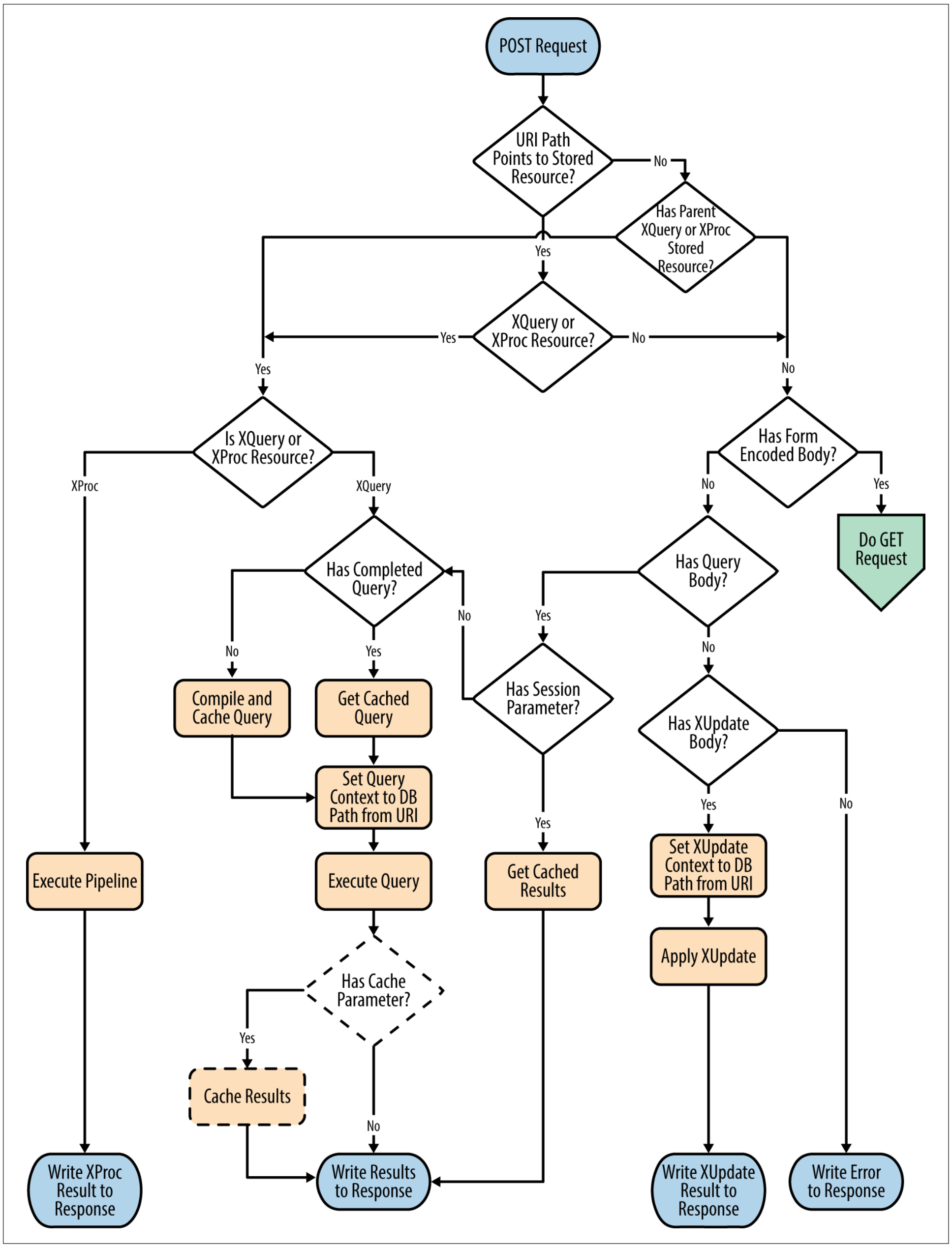


Figure B-5. Process flow for HTTP POST requests in the REST Server

The boxes in the flowchart with dotted outlines ("Has `cache` Parameter?" and "Cache Results") indicate that these steps are only considered when an XQuery is `POST`ed to the REST Server API, and *not* when a stored query is invoked using a `POST` request to the REST Server API.

REST Server Parameters

Next we detail the parameters available for use when you are making requests to eXist's REST Server.

Note that several of the parameters are similar for `GET` and `POST` requests. Parameters are always written lowercase; however, when they are used in the query of a URL for an HTTP `GET` request they are also prefixed with an _ (underscore) character.

HTTP GET Parameters

The parameters detailed here can be used in the query string of a URL when you are performing an HTTP `GET`.

> When placing values into query parameters you may need to URL-encode them, depending on the HTTP client that you are using. Many HTTP clients (such as web browsers) will auto-encode the parameters. If you are using Java's `java.net.URLConnection`, however, you should be aware that it does not auto-encode the parameters, and you will need to encode each value with `java.net.URLEncoder`.
>
> Many client and server applications have a limit on the amount of data that can be placed into the URL query string, so if you are making complex queries you should consider using `POST` instead of `GET`. For more information, see "HTTP POST Parameters" on page 536.

For clarity, the query parameter values in Table B-1 are shown *before* encoding.

Table B-1. HTTP GET URL query parameters

Name	Description	Example
`cache`	Used in combination with a `query` parameter, this parameter can instruct the REST Server to cache the results of the XQuery in RAM. This is especially useful when subsequently performing requests with the `session`, `start`, and `howmany` parameters to enable you to efficiently execute a query once, and then retrieve the results in pages. Once you are finished with a cached query, you should make another request with the `release` parameter set to `yes` to remove its results from memory on the server. Cached results are assigned a *session ID*, which you need to use in subsequent requests. The session ID is returned in the `exist:result/@session` attribute (see "wrap XML grammar" on page 540) if the `wrap` parameter is not set to `no`, and also in the HTTP response header `X-Session-Id`. The value of the parameter should be either `yes` or `no`.	`http://localhost:8080/exist/rest/db/`*`my-collection`*`/?_query=`*`/a/b/c`*`&_cache=`*`yes`*
`encoding`	Can be used to specify the character encoding that should be used by the serializer for the result document. The value of the parameter should be the name of a character encoding supported by your JRE (e.g., `UTF-16`). eXist uses `UTF-8` by default.	`http://localhost:8080/exist/rest/db/`*`my-collection`*`/?_query=`*`//my-node`*`&_encoding=UTF-16`
`howmany`	Used in combination with a `query` parameter to indicate how many results should be returned from the query (if the query returns a sequence of items). This can also be used in combination with `start` to easily create a paging mechanism. The value of the parameter should be an integer value of `1` or more. Using a number greater than the number of available results will simply result in all results being returned.	`http://localhost:8080/exist/rest/db/`*`my-collection`*`/?_query=`*`/a/b/c`*`&_howmany=`*`10`*

Name	Description	Example
`indent`	Instructs the serializer to apply indentation to the formatting of the result document. The value of the parameter should be either `yes` or `no`.	`http://localhost:8080/exist/rest/db/`*`my-collection`*`/?_query//`*`my-node`*`&_indent=`*`yes`*
`query`	Can be used to provide an XQuery to execute. The initial context of the XQuery is the document or collection indicated by the URI. The value of the parameter should be an XQuery.	`http://localhost:8080/exist/rest/db/`*`my-collection`*`/?_query/`*`a/b/c`*
`release`	When the `cache` parameter is set to `yes`, the results of XPath and XQuery executions by the REST Server are cached. This parameter can be used to release the cached results of a query. The value of the parameter should be the cached session ID returned in the result of the initial query.	`http://localhost:8080/exist/rest/db/?_query=//`*`my-node`*`&_cache=`*`yes`*`&_release=`*`7`*
`session`	Can be used to identify the results of a previous query where the `cache` parameter was used. The value of the parameter should be a cached session ID returned by eXist in the response to a previous request made using the `cache` parameter.	`http://localhost:8080/exist/rest/db/`*`my-collection`*`/?_session=`*`7`*`&_start=`*`11`*`&_howmany=`*`10`*
`source`	When addressing a stored XQuery document, this parameter can be set to `yes` to retrieve the source code of the XQuery rather than execute it. By default, this functionality is disabled in the configuration file *$EXIST_HOME/descriptor.xml*, but it can be enabled on a resource-by-resource basis if desired. The value of the parameter should be either `yes` or `no`.	`http://localhost:8080/exist/rest/db/`*`my-collection/my-query.xquery`*`?_source=`*`yes`*

Name	Description	Example
`start`	Used in combination with a `query` parameter to indicate where to start returning results from within the result sequence (if the query returns a sequence of items). This is typically used in combination with `howmany` to create a paging mechanism. The value of the parameter should be an integer value of `1` or more. The result subsequence is calculated from `start => results <= howmany`.	`http://localhost:8080/exist/rest/db/`*`my-collection`*`/?_query=`*`/a/b/c`*`&_start=`*`11`*
`typed`	Used in combination with a `query` parameter to instruct the query serializer to add typing information to the elements of the result document. The value of the parameter should be either `yes` or `no`.	`http://localhost:8080/exist/rest/db/`*`my-collection`*`/?_query=`*`//my-node`*`&_typed=`*`yes`*
`variables`	Can be used to provide one or more external variable values to bind into the XQuery static context, either in combination with a `query` parameter or through directly addressing and executing a stored XQuery. The value of the parameter should be an XML document containing the variables as set out in "variables XML grammar" on page 540. Due to the nature of the parameter being an XML document, it can be much simpler to send this as part of an HTTP `POST` request (see "HTTP POST Parameters" on page 536) rather than a `GET` request.	`http://localhost:8080/exist/rest/db/`*`my-collection/my-query.xquery`*`?_variables=`*`<variables xmlns="http://exist.sourceforge.net/NS/exist" xmlns:sx="http://exist-db.org/xquery/types/serialized"><variable><qname><localname>my-var</localname></qname><sx:sequence><sx:value type="xs:string">some value</sx:value></sx:sequence></variable></variables>`*
`wrap`	When sending queries to the REST Server by using the `query` parameter, by default the result of the XQuery will be wrapped in an XML document, as described in "wrap XML grammar" on page 540. This parameter allows you to disable that wrapping. When directly addressing and executing stored XQueries, this parameter has no effect. The value of the parameter should be either `yes` or `no`.	`http://localhost:8080/exist/rest/db/`*`my-collection`*`/?_query=`*`//my-node`*`&_wrap=`*`no`*

Name	Description	Example
`xpath`	Can be used in the query string of a URL to provide an XPath to execute. The context of the XPath is the document or collection indicated by the URI. The value of the parameter should be an XPath expression. This is deprecated in favor of the `query` parameter, which can accept either an XPath or an XQuery.	`http://localhost:8080/exist/rest/db/`*`my-collection`*`/?_xpath=//`*`my-node`*
`xsl`	Can be used to enable/disable the processing of XSL processing instructions in the result document by the serializer, or to apply a specific XSLT transformation to the result document. The value of the parameter should be either `yes`, `no`, or a path to an XSLT document in the database. When using to enable processing of XSL processing instructions, note that this needs to also be enabled in the configuration file *$EXIST_HOME/conf.xml*.	`http://localhost:8080/exist/rest/db/`*`my-collection`*`/?_query=//`*`my-node`*`&_xsl=`*`/db/my-stylesheet.xslt`*

HTTP POST Parameters

When making an HTTP POST request to eXist's REST Server, you can send an XML document in the body describing an XQuery to be executed and parameters to control the execution and serialization of results. In many ways this is very similar to the process described in the previous section, but it uses POST and an XML document as opposed to GET and URL query parameters.

The XML grammar describing the XML document that can be sent in the body of an HTTP POST to the REST Server is as follows:

```
<exist:query xmlns:exist="http://exist.sourceforge.net/NS/exist
    start? = number
    max? = number
    cache? = ("yes" | "no")
    session? = string
    typed? = ("yes" | "no")
    (wrap = ("yes" | "no") | enclose = ("yes" | "no"))?
    encoding? = string
    method? = string>
        (exist:text,
        exist:variables?,
        exist:properties?)
</exist:query>
```

The attributes on the exist:query element are documented in Table B-2, while the exist:text, exist:variables, and exist:properties elements are documented further in "text XML grammar" on page 540, "variables XML grammar" on page 540, and "properties XML grammar" on page 540, respectively.

Table B-2. HTTP POST request body parameters

Name	Description	Example

`cache`	Can be used to instruct the REST Server to cache the results of the XQuery in RAM. This is especially useful when subsequently performing requests with the `session`, `start`, and `max` parameters to enable you to efficiently execute a query once, and then retrieve the results in pages.
	
	Cached results are assigned a *session ID*, which you need to use in subsequent requests. The session ID is returned in the `exist:result/@session` attribute (see "wrap XML grammar" on page 540) if the `wrap` parameter is not set to `no`, and also in the HTTP response header `X-Session-Id`.
	
	The value of the parameter should be either `yes` or `no`.
		`<exist:query xmlns:exist="http://exist.sourceforge.net/NS/exist"`
		    `cache="yes">`
		        `<exist:text><![CDATA[`
		            `/a/b/c`
		        `]]></exist:text>`
		`</exist:query>`

`enclose`	When sending queries to the REST Server, by default the result of the XQuery will be wrapped in an XML document, as described in "wrap XML grammar" on page 540. This parameter allows you to disable that wrapping. When directly addressing and executing stored XQueries, this parameter has no effect.
	
	The value of the parameter should be either `yes` or `no`.
	
	This is deprecated in favor of the `wrap` parameter.
		`<exist:query xmlns:exist="http://exist.sourceforge.net/NS/exist"`
		    `enclose="no">`
		        `<exist:text><![CDATA[`
		            `//my-node`
		        `]]></exist:text>`
		`</exist:query>`

`encoding`	Can be used to specify the character encoding that should be used by the serializer for the result document.
	
	The value of the parameter should be the name of a character encoding supported by your JRE (e.g., `UTF-16`). eXist uses `UTF-8` by default.
		`<exist:query xmlns:exist="http://exist.sourceforge.net/NS/exist"`
		    `encoding="UTF-16">`
		        `<exist:text><![CDATA[`
		            `//my-node`
		        `]]></exist:text>`
		`</exist:query>`

Name	Description	Example

`max`	Indicates how many results should be returned from the query (if the query returns a sequence of items). This can also be used in combination with `start` to easily create a paging mechanism.
	
	The value of the parameter should be an integer value of `1` or more. Using a number greater than the number of available results will simply result in all results being returned.
		`<exist:query xmlns:exist="http://exist.sourceforge.net/NS/exist"`
		`    max="10">`
		`        <exist:text><![CDATA[`
		`            /a/b/c`
		`        ]]></exist:text>`
		`</exist:query>`

`method`	Can be used in the body of a `POST` request when supplying an XQuery; it causes the results of the query to be serialized using the supplied serializer method.
	
	Instructs the serializer to use a specific serialization method on the results of the query. For the JSON serialization format of XML used by eXist, see "JSON serialization" on page 121.
	
	The value of the parameter must be either `xml`, `xhtml`, `html`, `html5`, `json`, or `text`.
		`<exist:query xmlns:exist="http://exist.sourceforge.net/NS/exist"`
		`    method="json">`
		`        <exist:text><![CDATA[`
		`            /a/b/c`
		`        ]]></exist:text>`
		`</exist:query>`

`session`	Can be used to identify the results of a previous query where the `cache` parameter was used.
	
	The value of the parameter should be a cached session ID returned by eXist in the response to a previous request made using the `cache` parameter.
		`<exist:query xmlns:exist="http://exist.sourceforge.net/NS/exist"`
		`    session="7"`
		`    start="11"`
		`    max="10">`
		`        <exist:text><![CDATA[`
		`            /a/b/c`
		`        ]]></exist:text>`
		`</exist:query>`

Name	Description	Example

`start`	Indicates where to start returning results from within the result sequence (if the query returns a sequence of items). This is typically used in combination with `max` to create a paging mechanism.
	
	The value of the parameter should be an integer value of `1` or more. The result subsequence is calculated from `start => results <= max`.
		`<exist:query xmlns:exist="http://exist.sourceforge.net/NS/exist"`
		`    start="11">`
		`        <exist:text><![CDATA[`
		`            /a/b/c`
		`        ]]></exist:text>`
		`</exist:query>`

`typed`	Instructs the query serializer to add typing information to the elements of the result document.
	
	The value of the parameter should be either `yes` or `no`.
		`<exist:query xmlns:exist="http://exist.sourceforge.net/NS/exist"`
		`    typed="yes">`
		`        <exist:text><![CDATA[`
		`            //my-node`
		`        ]]></exist:text>`
		`</exist:query>`

`wrap`	When sending queries to the REST Server, by default the result of the XQuery will be wrapped in an XML document, as described in "wrap XML grammar" on page 540. This parameter allows you to disable that wrapping. When directly addressing and executing stored XQueries, this parameter has no effect.
	
	The value of the parameter should be either `yes` or `no`.
		`<exist:query xmlns:exist="http://exist.sourceforge.net/NS/exist"`
		`    wrap="no">`
		`        <exist:text><![CDATA[`
		`            //my-node`
		`        ]]></exist:text>`
		`</exist:query>`

Common XML Grammars for Parameters

This section details some of the common XML grammars used in GET and POST operations.

wrap XML grammar

The XML grammar describing the result document returned by the REST Server when the wrap request parameter is not set to no is formatted as follows:

```
<exist:result xmlns:exist="http://exist.sourceforge.net/NS/exist"
    hits = number
    start = number
    count = number
    session? = string>
        any*
</exist:result>
```

properties XML grammar

The XML grammar for the properties request parameter has this format:

```
<exist:properties>
    (exist:property+)
</exist:properties>

<exist:property
    name = string
    value = string/>
```

text XML grammar

The XML grammar for the text request parameter has this format:

```
<exist:text><![CDATA[

    (: Your XQuery code here! :)

]]></exist:text>
```

variables XML grammar

The XML grammar for the variables request parameter has the following format, and you can see it in context in Example B-1:

```
<exist:variables xmlns:exist="http://exist.sourceforge.net/NS/exist">
    (exist:variable+)
</exist:variables>

<exist:variable xmlns:sx="http://exist-db.org/xquery/types/serialized">
    (exist:qname,
    sx:sequence)
</exist:variable>
```

```
    <exist:qname>
        (exist:prefix?,
        exist:localname,
        exist:namespace?)
    </exist:qname>

    <sx:sequence>
        (sx:value+)
    </sx:sequence>

    <sx:value type? = string>
        (text() | element())
    </sx:value>
```

Example B-1. variables XML document

```
<variables xmlns="http://exist.sourceforge.net/NS/exist"
  xmlns:sx="http://exist-db.org/ xquery/types/serialized">
    <variable>
        <qname>
            <localname>my-var</localname>
        </qname>
        <sx:sequence>
            <sx:value type="xs:string">some value</sx:value>
        </sx:sequence>
    </variable>
    <variable>
        <qname>
            <localname>author</localname>
            <namespace>http://book/example/</namespace>
        </qname>
        <sx:sequence>
            <sx:value>
                <firstName xmlns="">Adam</firstName>
            </sx:value>
            <sx:value>
                <lastName xmlns="">Retter</lastName>
            </sx:value>
        </sx:sequence>
    </variable>
</variables>
```

Index

About the Authors

Erik Siegel runs Xatapult, a consultancy that specializes in content engineering and the application of XML: strategic use, standards, design, processing platforms, applications, transformations, and training. Xatapult supports companies that either lack the knowledge or the capacity in these fields of expertise.

Adam Retter is the Director of Evolved Binary Ltd and a cofounder of eXist Solutions GmbH. Since 2005, Adam has been a core developer on the eXist NoSQL Application Platform. Passionate about the Web, code quality, standards, and portable code, Adam has been promoting XQuery as a web application development language. Adam has over 11 years of commercial experience in Java application development and data processing. Naturally fascinated by new technologies, for the last few years Adam has been developing in Scala and Akka. Adam is a member of the XML Guild, an Invited Expert to the W3C XML Query and CSV on the Web working groups, and on the program committee of the XML Prague, XML London, Balisage, and XML Summer School conferences.

Colophon

The animal on the cover of *eXist* is a lettered araçari (*Pteroglossus inscriptus*), a species of toucan that lives in swampy regions and forests around South America. It has been recorded at heights of approximately 4,000 feet in the Andean foothills, although lowland forest (even heavily disturbed rainforest) seems to be its preferred habitat.

The lettered araçari is only about a foot in length and weighs in at approximately one quarter of a pound. Its large, colorful bill helps members of the species recognize each other for mating purposes. It also allows the lettered araçari to grab and eat fruit, insects, and small birds such as finches. Birds of both genders have bodies that are green on top with striking yellow undersides, red rear ends, golden bills, and blue patches surrounding dark eyes.

A social animal, the lettered araçari roosts in groups throughout the year. It nests in cavities and sleeps with its tail folded over its back, along with up to five other adults and their offspring.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from Johnson's *Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.