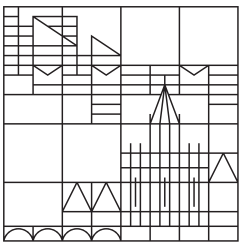
Input and Output with XQuery and XML Databases

Rositsa Shadura

Master Thesis in fulfillment of the requirements for the degree of
Master of Science (M.Sc.)

Submitted to the Department of Computer and Information Science at the
University of Konstanz

Universität
Konstanz



1st Referee: Prof. Dr. Marc H. Scholl
2nd Referee: Prof. Dr. Marcel Waldvogel
Supervisor: Dr. Christian Grün

Abstract

XML and XQuery provide a convenient way to model, store, retrieve and process data. As a result, XML databases have become more and more popular in recent years. Consequently, their usage scenarios have gone far beyond handling XML exclusively. This thesis focuses on the challenges which emerge from unifying the input and output processing of data in XML databases. Based on the analysis of use cases and existing solutions, we define several requirements which shall be met for generalized data processing. Following those we introduce a generic framework, which can serve as a blueprint when designing the input and output data flow in an XML database. Furthermore, we propose a solution how this framework can be applied in an existing open source XML database, named BaseX, in order to improve its current approach of data processing.

Zusammenfassung

Die flexible und standardbasierte Modellierung, Speicherung, Abfrage und Verarbeitung von semistrukturierten Daten sind die häufigsten Beweggründe für den Einsatz von XML Technologien. Stetig wachsende Datenmengen steigern dabei nicht nur die Popularität von XML Datenbanken, sondern stellen neue Voraussetzungen für die Implementierungen: viele Anwendungen erfordern mehr als *nur die Speicherung* von Daten die originär in XML vorliegen. Diese Arbeit untersucht die Herausforderungen bei der Umsetzung von einheitlichen Ein- und Ausgabeschnittstellen in XML Datenbanken. Nach einer Analyse bestehender Implementierungen und verschiedener Anwendungsszenarien stellen wir Anforderungen an Im- und Exportschnittstellen fest. Basierend auf diesen Überlegungen definieren wir ein generisches Framework zur Implementierung von Ein- und Ausgabe in XML Datenbanken. Schließlich stellen wir aus, wie man das Framework in BaseX, einem Open Source XML Datenbankmanagementsystem, umsetzen kann.

Acknowledgements

First of all, I would like to thank Prof. Dr. Marc H. Scholl and Prof. Dr. Marcel Waldvogel for being my referees and giving me the opportunity to work on this topic.

I am truly grateful to Dr. Christian Grün for advising me not only on the writing of this thesis but throughout the whole process of my studies at the University of Konstanz. I think that being part of the BaseX team is great. Thank you!

Special thanks I owe to Alexander Holupirek, Dimitar Popov, Leonard Wörteler, Lukas Kircher and Michael Seiferle for the numerous discussions we had around BaseX and for being such good friends!

Last but not least, I want to thank my family for the understanding and support they have always given me.

Contents

1. Introduction

1.1. Motivation

Simple, general and usable over the Internet – these were the main goals the W3C working group set while designing the first XML specification back in 1998. Since then, XML has proven undoubtedly to possess these features but what is more significant – made its way from a widely accepted data exchange format to the world of databases – as a slowly but triumphantly emerging database format.
Why XML databases when there are the good old known relational databases? Well, if we look around, we can observe that actually quite a small portion of the existing data can be represented directly in rows and columns. The majority of it is unstructured and unformed; thus difficult to put into a regular "shape". What XML gives is flexibility, self-description, schema freedom – qualities which make it the better choice for storing such data.
However, what we care about at the end of the day is not how our data is represented or stored on the disk but the information that stays behind it. We need to do something with it, process it, change it, manipulate it. When XML is our data format, XQuery is our friend in need. From a language designed for querying XML, in the last few years it has evolved into a very powerful programming language. This progress XQuery owes to its processing model[BCF+], which specifies that XQuery expressions operate on instances of the XDM data model[XDM] and such instances can be generated from any kind of data source. Thanks to this flexibility, it is able to work not only with XML data but with any kind of data. Furthermore, XQuery is constantly extended with additional features which go beyond XML query processing[BBB+09].
All these aspects make the usage of XML databases and XQuery processors in various data processing applications more and more attractive. This adds a whole new set of requirements to them such as: support of different ways to access the stored data, ability to work with heterogeneous data sources, which may provide also non-XML data, user-friendly interfaces to interact with the database and processor. The fulfillment of these needs raises questions about the input and output with XQuery and XML databases – how shall they be organized; what is the best way to implement them; what kind of ways do exist to store both XML and non-XML data. The answers to these questions stay in the focus of this thesis.

1.2. Overview

This master thesis is organized as follows: in Chapter 2 we define three major use cases for input and output in an XML database along with the actors associated with them. Chapter 3 analyzes several existing XML databases and XQuery processors with respect to the input and output formats they support and the data channels they provide. Chapter 4 presents the central work of the thesis – a generic framework for input and output of data in an XML database. Chapter 5 describes how this framework can be integrated into BaseX and especially how data processing will profit from this foundation in the future. Chapter 6 discusses some possible enhancements for the proposed framework. Finally, Chapter 7 concludes the thesis.

2. Use Cases

The foundation of every good software solution is a detailed analysis of the use cases in which it can participate. Such an analysis always gives a convenient overview of *who* and *what* will interact with the system and in which kind of way.
The topic about input and output with XQuery and XML databases sounds quite a broad one and this is why it would be useful to start with discussing several use cases and the requirements associated with each. These will serve as guidelines for finding an appropriate solution for input and output architecture.

2.1. Actors

We start by defining three main types of users who may interact with an XML database:

- Regular User
  This actor usually communicates with the system through some kind of user interface – it can be a graphical one or just a command line. He/she does not need to be familiar with the architecture and implementation of the system as well as to be acquainted with XQuery and XML. Their everyday interaction with the database includes adding, deleting, searching and eventually modifying documents.

- Application Developer
  This actor uses the system for the same purposes as the regular user but the way he/she communicates with it differs. In this case the actor develops applications which interact with the XML database or XQuery processor through APIs provided by the development team or directly through XQuery. He/she is familiar with XML and XQuery though in-depth knowledge of the system's architecture and implementation is not needed.

- XML database developer
  This actor is part of the team implementing the XML database and XQuery processor. He/she is acquainted in details regarding what goes on behind the scenes. Their tasks include the development of new channels for input and output (new APIs for example) as well as adding support for new data sources. This actor is not

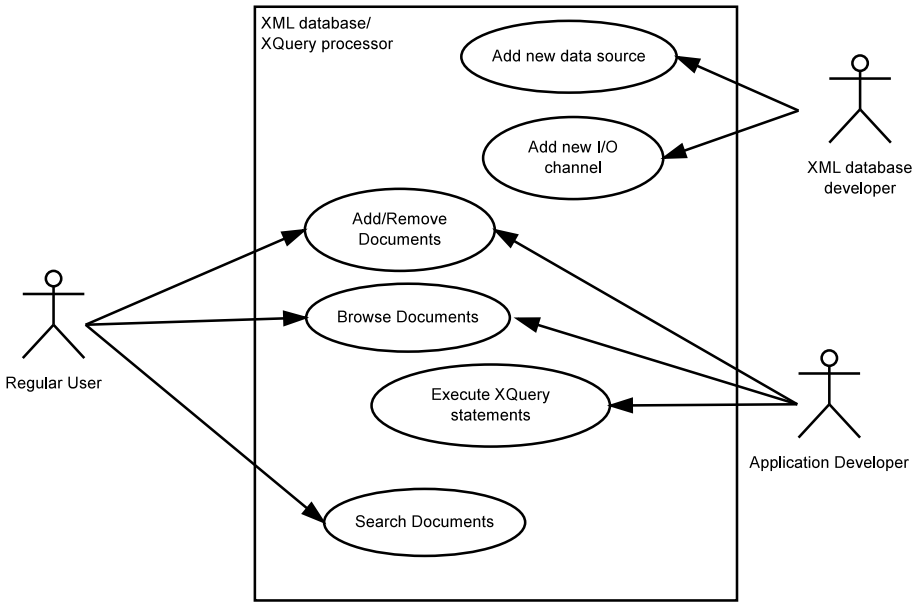


Figure 2.1.: Use Case Diagram

directly involved in the input and output of data in the system but their role is important because it determines the complexity of the whole system and influences the way the other two actors communicate with it.

Having these three types of XML database users we can define three major use cases and the corresponding requirements for them. They are illustrated in the following three sections.

2.2. Storing and querying document-centric documents

This is probably the most popular use case for XML databases. Two fields in which it is often put into practice are publishing and content management. A detailed list with examples from the real world can be found in [Boua].
Document-centric documents are (usually) documents that are designed for human consumption. Examples are books, email, advertisements, and almost any hand-written XHTML document. They are commonly written by hand in XML or some other format, such as RTF, PDF, or SGML.[Boub] This is why one of the major requirements to an XML database in order to be applicable in this use case is to support also non-XML formats both as input and as output. This means that a user shall be able to add documents in various formats to his/her database and retrieve them in the same or even other format. Furthermore, this shall happen transparently meaning that he/she must not take care for

any transformations or conversions. Another requirement, which comes from the fact that the common actors in this use case are regular users, is the presence of a convenient user interface to interact with the system.

2.3. Application Development

This use case acquires more and more importance due to the wide spread usage of XML and the growing popularity of XQuery as a processing language. Here we can talk about two main types of applications that can be developed.
The first ones are those that communicate with the system through some kind of APIs like XML:DB, XQJ, REST or an API specific for the used database. This communication can happen either locally, in which case the database is most probably embedded in the application, or remotely – using HTTP or some other protocol supported by the system, e.g. XML:RPC.
The second type are the applications developed entirely in XQuery. They may either manipulate the data stored in the database and use it for some purpose or may receive data coming from external data sources and store it or just process it. The extension of XQuery with additional modules like these for sending HTTP requests[HTT], file system operations[FIL] and querying relational databases[SQL], makes this possible. Five practical scenarios for XQuery applications are listed in [XQA].
In order to be applicable in a greater variety of programs, an XML database shall offer APIs which are capable of handling non-XML data, too. The same requirement is true also for the XQuery functionality covered by the XQuery processor. It shall not be restricted to the standard XQuery functions[XQF] but must include also such for interaction with the database and for collecting and processing different kinds of data.

2.4. Extending the Input and Output Functionality

The actors in this use case are the developers of the XML database. As already mentioned they are not direct participants in the input and output process but those who are supposed to extend the system with support of new data sources and formats – something that determines how useful and convenient it is for the other two actors to work with it.
The requirements here are related to the implementation of the input and output of data in the XML database. First, it has to be central, which means that the same functionality for parsing, for example, can be reused by all data channels – APIs, XQuery, GUI, etc. In order this to be possible a second requirement has to be fulfilled namely the one for strong decoupling from the rest of the functionality in the system. In other words the

logic which is responsible for the input and output must do only what it is supposed to do – convert data to the XML internal representation specific for the database or vice versa. It must not interfere or depend on other components of the system. Meeting these two requirements will save a lot of work to the developers when adding support for new kinds of data, and on the other hand, make the whole system far more flexible.

3. Existing Solutions

Apart from analyzing the use cases for achieving a certain goal, it is always practical to see what kinds of solutions do already exist in the same direction. In this chapter we present the results of a short investigation on the input and output features of some native XML databases and XQuery processors. Among the aspects which stay in focus are the data channels supported by these systems and the kinds of data which can be used with them.
As a preliminary to the research it should be noted that generally there are two possible ways to store a resource in a native XML database – either as XML using the database-specific internal representation or directly as raw data. Since we are more interested in the first case, whenever we consider the input and output supported by a given data channel, we will be looking at the data formats which can be handled by it and stored or converted to XML and vice versa. Though, if binary data can be handled, too, this will be denoted.

3.1. Qizx

Qizx is an embeddable engine which allows storing and indexing XML documents. It can be directly integrated in a standalone Java application, or it can be the core of a server[QIZa, QIZb]. The analysis in this section is done with version 4.4 of the free engine edition of Qizx.
Since it was designed to be used as an embedded database, Qizx offers an API which lies at the heart of all channels for data input and output. Hence, it is not surprising that the API itself is the best approach for data processing. Apart from XML HTML, JSON and raw input is supported, too. When an application developer wants to load data in some of these formats, they can use the corresponding `ContentImporter` class. For example:

```
// Create an HTML Importer
final HTMLImporter htmlImp = new HTMLImporter();
// Read HTML input
final FileInputStream input = new FileInputStream(pathToHtml);
// Set HTML input
htmlImp.setInput(input);
// Import HTML file to a library
```

```
lib.importDocument(pathInLib, htmlImp);
lib.commit();
```

Along with the standard serialization methods[XQS] one can also output JSON and raw data. This functionality, except the support of binary data input and output, is exported to XQuery extension functions. For instance, the following XQuery code snippet[QIZb]:

```
x:content-parse('{ "a" : 1, b:[true, "str", {}], nothing:null}', "json")
```

produces:

```
<?xml version='1.0'?>
<map xmlns="com.qizx.json">
  <pair name="a">
    <number>1.0</number>
  </pair>
  <pair name="b">
    <array>
      <boolean>true</boolean>
      <string>str</string>
      <map/>
    </array>
  </pair>
  <pair name="nothing">
    <null/>
  </pair>
</map>
```

Other ways for data import and export in Qizx are provided by the graphical user interface, the command line tool and the REST API. However, although they internally use the Qizx API, the range of data formats covered by them is more limited. For instance, a user cannot import HTML or JSON through the GUI. Table 3.1 gives an overview of the input and output channels offered by Qizx and the kinds of data which can flow through them.

	XQuery	GUI	Command Line	Qizx API	REST API
HTML	✓/✓	-/✓	-/✓	✓/✓	-/✓
JSON	✓/✓	-/-	-/-	✓/✓	-/✓
Text	-/✓	-/✓	-/✓	✓/✓	-/✓
binary formats	-/-	✓/✓	-/-	✓/✓	✓/✓

Table 3.1.: Qizx 4.4: Input/Output

3.2. eXist-db

eXist-db is an open-source database management system written in Java. It stores XML instances according to the XML data model and features efficient, index-based XQuery processing. Out of the box, eXist runs inside a web application served by a pre-configured Jetty server[EXIb]. The analysis in this section is done with eXist Tech Preview 2.0.
eXist provides various ways for data input and output. It offers XML:DB, REST, SOAP, XML-RPC and WebDAV APIs. No matter which of these APIs is used, the data that comes through it is always stored, and possibly converted beforehand, depending on what is defined for its content type in a central XML configuration file. Consequently all XML-based formats, e.g. xsd, wsdl, gml, nvdl, application/xml, image/svg+xml, etc. are stored as XML and the remainder is treated as binary. As far as the output is concerned, all APIs except for SOAP and WebDAV support in addition to the standard serialization methods, JSON and HTML5. eXist's XQuery implementation allows working with non-XML data, too. There are extension functions for HTML and CSS parsing and such for executing XSLT transformations and XSL-FO processing. Furthermore, one feature, which is still in development, is a module for content extraction based on Apache's Tika[1]. It offers three XQuery functions – one for metadata extraction from a resource, one for both metadata and content extraction and one which is a streaming variant of the other two[EXIa]. All functions produce XHTML. The following example illustrates how using this module we can extract the metadata from a sample PNG file:

```
import module namespace c="http://exist-db.org/xquery/contentextraction"
  at "java:org.exist.contentextraction.xquery.ContentExtractionModule";

let $path := "/db/test/samplePNG.png"
let $binary := util:binary-doc($path)
return c:get-metadata($binary)
```

This will return:

```
<html>
  <head>
    <meta name="Compression Lossless" content="true"/>
    <meta name="Dimension PixelAspectRatio" content="1.0"/>
    <meta name="iCCP" content="profileName=Photoshop ICC profile,
      compressionMethod=deflate"/>
    <meta name="tiff:ImageLength" content="1427"/>
    <meta name="height" content="1427"/>
    <meta name="pHYs" content="pixelsPerUnitXAxis=11811,
```

[1] A content extraction framework based on Java

```
      pixelsPerUnitYAxis=11811, unitSpecifier=meter"/>
    <meta name="tiff:ImageWidth" content="2846"/>
    <meta name="Chroma BlackIsZero" content="true"/>
    <meta name="Data BitsPerSample" content="8 8 8"/>
    <meta name="Dimension VerticalPixelSize" content="0.08466683"/>
    <meta name="tiff:BitsPerSample" content="8 8 8"/>
    <meta name="width" content="2846"/>
    <meta name="Dimension ImageOrientation" content="Normal"/>
    <meta name="Chroma Gamma" content="0.45453998"/>
    <meta name="Compression CompressionTypeName" content="deflate"/>
    <meta name="cHRM" content="whitePointX=31269, whitePointY=32899,
      redX=63999,redY=33001, greenX=30000, greenY=60000, blueX=15000,
      blueY=5999"/>
    <meta name="Data SampleFormat" content="UnsignedIntegral"/>
    <meta name="Dimension HorizontalPixelSize" content="0.08466683"/>
    <meta name="Transparency Alpha" content="none"/>
    <meta name="Chroma NumChannels" content="3"/>
    <meta name="Compression NumProgressiveScans" content="1"/>
    <meta name="Chroma ColorSpaceType" content="RGB"/>
    <meta name="IHDR" content="width=2846, height=1427,
      bitDepth=8, colorType=RGB, compressionMethod=deflate,
      filterMethod=adaptive, interlaceMethod=none"/>
    <meta name="Data PlanarConfiguration" content="PixelInterleaved"/>
    <meta name="gAMA" content="45454"/>
    <meta name="Content-Type" content="image/png"/>
    <title/>
  </head>
</html>
```

Since Tika is capable of handling content and metadata from a wide range of formats – PDF, Microsoft Office and Open Document, various image and audio formats, etc. this module will contribute to eXist's XQuery input functionality a lot. Apart from the above listed ways for data input and output, there is also a Java-based admin client, which is able to import and export XML and binary data from a database. The following table shows the data channels present in eXist 2.0 along with some of the input and output formats supported by them:

	XQuery	REST	XML:DB	WebDAV	SOAP	XML-RPC	Admin Client
HTML	✓/✓	-/✓	-/✓	-/-	-/-	-/✓	-/-
HTML5	-/✓	-/✓	-/✓	-/-	-/-	-/✓	-/-
Text	-/✓	-/✓	-/✓	-/-	-/-	-/✓	-/-
JSON	-/✓	-/✓	-/✓	-/-	-/-	-/✓	-/-
CSS	✓/-	-/-	-/-	-/-	-/-	-/-	-/-
MS Office formats	✓/-	-/-	-/-	-/-	-/-	-/-	-/-
OO formats	✓/-	-/-	-/-	-/-	-/-	-/-	-/-
PDF	✓/-	-/-	-/-	-/-	-/-	-/-	-/-
EPUB	✓/-	-/-	-/-	-/-	-/-	-/-	-/-
binary formats	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓

Table 3.2.: eXist 2.0: Input/Output

3.3. MarkLogic

MarkLogic is a commercial XML database developed in C++, which is able to handle "Big Data" and unstructured information. The following analysis is done with version 5.0.2 of MarkLogic Server Standard Edition.
MarkLogic was designed to meet the needs of a wide range of customers – from the media and public sector to healthcare and financial services. It is primarily used as a content-repository and this is why it is able to work with a great variety of data formats. Apart from that, it offers diverse ways to make use of the data it stores. From a user's perspective, MarkLogic offers a browser based Information Studio. It allows quick and straightforward creation of databases and loading of documents in them. Using it one can collect content from different data sources, process it with XSLT and built-in transformation logic, and subsequently import it into a database[MLI]. Other input and output channels offerred by MarkLogic are its own specific API – XCC, a rich XQuery implementation and support for various WebDAV clients. Apart from these, there is also a command line tool, which was developed as a community project.
Every document in a MarkLogic Server database has a format associated with it. The format is based on the root node of the document and can be XML, Binary or Text(CLOB)[MLA]. The documents which enter a database through the various channels, as decribed above, are stored depending on the mime types configuration associated with the database. This configuration is central and contains a mapping between a mime type and the format in which it must converted before being stored. Users can customize the mapping according to their needs. This mapping will be applied on any incoming data, no matter which way is used for its input – API, UI, XQuery.
Obviously the format that allows XQuery to perform best is XML, yet not every input format can be processed with pre built transformation scenarios. For this purpose, MarkLogic provides its content processing framework. In short this is a framework consisting of two main types of components – domains and pipelines. The domains define groups of documents which are similar and thus are supposed to be processed in a common way. Pipelines are the means through which the documents in a domain are processed. They consist of conditions and actions which themselves are either XQuery or XSLT scripts. The following example shows a sample pipeline for HTML conversion[MLC]:

```
<?xml-stylesheet href="/cpf/pipelines.css" type="text/css"?>
<pipeline xmlns="http://marklogic.com/cpf/pipelines"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
  "http://marklogic.com/cpf/pipelines pipelines.xsd">
  <pipeline-name>HTML Conversion</pipeline-name>
  <success-action>
    <module>/MarkLogic/cpf/actions/success-action.xqy</module>
  </success-action>
  <failure-action>
    <module>/MarkLogic/cpf/actions/failure-action.xqy</module>
  </failure-action>
```

```
<state-transition>
  <annotation> Convert HTML documents and only HTML documents.
  </annotation>
  <state>http://marklogic.com/states/initial</state>
  <on-success>http://marklogic.com/states/converted</on-success>
  <on-failure>http://marklogic.com/states/error</on-failure>
  <priority>9200</priority>
  <execute>
    <condition>
      <module>
       /MarkLogic/cpf/actions/mimetype-condition.xqy
      </module>
      <options
        xmlns="/MarkLogic/cpf/actions/mimetype-condition.xqy">
        <mime-type>text/html</mime-type>
      </options>
    </condition>
    <action>
      <module>
       /MarkLogic/conversion/actions/convert-html-action.xqy
      </module>
      <options
       xmlns=
       "/MarkLogic/conversion/actions/convert-html-action.xqy">
       <destination-root/>
       <destination-collection/>
      </options>
    </action>
  </execute>
 </state-transition>
</pipeline>
```

Such a framework allows to convert arbitrary input data to XML if one supplies the system with rules and processes to apply. MarkLogic delivers a default content processing option, which includes pipelines for transforming PDF, MS Office, DocBook and other formats to XML. When this option is activated for a given database, documents which enter it and have one of these formats are automatically stored as XML. A user can add also their own custom pipelines.

3.4. Zorba

Zorba is an open-source XQuery processor written in C++. It is designed to be embedded into other systems and consequently is able to process XML stored in different places - main memory, mobile devices, browsers, disk-based, or cloud-based stores. The analysis here is conducted with version 2.1.0 of Zorba.
In order to be pluggable in diverse kinds of systems, an XQuery engine has to be able to

work with various data sources and data formats. Zorba achieves this by shipping a rich XQuery library and a C++ API, which allows the execution of queries. Most prominent among the available modules for input and output is the *fetch module*, which offers functions for getting the content or content type of a resource identified by a URI. Another helpful module is the HTTP client providing functions for performing HTTP requests. As far as the supported data formats are concerned, Zorba ships several extension functions for handling data different from XML. Examples of these are such for conversion between CSV and XML and vice versa, for tidying HTML and for conversion between JSON and XML. Apart from these XSL transformations and XSL-FO processing are supported, too. The next code snippet[ZOR] demonstrates how a simple XSL-FO document can be converted to PDF and stored on the file system:

```
import module namespace fop="http://www.zorba-xquery.com/modules/xsl-fo";
import module namespace file="http://expath.org/ns/file";
declare namespace fo = "http://www.w3.org/1999/XSL/Format";

(: PDF text :)
let $xsl-fo := 'Hello, world!'

(: Generate PDF :)
let $pdf := fop:generator($fop:PDF, $xsl-fo)

(: Write PDF into a file :)
return file:write-binary("simple.pdf", $pdf)
```

Aside from PDF, the XSL-FO module can convert documents to PS, PCL, AFP, Text, PNG, Postscript, RTF and TIFF, too.

3.5. BaseX

BaseX is a light-weight and high-performance XML database and XQuery engine. Since it will be presented in detail in Chapter 5, we will only have a quick look at its current input and output features just for the sake of comparison with the other presented systems. The analysis is done with version 7.1.1.
Among the data channels offered by BaseX are a graphical user interface, a command line tool, REST, XML:DB and WebDAV APIs and a lot of XQuery extension functions. Besides XML, BaseX supports HTML, JSON, CSV and binary data. Documents with such formats can be easily imported into a database via GUI or command line. The next example shows how using the commands provided by BaseX a user can use the CSV parser and specify parser options:

```
SET PARSER csv
SET PARSEROPT encoding=utf-8, lines=true,
              format=verbose, header=false, separator=comma
```

After these lines are executed, BaseX will handle files, entering the currently opened database, as CSV files. BaseX will process them using the CSV parser and the specified options. The same functionality can be used from the GUI and XQuery, too. Though a separate XQuery function for parsing CSV is not provided at present. The REST and XML:DB APIs can also handle the above listed formats. The XQuery implementation offers functions for storing raw data and converting JSON to XML representation. As far as the output is concerned, the same formats excluding CSV can be returned by the command line, the REST API and corresponding XQuery extension functions. Execution of XSL transformations is supported through XQuery. WebDAV is capable of handling only XML data. BaseX can work with both local and remote data sources. Besides the GUI and command line this functionality is exported as an XQuery module for reading and writing files on the file system and sending HTTP requests.

	XQuery	GUI	Command Line	REST	XML:DB	WebDAV
HTML	-/✓	✓/✓	✓/✓	✓/✓	✓/-	-/-
Text	-/✓	✓/✓	✓/✓	✓/✓	✓/-	-/-
CSV	-/-	✓/-	✓/-	✓/-	✓/-	-/-
JSON(ML)	✓/✓	✓/✓	✓/✓	✓/✓	✓/-	-/-
binary formats	✓/✓	✓/-	✓/✓	✓/✓	✓/✓	✓/✓
OO formats	-/-	✓/-	-/-	-/-	-/-	-/-

Table 3.3.: BaseX 7.1.1: Input/Output

3.6. Conclusion

The conducted investigation shows that it is not unusual for an XML database and XQuery processor to handle data different from XML. However, a quick look at the above tables reveals several shortcomings in the existing solutions. If we take the data channels provided by a system and the data formats, which can flow through them, we can observe some lack of *harmony* between input and output. In other words if data in some format can enter a database and be kept there as XML, this does not mean that it can leave the database in the same one format and often this is actually expected. Furthermore, in most cases a mechanism is absent, which allows a user to indicate in some way how they want to store their data or how they want to retrieve it out of the database. Consequently often the functionality offered by one channel for input and output does not match this offered by another one. If a system aims to be equally useful to each of the actors described in Chapter 2, this should not happen.

4. Generic Architecture for Input and Output

The analysis in the previous chapter has shown that in many cases there are inconsistencies between the supported input and output in an XML database or XQuery processor. Often data can enter a system in a given format through a given channel, but: cannot leave it in the same format through the same channel, or cannot leave it in the same format through any channel at all. We believe that at the root of this dissonance lies almost always a badly designed interface for data input and output. In addition it is hard to export this functionality consistently through XQuery, APIs and other user interfaces. This is why in this chapter we will start by defining some general requirements, which shall be met by such functionality. Based on them we will propose a generic architecture for data input and output, which can be implemented by any XML database. At the end we will show how this functionality can be exposed to the different types of actors.

4.1. Requirements

The requirements we are going to define here are directly related to those mentioned in Chapter 2 when describing the use case for extending the input and output functionality of an XML database. Thus we will discuss aforementioned ideas in some more depth.
First, the logic which takes care for data input and output has to be strongly decoupled from the rest of the system's components. This leads to the notion of modular architecture. As it is defined in [MOD], the beauty of such an approach is that one can replace or add any one component (module) without affecting the rest of the system and this is what we actually strive for. The architecture must allow users to plug in support for new data formats, i.e., adding new parsers or serializers, invisible to the remaining parts of our XML database. We want the storage implementation, the XQuery implementation, the various APIs and user interfaces to be absolutely unaffected by such changes. They should neither care for the format of the data and how it shall be treated and brought to XML nor vice versa. They should only receive it after it was processed by a parser or a serializer and either store it or give it back to the user in the form he/she has requested.
Second, this logic has to be centralized so that it can be reused from everywhere. This requirement can be easily met when the first one is fulfilled. This is why we will not discuss it further or separate it as an individual one. However, centralization remains an important aspect when it refers to defining diverse options for import and export of

data. Among these are how the various data formats shall be stored – as XML or as raw data; what shall be stored when it comes to binary formats – content, metadata or both; how a specific format must be parsed to XML, i.e. which parser options must be applied; how XML must be serialized to a specific format or which serialization options must be applied. All of these possible settings shall be configurable and accessible through all data channels offered by a system.
These are the two main requirements we are going to follow while designing our solution for data input and output – modularity of the functionality for handling different data formats and central configuration of input and output options, parser and serialization parameters. The prize that we will win if we stick to them is a consistent implementation, which satisfies the needs of all three types of users and can always serve as a basis when extending the XML database with new data channels.

4.2. Architecture

4.2.1. Data Flow

Before we continue with the actual design of our architecture it would be useful to analyse how the input and output processes in the system should look like if we follow the mentioned requirements. That is why we will begin with a brief investigation on the data flow in the system, which will help us later to model the main components of our solution.

4.2.1.1. Input

We begin with the input. Figure 4.1 gives an overview of the steps which must be taken once data enters an XML database through some of the provided channels. The first one is to determine its content type. This is needed in order to decide how it shall be processed and to choose the appropriate parser for it. We already mentioned that an XML database may offer diverse options through which a user can manage the way their data is handled. Among these options are such that indicate how data with given content type must be processed – as XML, as raw as well as such that refer particularly to the parser to be used. Let us call the first ”input options” and the latter – ”parser options”. Input options also dictate what actually shall be processed – only metadata, only content or both. This is important when a user has to deal especially with binary files like images and videos, for instance. In such case it is clear that the content cannot be turned into XML. Representing it as a *Base64* item is not a good option, too. However, leaving it raw, i.e. in its original format, and converting its metadata to XDM, may be a better approach.

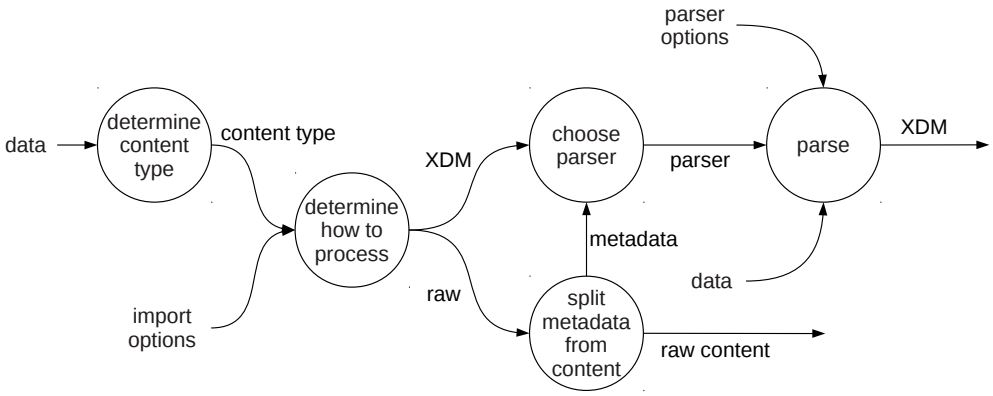


Figure 4.1.: Input Data Flow

Another possible solution is to work exclusively with metadata because in most of the cases it contains the useful information to deal with. Once the content type is known, the input options have to be checked in order to determine how to proceed. If the data has to be converted to XML, an appropriate parser has to be chosen. If the data must be kept raw in its original format and it is indicated to parse its metadata separately, then an appropriate parser has to be selected for the format of the metadata. The last step from the process is the actual parsing which is done using the appropriate parser, the corresponding options for it and the data itself. The final result is the parsed data in the database-specific XDM representation. In case of binary data, the content is treated in a way specific to the system in use.

4.2.1.2. Output

We continue with the output process. It always depends on the target format in which the data has to be converted. Another important thing is whether the data is an XDM instance or it is in its original raw format. In case of XDM the next step is clear – it is serialized to the target format taking into account the serialization options. In case of raw data the only serialization which takes place is this of the metadata and it is transformed to the corresponding metadata format. A simple example can be given with an MP3 file which content is stored as raw and its metadata – as XML. Once this file is to be exported from the XML database, its metadata will be serialized back to ID3 and synchronized with the raw content in case any changes have been performed beforehand.

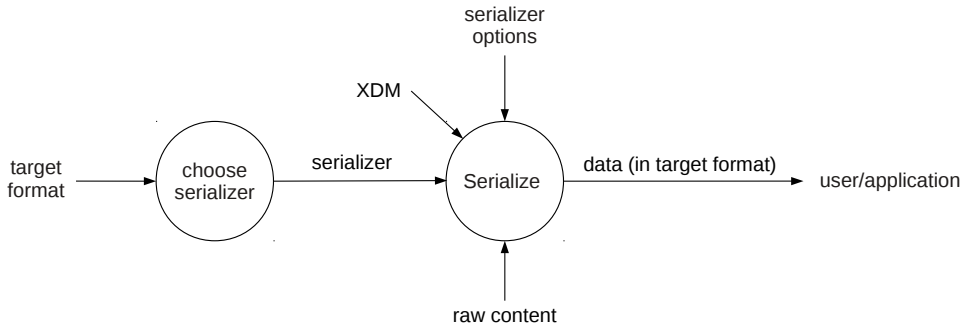


Figure 4.2.: Output Data Flow

4.2.2. Input

At this point we have a general idea how our mechanism for input and output shall work. This is sufficient to start taking a closer look at the described steps and think what kind of components are needed for a concrete implementation of the concept. In this and the next section we will model a framework of several classes which will serve to achieve the presented data flows. We will try to make it flexible enough to meet the two requirements we have defined at the beginning of the chapter. The definition language will be UML for Java. Though, the framework shall be implementable in any other object oriented language.
As in the previous section we are going to look individually at the input and output processes and we will start with the input. The most intuitive way to begin is to consider which are the main ”participants” in the data flow and which are the main actions taking place.

4.2.2.1. Data Sources

The input process always starts with a data source. This can be a file or a collection of files – directory or archive, on a local or a remote machine. It can be also a data stream. Nevertheless, there are several things which have to be known in order to proceed with the processing. First, the content type of the data is needed because based on it an appropriate parser has to be chosen. Second, in some cases the name of the resource and the data size may be necessary and thus they have to be provided, too. Another important point, which may influence the next steps, is whether the incoming data is a simple file or a collection of files. Finally, a data source implementation shall provide

a way to read the data itself. Based on these requirements we can define an interface which will allow these necessary actions to be performed. Figure 4.3 shows the UML diagram corresponding to it. The classes which will implement it will represent different types of data sources.

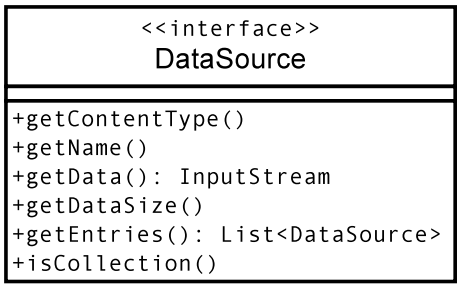


Figure 4.3.: Interface `DataSource`

The method `getData()` returns the stream from which the actual data can be read. `getContentType()` gives back the content type. The methods `isCollection()` and `getEntries()` can be used to check if the data source is a directory/archive and to get the corresponding entries from it as a list of `DataSource` instances.
Listings 4.1 and 4.2 are example Java implementations of `DataSource`. `HttpDataSource` represents a data source located on an HTTP server. As it can be seen in its constructor is opened an HTTP connection to the address on which the resource can be found. The content type is taken from the header *Content-Type* and the data is read from the input stream of the established connection.
The class `LocalDataSource` is a sample implementation of a data source located on a local machine. Here the constructor has a different behavior as we are working with resources on the file system. The only thing that it does is to create an instance of `java.io.File` for the file with the given address. The way the content type is determined depends on the implementation. The data is read from the input stream associated with the given file.

Listing 4.1: HttpDataSource.java

```
public class HttpDataSource implements DataSource {
    private URLConnection conn;

    @Override
    public HttpDataSource(String address) {
        URL url = new URL(address);
        conn = url.openConnection();
  }

    @Override
    public String getContentType() {
```

```
        return conn.getContentType();
    }

    @Override
    public InputStream getData() {
        return conn.getInputStream();
    }
}
```

Listing 4.2: LocalDataSource.java

```
public class LocalDataSource implements DataSource {
    private File sourceFile;

    @Override
    public LocalDataSource(String address) {
      sourceFile = new File(address);
    }

    @Override
    public String getContentType() {
      // Get the content type of the resource
      return determineContentType(sourceFile);
    }

    @Override
    public InputStream getData() {
      return new FileInputStream(sourceFile);
    }
}
```

4.2.2.2. Parsers

The `DataSource` interface defines a common way to work with data sources. They can provide data with different content types. However, there are always only two options to process it – either to turn it into an XDM instance or to leave it as it is in its original format. This leads to the need for a unified way to parse data with various content types. Here we are going to define how shall look like the interface of a common parser used in an XML database.
First, if data with a given format cannot be converted to XDM or it is explicitly stated that it shall be left raw, then it suffices to just read it from the data source and do not parse it. Of course, it can always be encoded in *Base64* and stored in the database or returned as an item but this does not make much sense. In that form it will not be useful since querying and manipulating via XQuery is impossible. Second, data often comes with other data which describes it, namely metadata. It is not unusual if the metadata is sometimes more helpful to a user than the content it refers to. This is why a good mechanism for input shall be able to treat content and metadata separately.

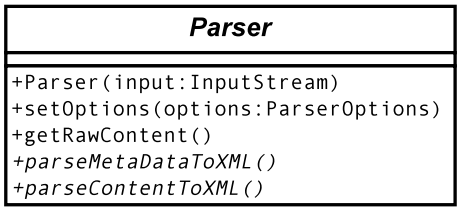


Figure 4.4.: Abstract Class `Parser`

Having in mind these requirements we define the next important part of our mechanism – the abstract class `Parser`. Figure 4.4 shows its UML definition. No matter what is parsed – just metadata or both metadata and content, this operation depends always strongly on the format of the data we are dealing with. That is why we leave the methods `parseMetaDataToXML()` and `parseContentToXML()` abstract. They will be implemented differently for each content type. On the other hand, getting the content in its raw form from a data source is trivial. Thus `getRawContent()` must be implemented directly in the class `Parser` with the database-specific logic.
The advantage of the `Parser` class is that it offers flexibility. In other words a potential user – an application developer, for example, can work with any ”part” of their data and they can have it in both possible forms – XML and raw. In this way, if they want to retrieve the metadata of an image file, for instance, they can use the `parseMetaDataToXML()` method and have it as XML. If they want just the raw content, they can use `getRawContent()`. Furthermore, if they require the whole image file as XML, a possible way to have this is if the corresponding implementation returns a sequence of two items - one element representing the metadata and a second one with the *Base64* encoded content.
Working separately with metadata and content is convenient but the relation between them has to be maintained in some way because they together constitute a whole resource. Once they are parsed individually, they still have to remain connected because a change in the metadata always has to be reflected on the content. Therefore, we need a component which represents a resource after it has been processed, i.e. an encapsulation of the parsed metadata and the parsed or raw content. Figure 4.5 shows the corresponding UML diagram. The `Resource` class represents a wrapper around an already parsed resource. Since such a resource can be instantianted in various ways – only with metadata, both with metadata and raw or XML content, only with content, the *Builder* design pattern shall be used for its implementation. Once the input is parsed, it can be packed in such a wrapper and handled directly to the storage mechanism, for example. Furthermore, when data comes from the database, i.e. in case of output, it can be wrapped again in this way by the storage and handled to a preferred serializer, for instance.

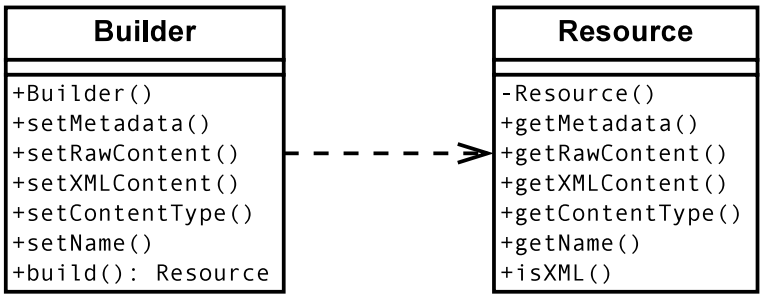


Figure 4.5.: Class `Resource`

4.2.2.3. Options

The components we have defined until now meet to great extent the aimed requirement for modularity. The `DataSource` interface and the `Parser` abstract class allow to add new functionality for input to the system without affecting any other part of it. Furthermore, as we tried to stick to the rule that each separate component must be responsible for one particular task, the resulting interfaces are intuitive enough to be easily learned and used by an application developer who is not acquainted with the specifics of the XML database. However, the whole picture is still not complete. A developer does not need to be familiar with the internals of the system but if they want to use its input functionality, they have to know how each content type can be processed, which is the corresponding parser for it and what options are associated with this parser. If the variety of supported content types is wide, this becomes a difficult task, which automatically decreases the system's user-friendliness. This brings us to the second requirement we have defined, namely the centralization of input and parser options.
Since we have already mentioned self-description as one of the advantages of XML and since we deal with XML databases, the most natural way to implement the concept of centralization is to use XML itself. A way to do this is to keep the necessary information about content types and parsers in the form of XML files on exactly one place in the system. Where this place should be depends on the concrete XML database. The structure of these files is defined by the XML Schemata A.1 and A.2. A closer look at them reveals the basic idea. In the XML file defining the input options, each element `<input>` corresponds to an input format supported by the system. It has four attributes with the following meanings:

- `content-type`: indicates the MIME type of the data as specified in RFC2046
- `process-as`: indicates how content shall be processed – as XML, as raw, as mixed or none meaning that it shall not be processed at all

- `process-metadata`: indicates if the metadata shall be processed separately or not
- `parser`: indicates the name of the parser responsible for the parsing from `name` to `format`

In this way, if an XML database supports processing of MP3 data and offers a parser for ID3 metadata, this can be made clear as follows:

```
<input content-type="audio/mpeg"
       process-as="raw"
       process-metadata="true"
       parser="input.parsers.MP3Parser"/>
```

This means that when *audio/mpeg* data is processed, its content will be left in its original format, its ID3 metadata will be parsed to XML and all this will be done by the parser `input.parsers.MP3Parser`. If only metadata shall be processed, then `process-as` must be set to `none`.
The options associated with each parser are listed in a separate XML file. In it each `<parser>` element corresponds to a parser. Its name is specified by the `name` attribute which must have the same value as the `parser` attribute in the according entry in the input options. The offered parser options are declared as children of the `<parser>` element. In this manner an HTML parser, for instance, can be presented in the following way:

```
<parser name="input.parsers.HTML">
  <options>
    <doctype>omit</doctype>
    <char-encoding>utf-8</char-encoding>
  </options>
</parser>
```

With these two XML files placed somewhere in the system the concept of centralization is realized. Although this is a consistent and convenient way to manage the data processing, it is restrictive to some extent. This comes from the fact that the options are specified on a system-wide level. If a user wants to store files with particular content type as XML in one database but as raw in another or just wants to parse them differently in the different databases, this would be impossible or they will have to change the settings every time they switch the database. This seems quite annoying and this is why it would be much better if the same options could be controlled on a database level, too. This could be accomplished by allowing the user to make their own configuration of the same settings for each database. If the system-wide configuration satisfies their needs, however, they could use it as a default one. This feature can be provided by keeping two XML files with exactly the same structure for every database. They shall contain only the entries for the ”affected” content types and parsers. Whenever data enters a database,

first it will be checked if this database has a configuration associated with it and if yes - it will be taken into account. Otherwise the default one will be used.

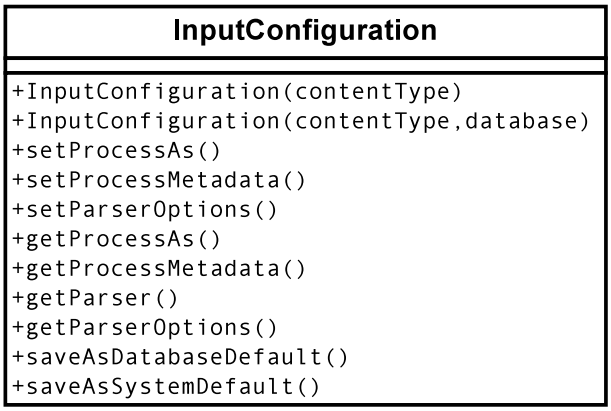


Figure 4.6.: Class `InputConfiguration`

In order the described options to be easily accessible and manageable, we will define a class which will be dedicated exclusively to this purpose. Let us call it `InputConfiguration`. It represents the settings referring to a single content type supported by the system. It is always instantiated with a content type name or with a content type and a database name. If no database is set or the given database does not have a configuration associated with it, the above three attributes are read from the system-wide configuration. Otherwise, the database-specific one is used. The corresponding `get` and `set` methods can be used to retrieve and change the existing settings. A change can be persisted either as database-specific using the method `saveAsDatabaseDefault()` or as system-wide using `saveAsSystemDefault()`. Figure 4.6 shows the UML diagram for `InputConfiguration` and Figure 4.7 depicts the initialization process.

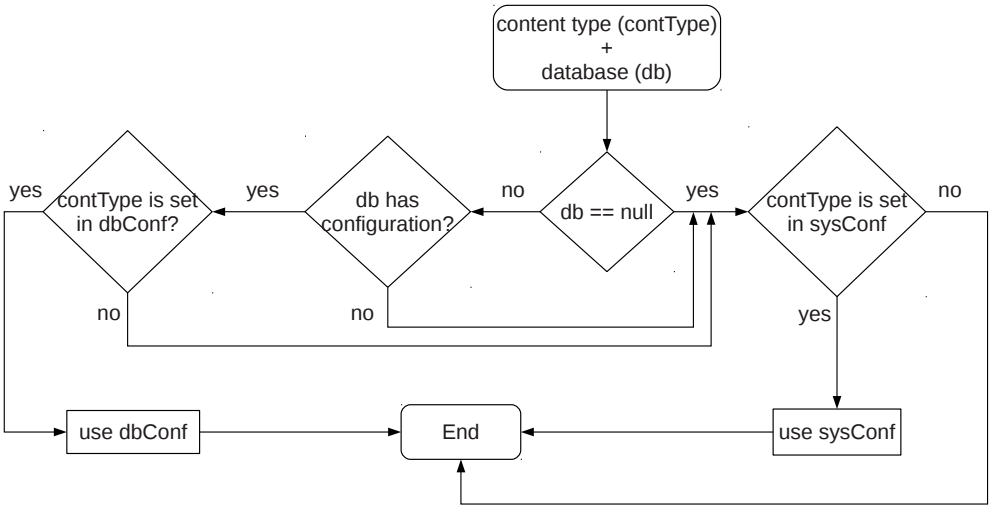


Figure 4.7.: `InputConfiguration` Initialization

4.2.2.4. Direct Processing

The classes presented in the previous sections correspond to separate components that can be used together to accomplish the input data flow depicted on Figure 4.1. Although they define an intuitive way for data processing, it would have been more convenient if the whole workflow can be "automated" in some way and controlled by just one module. In other words it would have been quite well if a user can just pass their data to the database and it itself decides how to process it. For this purpose we need one last component which will make our input architecture complete. Its UML definition is given in Figure 4.8.

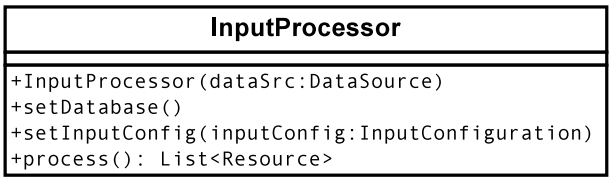


Figure 4.8.: Class `InputProcessor`

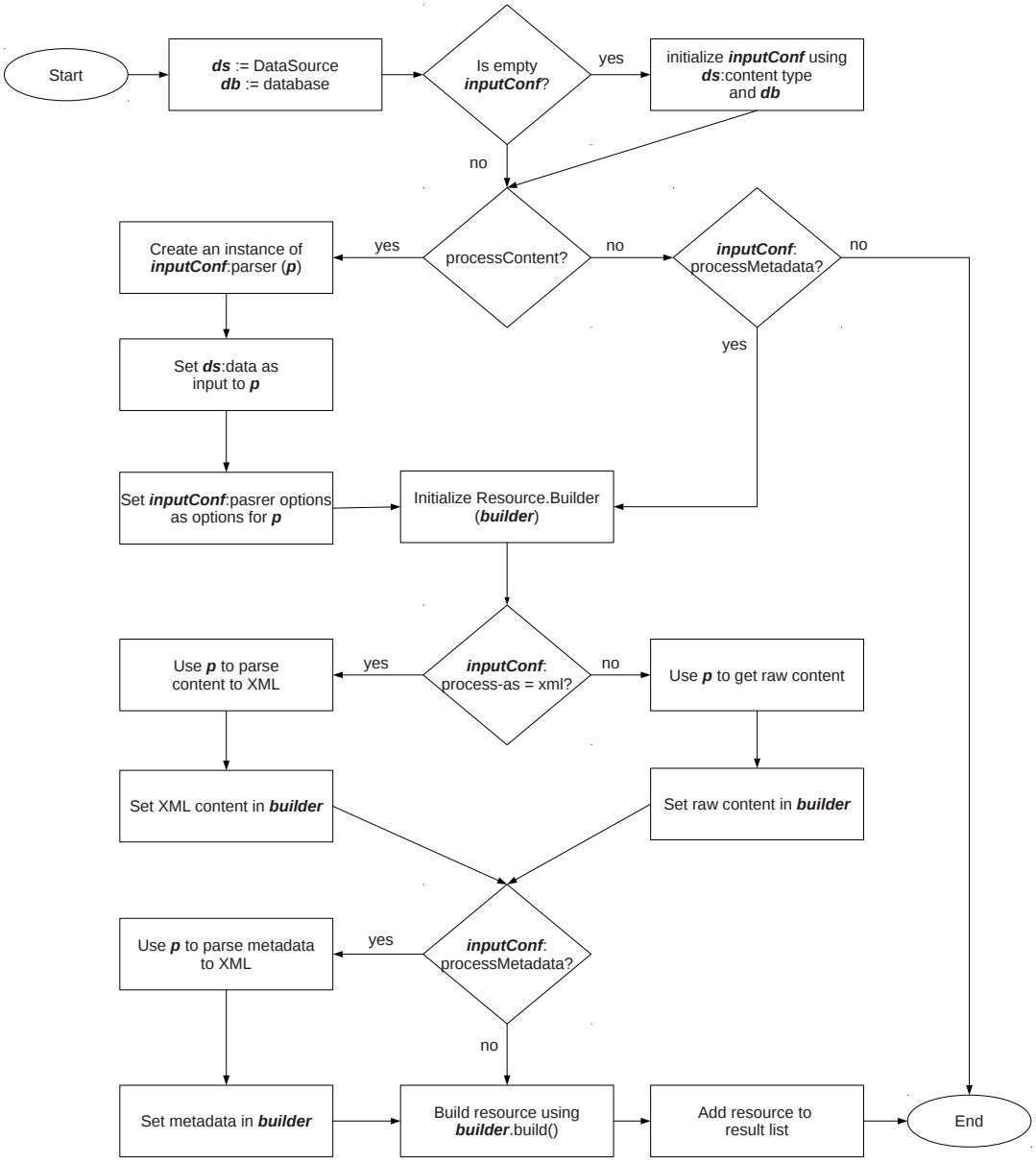


Figure 4.9.: Process a single file

The `InputProcessor` class works hand in hand with the rest of the components. It is always instantiated with a data source from which the input is read. If it is required to use the input configuration of a particular database, then the name of the relevant database can be set via `setDatabase()`. Another option is to set directly a ready input configuration using `setInputConfig()`. If no database or configuration is set, the default system-wide configuration will be used. The `process()` method represents the above mentioned concept of automation. The flowchart on Figure 4.9 describes the way it works for a single file. First, it is checked if there is an initialized input configuration and if not – one is initialized as shown on Figure 4.7. The next step is to check if the content has to be processed. If yes – an instance of the corresponding parser is created, the data from the data source is set as its input and the options from the configuration – as its options. After that a `Resource` instance is created using the `Builder` class. It is later populated with raw or XML content and metadata depending on what is written in the input configuration. Finally the resource is constructed and added to the result list.

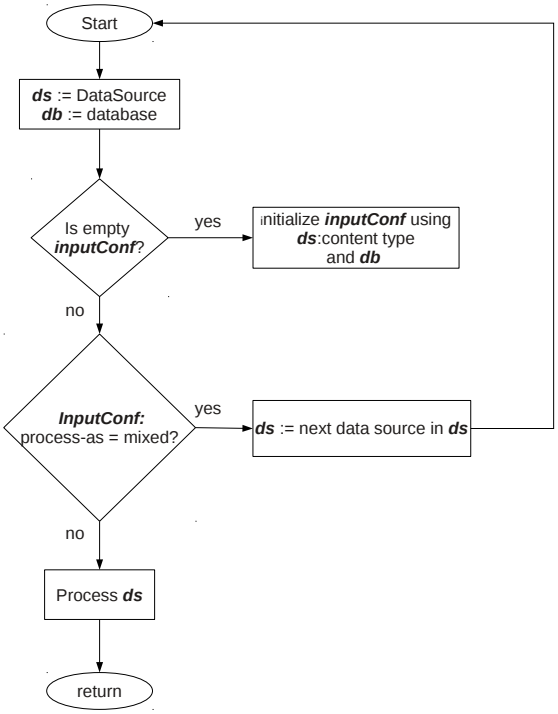


Figure 4.10.: Process a directory or archive

Figure 4.10 demonstrates how the `process()` method works when the data source is

a directory or archive. The input configuration is initialized in the same way but the following step checks if the indicated format is mixed. If this is the case, the method starts from the beginning with the next data source. If the format is XML or raw, the data source is processed as described above and the method returns to continue with the next one.

4.2.3. Output

In this section we are going to define the components which shall take care for the output in an XML database. The whole idea remains quite similar to the one used for the definition of the input components. However, as it will be seen, the output process is probably a little simpler and determined to a great extent by the user.

4.2.3.1. Serializers

We begin with the serializers. Their only task is to transform data from XDM to some content type desired by the user. Of course, this process always depends on what is this content type and in case the data comes from the database – how it was stored there. When it comes to data which was stored entirely as XML, the serialization process is straightforward – the internal representation has to be transformed to the target format and the result has to be written in some destination given by the user. When, however, it comes to binary data which content was stored as raw and its metadata – as XML, then some additional processing has to be done, e.g. synchronizing the metadata in case it was updated in the meantime.[1] Furthermore, it may be required to output exclusively the content of a resource – without any metadata – in order to reduce its size. Apart from that a serializer may accept various options which shall be possible to set before processing. Having all these requirements in mind, we can define how the interface of a serializer shall look like. Figure 4.11 shows the corresponding UML diagram for it.

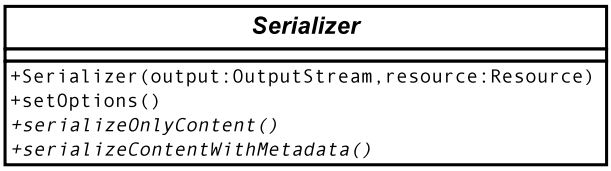


Figure 4.11.: Abstract Class `Serializer`

[1] Clearly ”serialization” (and in case of input - ”parsing”) is not the correct term to use in case we have binary data but for the sake of a unified approach for input and output, we will leave this like that.

A serializer is always instantianted with a resource which must be serialized and an output stream where the result shall be written. The serialization process is strongly dependent on the requested output format. This is why the methods `serializeOnlyContent()` and `serializeContentWithMetadata()` are abstract. They must be properly implemented in the relevant serializers. On the contrary – setting the preferred options shall be common to all serializers and thus this method is left non-abstract.

4.2.3.2. Options

Following the course of definition we used for the input, we arrive at the point where the centralization concept in case of output shall be realized. Here is used the same idea – there are two XML files which hold the system-wide ouput and serializer options and whenever a database-specific configuration has to be made, XML files with the same structure but containing only the relevant output content types and/or the relevant serializers are created for the given database. The corresponding XML schemata are shown in A.3 and A.4 respectively. As it can be seen, the output options have a little simpler structure than the these for input. For each possible output content type there is an element `output` which has three attributes:

- `content-type`: name of the target content type as specified in RFC2046
- `serializer`: name of the serializer which is responsible for serialization to `content-type`
- `metadata`: indicator showing if metadata shall be serialized as well. This attribute is optional and shall be used only when it comes to binary content types which were originally stored as raw content plus XML metadata.

In that way, an entry specifying `application/json` as a target content type would look like as follows:

```
<output content-type="application/json"
        serializer="output.serializers.JSONSerializer"/>
```

Consequently the corresponding entry in the serialization options may be defined in the following way:

```
<serializer name="output.serializers.JSONSerializer">
  <options>
    <json-format>JsonML-array</json-format>
    <whitespace>indent</whitespace>
  </options>
</parser>
```

We define the class `OutputConfiguration` for accessing and managing the output and serialization options. Its UML diagram is shown on figure 4.12. As it can be seen there are two ways to instantiate it – either only with a target content type or with a target content type and additionally a database name. The initialization process is the same as by the input: in case a database is specified and it has an output configuration with the given target format in it, it will be used. Otherwise, the system-wide one will be taken into account. Changes to the existing configuration can be made via the methods `setOnlyContent()` and `setSerializerOptions()`. These changes can be saved either as system-wide or as database-specific.

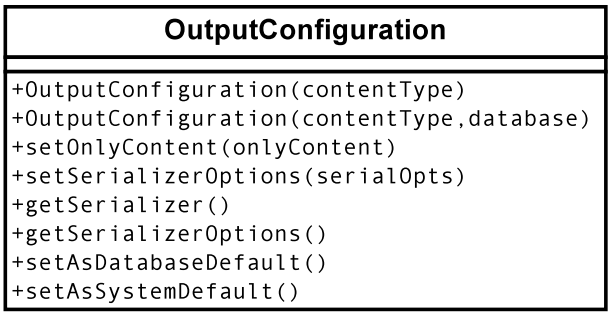


Figure 4.12.: Class `OutputConfiguration`

4.2.3.3. Direct Processing

As by the input, it would be useful to provide a more "intelligent" way for the output data flow. For this purpose again we will use a class which will work together with the other two defined components – serializers and configurations. Figure 4.13 shows how it shall look like.

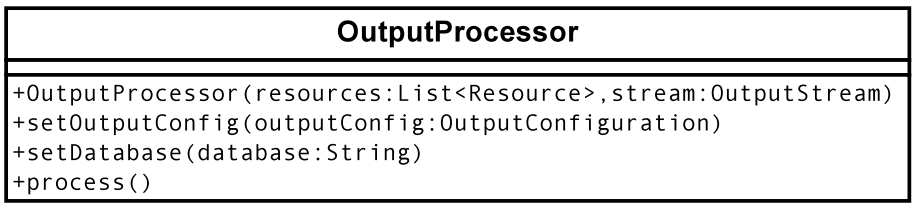


Figure 4.13.: Class `OutputProcessor`

`OutputProcessor` is always instantiated with two parameters:

- `resources`: this is a list of `Resource` instances holding the documents to be serial-

ized

- `stream`: this is the output stream in which the result from the serialization must be written

If no special options have to be set or the system-wide configuration is acceptable, calling the `process()` method shall suffice to complete the serialization process. If a database-specific configuration must be used or the existing configuration has to be overwritten, then `setDatabase()` or `setOutputConfig()` can be called beforehand. The way `process()` works is described by the flowchart in figure 4.14. In case multiple data sources have to be packed in an archive, the same logic will be executed for each of them and finally the output stream in which the result from the serialization is written can be passed to an archiving mechanism, for instance.

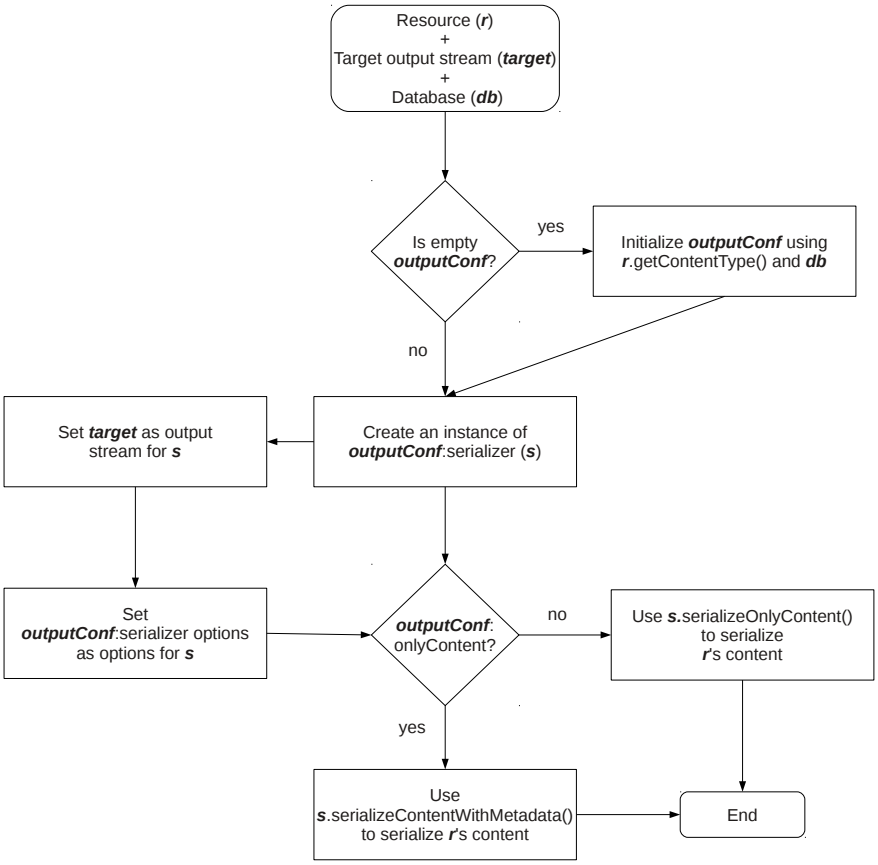


Figure 4.14.: Output processing with the `OutputProcessor` class

4.3. Usage

One of the advantages of the previously defined functionality is that it can be reused. Consequently, the same features can be easily exposed via various interfaces to the diverse types of users. In this section we will demonstrate how this can be accomplished for the three use cases listed in Chapter 2.

4.3.1. Extending the Input and Output Functionality

We start with showing how support for new content types can be added to an XML database implementing the above proposed architecture. Three examples will be presented – one with a content type which can be parsed to XML, one with binary data and one with archived data.

4.3.1.1. Input and Output of HTML data

We begin with a simple example demonstrating how to extend an XML database with support of HTML data. In order this to be possible, first we have to create a class inheriting the abstract class `Parser`. Let us call it `HTMLParser`. The next step is to implement the methods `parseContentToXML()` and `parseMetaDataToXML()`. The concrete parsing logic is not interesting to us since it can vary from system to system. For the sake of the example, however, `parseContentToXML()` can tidy up the HTML code and then transform it to XHTML and `parseMetaDataToXML()` can read exclusively the `<meta>` tags. Once the `HTMLParser` is developed, we can register it in the system-wide input options. A sample entry may look as follows:

```
<input content-type="text/html"
       process-as="xml"
       process-metadata="true"
       parser="input.parsers.HTMLParser"/>
```

The corresponding entry in the system-wide parser options may be the following:

```
<parser name="input.parsers.HTML">
  <options>
    <doctype>omit</doctype>
    <char-encoding>utf-8</char-encoding>
    <newline>LF</newline>
    <tidy-mark>no</tidy-mark>
  </options>
```

```
</parser>
```

Adding support for HTML output is done in a similar way. First a serializer class has to be created – `HTMLSerializer`, which inherits the abstract class `Serializer`. The methods `serializeOnlyContent()` and `serializeContentWithMetadata()` must be implemented. For example, if the XML database supports XSLT transformations and the relevant logic is easy to reuse, an XSLT stylesheet can be passed as an option to the serializer and it will execute the corresponding transformation. Thus, a sample entry in the output options would look like:

```
<output content-type="text/html"
        serializer="output.serializers.HTMLSerializer"/>
```

The entry specifying the serializer options can be the following:

```
<serializer name="output.serializers.HTMLSerializer">
  <options>
    <stylesheet>xmlToHtml.xslt</stylesheet>
  </options>
</parser>
```

4.3.1.2. Input and output of JPEG files

Adding support for JPEG data is done in a similar way. A sample implementation of `parseMetaDataToXML()` can read the EXIF metadata of a JPEG file and represent them as an XDM instance. `parseContentToXML()` can convert the JPEG file to a sequence of two items - one element containing the parsed EXIF data and a *Base64* item for the content. The entry in the system-wide input options may look as follows:

```
<input content-type="image/jpeg"
       process-as="raw"
       process-metadata="true"
       parser="input.parsers.JPEGParser"/>
```

As far as the output is concerned, again the `Serializer` abstract class has to be inherited. The corresponding output options may be the following:

```
<output content-type="image/jpeg"
        serializer="output.serializers.JPEGSerializer"
        onlyContent="false"/>
```

According to it when a JPEG file is requested from the database both content and metadata will be ouput.

4.3.1.3. Input and Output of .chm files

Until now we have shown how to add support for HTML and JPEG data. Next, it would be interesting to demonstrate how a file consisting both of HTML and image files can be processed in an XML database. Suppose that a user develops a web application consisting of several HTML pages containing text and images and they use an XML database as a backend. The pages are compressed in order to save loading time and thus when they are imported into the database they come in a *.chm* format. Adding support for this format will be quite easy since the logic for HTML and JPEG data is already present. What we need to do is just to add the *.chm* MIME type to the input options and indicate that we want to have it ”mixed” in the XML database. This means that HTML data will be parsed to XML and JPEG files will be left in their original format. The metadata of both file formats will be converted to XML.

```
<input content-type="application/vnd.ms-htmlhelp"
       format="mixed"/>
```

As it can be seen, no parser is specified for the given content type. This comes from the fact that *.chm* files consist of ”simple” files for which parsers do already exist. We need only to declare it in the configuration in order to make it clear to the database that it can accept such data. The situation with the serialization is similar. The entry in the output configuration shall look like as follows:

```
<output content-type="application/vnd.ms-htmlhelp"
        onlyContent="false"/>
```

No specific serializer is needed for the MIME type *application/vnd.ms-htmlhelp* because the necessary serializers are already implemented. Of course, a *.chm* file can contain other data apart from HTML and JPEG for which support will have to be added but the example in this case is left as simple as possible. Furthermore, once the needed data is serialized back, it has to be zipped into an archive. This logic, however, shall not be part of the input and output framework described here but a separate module which cares only for packing the data.
The three examples given above aimed to show that if the functionality for input and output in an XML database is as much as possible isolated from the rest of the system, adding support for new content types becomes a straightforward task. Database developers can concentrate only on the logic for parsing and serializing since no dependencies to other components exist.

4.3.2. Application Development

In this section we are going to see how an application developer can use the functionality integrated with the above examples. We have already mentioned in Chapter 2 that when it comes to applications we can divide them in two main groups – such that communicate with an XML database through some kind of APIs and such that are written purely in XQuery. Both groups will have to use in the end the described framework, however. The examples here will show how the relevant classes can be called for specific scenarios and how the functionality offered by them can be exported to XQuery functions. As we do not work with a particular database, everything that will be demonstrated is just a suggestion and, of course, can be changed or done in another way.

4.3.2.1. Input and Output of HTML data

We begin again with the input and output of HTML data. Listing 4.3 shows a Java example how an HTML resource can be retrieved and stored in an XML database. As it can be seen, first an instance of `DataSource` has to be created in order to get the data from the resource. In the second step the input configuration for the given content type is retrieved and the parser options are updated. After the change, the configuration is stored as database default. This means that from now on the new parser options will be applied on each document with content type *text/html*, which enters the database with name *myDb*. At the end the corresponding parser is instantianted and the data is parsed. With the result from the parsing step, a new `Resource` instance is built which is passed to the storage mechanism. If there is no need to update the input configuration this whole process can be a lot shorter - only a `DataSource` instance will have to be created, passed to the `InputProcessor` constructor and finally the `process()` method will do the entire work and return a ready resource.

Listing 4.3: Storing an HTML resource

```
// Create a data source
HttpDataSource dataSource = new HttpDataSource(address);

// Get inputConfig for input content type
String contentType = dataSource.getContentType();
InputConfiguration ic = new InputConfiguration(contentType, "myDb");

// Update inputConfig and save it as database default
ParserOptions parserOptions = ic.getParserOptions();
parserOptions.setOption("newline", "CRLF");
ic.saveAsDatabaseDefault();

// Instantiate HTML Parser
HTMLParser htmlParser = getParser(ic.getParser(), dataSource.getData());
htmlParser.setOptions(parserOptions);
```

```
// Parse HTML Data
XMLData data = htmlParser.parseToXML();

// Build a resource with the parsed data
Resource.Builder htmlResourceBuilder = new Resource.Builder();
htmlResourceBuilder.setXMLContent(data);
Resource htmlResource = htmlResourceBuilder.build();

// Store resource
Storage.store(htmlResource);
```

This same logic can be represented with XQuery in different ways. The idea of the modular approach makes things easy because the underlying functionality is decoupled enough and almost each part of it can be exported in the form of an XQuery function. Thus, for example, the retrieval of input configuration for a particular content type can be done with a function with the following signatures:

```
get-input-configuration($content-type as xs:string) as node()

get-input-configuration($content-type as xs:string,
                        $database as xs:string) as node()
```

The HTML parsing itself can be exported as follows:

```
parse-html($path as xs:string) as node()

parse-html($path as xs:string, $options as item()) as node()
```

What happens ”underneath” the XQuery processor will be exactly the same as the shown in Listing 4.3. At the end, if the XML database in use offers extension functions for database management, the result from `parse-html()` can be passed to an appropriate function and stored in the database.
A more generic XQuery approach for parsing any kind of data supported by an XML database can be achieved by exporting the functionality of the `process()` method of `InputProcessor`. The resulting function can be similar to the standard XQuery `doc()` function:

```
resource($address as xs:string) as item()
```

The behaviour behind it shall include just the creation of a `DataSource` instance with `$address` and then calling the `process()` method of `InputProcessor`. In that way a resource will be retrieved and parsed according to what is written for it in the system-wide configuration.
Serializing XML to HTML is done in a similar way. Listing 4.4 demonstrates a sample approach. Here the process starts from the "opposite side", i.e. the XML database. There is a resource and it has to be exported as HTML to a file on the file system. A specific XSLT stylesheet has to be applied for the transformation and thus the corresponding serialization option is overwritten for the case. At the end, the resource, the options and the file output stream are set to the corresponding serializer and it does the rest of the work. Again, the same result can be achieved with less code when the `OutputProcessor` is used.

Listing 4.4: Exporting XML as HTML to a file

```
// Get outputConfig
OutputConfiguration oc = new OutputConfiguration("text/html", "myDb");

// Update serializer options
SerializerOptions serialOpts = oc.getSerializerOptions();
serialOpts.setOption("stylesheet", "/home/xslt/myStylesheet.xslt");

// Prepare target output
File htmlOutput = new File("/home/html/myHtml.html");
FileOutputStream fos = new FileOutputStream(htmlOutput);

//Instantiate HTML serializer
HTMLSerializer htmlSer = getSerializer(oc.getSerializer(), resource,
        fos);
htmlSer.setOptions(serialOpts);

// Serialize db resource to HTML
htmlSer.serializeContentWithMetadata();
```

The output functionality can be delivered with XQuery, too. A sample XQuery function for HTML serialization may have the following signatures:

```
serialize-html($resource as item(), $target-path as xs:string)

serialize-html($resource as item(), $target-path as xs:string,
                                    $options as node())
```

4.3.2.2. Input and Output of JPEG files

The steps for importing JPEG data in an XML database do not differ in any way from the ones used by the HTML import. What here may be interesting is how the equivalent XQuery functionality would look like. It shall allow to retrieve exclusively metadata as well as both metadata and content. A possible way to do this is the following XQuery function:

```
parse-jpeg($path as xs:string, $only-metadata as xs:boolean)

parse-jpeg($path as xs:string, $only-metadata as xs:boolean,
                               $options as node())
```

In this way the result from calling it with `$only-metadata` set on *true* may be:

```
<jpeg-image name='myImage.jpeg'>
  <metadata format='EXIF'>
    <make>EASTMAN KODAK COMPANY</make>
    <model>KODAK DX6490 ZOOM DIGITAL CAMERA</model>
    <created>11.05.2011 08:47:28</created>
    <aperture>F5.6</aperture>
    <focal>6.3mm</focal>
    <exposure>1/500s</exposure>
    <sensitivity>80/1 ISO</sensitivity>
    <mode>auto</mode>
    <flash>no</flash>
    <white-balance>auto</white-balance>
  </metadata>
</jpeg-image>
```

If the existing input configuration for content type *image/jpeg* is acceptable and does not need to be influenced, the `resource()` function can be used, too. In case content has to be returned, it can be encoded in *Base64* and added after the `metadata` element in the above example result.
As far as the output is concerned, again the steps using the Java implementation are similar to these when XML is serialized to HTML. The corresponding XQuery function may have the following signatures:

```
serialize-jpeg($resource as node(),
               $target-path as xs:string)

serialize-jpeg($resource as node(),
               $target-path as xs:string,
```

```
                $only-content as xs:boolean)

    serialize-jpeg($resource as node(),
                $target-path as xs:string,
                $only-content as xs:boolean
                $options as node())
```

`$resource` is the resource from the database which has to be serialized. Another way to define this parameter is as `xs:string` in which case it can be the path to a resource in the XML database. `$target-path` is the path to the file in which the data has to be exported. `$only-content` indicates if just content has to be exported, i.e. the metadata will not be included in it, e.g. it will stripped off in the serialization process. The last parameter `$options` specifies the serialization options to be used.

4.3.2.3. Input and output of .chm files

Finally we are going to show how a collection of data sources can be processed. If the `process()` method of `InputProcessor` is implemented as shown on Figure 4.10, this will be easy – just the `DataSource` instance corresponding to the *.chm* file has to be passed to the constructor. If this is not the case, however, the long way has to be taken and it should be iterated over the entries in the collection. For each of them the matching parser has to be called. Listing 4.5 shows the approach. As it can be seen the logic for parsing separate data sources in a collection is the same as the shown above. A possible way to retrieve the entries in such a resource using XQuery may be a function with the signature:

```
    get-resource-entries($address as xs:string) as item()*
```

Listing 4.5: Importing a .chm file

```
ListIterator<DataSource> entries = dataSource.getEntries()
        .listIterator();
while (entries.hasNext()) {
    // Get next entry
    DataSource nextEntry = entries.next();
    // Get input config according to content type
    InputConfiguration ic = new InputConfiguration(
    nextEntry.getContentType());
    // Instantiate corresponding parser
    Parser parser = getParser(ic.getParser(), nextEntry.getData());
    // Parse content
    Data content;
```

```
    if ("xml".equals(ic.getFormat()))
        // XML content
        content = parser.parseToXML();
    else
        // Raw content
        content = parser.getRawContent();
    XMLData metadata;
    if (ic.getProcessMetadata())
        metadata = parser.parseMetaData();

    // Build a resource with the parsed data...
}
```

The underlying logic will rely on the `InputProcessor` class. A `DataSource` instance will be created with the address of the *.chm* file and then it will be passed to a an `InputProcessor` instance. The rest of the work will be done by `process()` which will return a list of `Resource` instances. It can be presented as a sequence of XQuery elements, for example:

```
<resource name='resource1'>
  <metadata>
    ...
  </metadata>
  <content>
    ...
  </content>
</resource>
```

Of course, the elements `<metadata>` or `<content>` can be missing depending on the input configuration.
When several resources from the database have to be packed into a *.chm* file and exported, the process is similar as by the import. Here, however, each resource is passed to a serializer while the same output stream is used.

4.3.3. Input and Output through a User Interface

As it can be noticed the examples from the previous section are quite general and are not related to any existing implementation. They serve just to show how the proposed framework for input and output can be applied. The same is with the user interface – each XML database exposes in a different way its functionality to the end user. That is why here we will not go into details but just give some guidelines how the logic for input and output can be delivered to a regular user – not a developer but one who does not have a deep knowledge in XML, XQuery and does not have to be familiar with the internal implementation. In Chapter 5 we will describe how this can be done for an

existing XML database.
The UI should be kept simple and intuitive. It should provide access only to features which will be of interest to a regular user. For example, it would be useful to control the data processing and the various parser and serializer options. On the other hand – it is not necessary to know which parser or serializer class is responsible for the processing. Furthermore, it is important that the various options are presented in a consistent and understandable way. For instance, it should be visible for each supported content type – no matter input or output – what parser and serializer settings are valid. Another convenient feature would be to add easily new content types for which support was implemented. It is, however, controversial, if this should be allowed to any regular user or just to faciliate the work of developers and administrators.

4.4. Conclusion

In this chapter we defined a framework of classes, which can be used to organize the input and output data flow in an XML database. Among its most important advantages are modularity and separation of concerns. Each component is dedicated to a specific task and does not depend on the rest of the system. Furthermore, the framework provides two more “intelligent“ modules for input and output, which based on configurations can manage the workflow themselves. The proposed solution offers a unified way to control the data processing in a system and can be used as a blueprint by its design.

5. BaseX: Improving the Input and Output

In this chapter we will demonstrate how the above defined framework can be integrated into an existing XML database, namely BaseX[BAS]. For this purpose we are going to take a closer look at its current input and output implementation and see what advantages and disadvantages it has. Based on this we will continue with a proposal how the generic framework can be used to make this functionality more consistent and flexible.

5.1. Preliminaries

5.1.1. Overview

We start with an overview of the input and output in BaseX. In Chapter 3 it was already presented which data channels exist and which formats are supported by them. A closer observation of Table 3.3 leads to the conclusion that the GUI, command line, REST and XML:DB interfaces have a lot in common when it comes to input. The reason for this is that the actions from the user interface, the REST requests and the XML:DB methods all call in the end the logic which underlies the commands. The same is true for WebDAV, too, but due to its nature and partially to the current input and output implementation in BaseX, it allows only import of XML. Any non-XML data is stored as binary. Regarding the output, again one and the same functionality is used by these channels but by XML:DB and WebDAV a limitation comes, which is related to their interface definition. As far as XQuery is concerned, there the flow is different and relies directly on the XQuery processor within BaseX. Figure 5.1 shows how is generally organized the input and output. As it can be seen, there are two "main" data channels, which have to be considered – the command interface and the XQuery engine. Although the fact that the command functionality is reusable can be counted as an advantage, there are many existing inconsistencies which are hidden behind it. In the next sections we will examine the reasons for them.

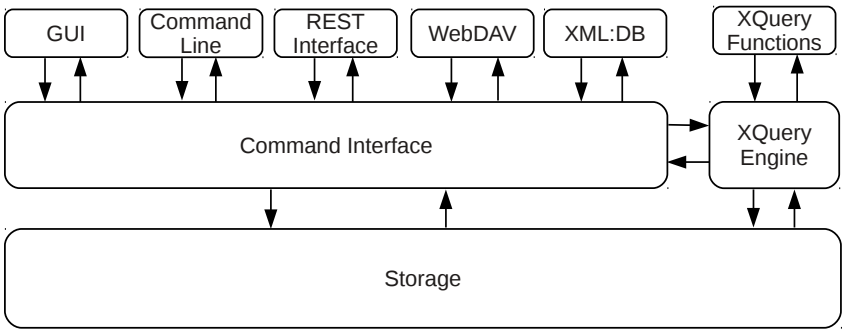


Figure 5.1.: BaseX Input and Output: Overview

5.1.2. Storage and XDM

Since the ultimate goal of parsing is to bring data to the database-specific XDM representation, we start with a few preliminaries on the classes in BaseX which care for the storage of data and its representation as an XDM instance.
BaseX is able to create both persistent and non-persistent databases. The logic for the first ones is encapsulated by the class `DiskData` while for the latter is used `MemData`. `Data` is their generic abstract parent. No matter disk-based or in-memory based, a database instance is always constructed using a dedicated interface for this purpose, namely `Builder`. Here again it can be distinguished between a builder for persistent databases – `DiskBuilder` and such for non-persistent ones – `MemBuilder`. Figure 5.2 gives an overview on the storage classes in BaseX.

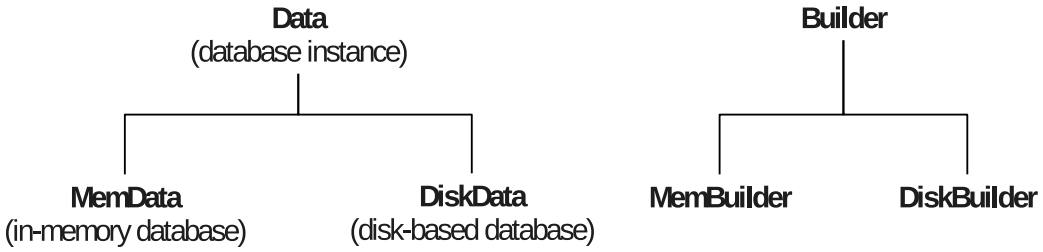


Figure 5.2.: BaseX: Storage Classes

The XQuery Data Model is represented in BaseX with a hierarchy of classes corresponding to the various items. What is more important for our further investigation, however, is that basically there are two main types of nodes – disk-based and main memory based. The classes corresponding to them are `DBNode` and `FNode` respectively. The XQuery functions in BaseX work internally mainly with `FNode`, `Item` and their inheritors(exception here are the functions which execute database operations). The command interface usually uses `DBNode` instances.

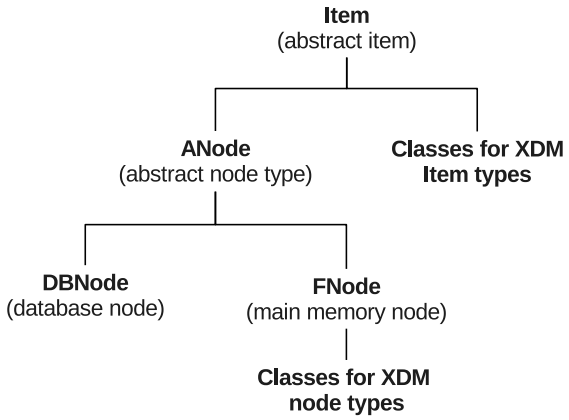


Figure 5.3.: BaseX: Nodes

5.2. Current Implementation

5.2.1. Input

5.2.1.1. Data Sources

We start with describing how data arrives in the database. Currently BaseX offers three types of input – in the form of a byte array, file or directory on a local machine, and a resource, which can be addressed using an URL. This separation is consistent and has a lot in common with the examples given in Listings 4.1 and 4.2. However, along with the necessary logic for retrieving the data, determining its content type and whether it is a file, an archive or a directory, there is also functionality which does not fit that well with the main purpose of these classes. Thus, for example, there are methods for copying a resource, for renaming it, for determining the database name. In order to avoid this superfluous code we will encapsulate in the relevant `DataSource` implementations only what is needed.

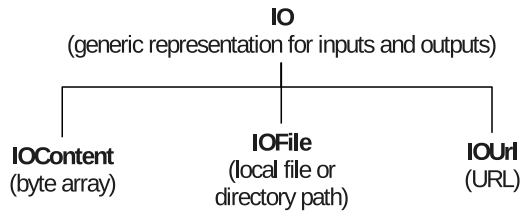


Figure 5.4.: BaseX: Input and Output Classes

5.2.1.2. Parsing

In this section we will have a look at the parsers which are currently available in BaseX. Figure 5.5 gives an overview of their class hierarchy.

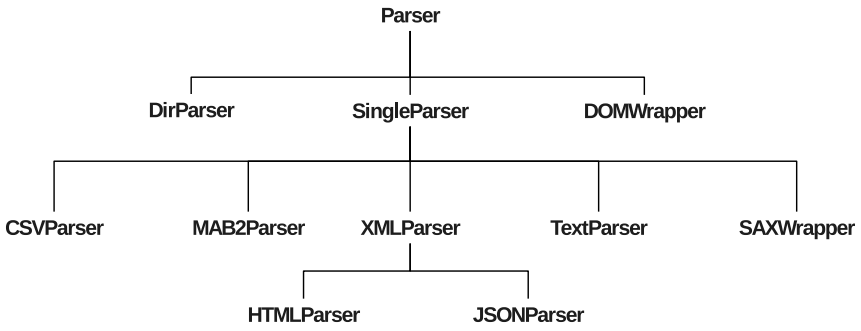


Figure 5.5.: BaseX: Parsers

According to the content type of the parsed resource they can be categorized into XML parsers and non-XML parsers. BaseX uses altogether three XML parsers – a built-in one, Java's SAX parser, which is wrapped by the `SAXWRapper` class, and a DOM parser wrapped by `DOMWrapper`. The first two are used most often. The latter one works exclusively with DOM instances, which are used only by the XML:DB API. According to the structure of the parsed resource, i.e. is it a file or is it a collection of files – directory or archive, there are two parsers – `SingleParser` and `DirParser` respectively. As it can be seen most of the parsers inherit the `SingleParser` class. This logically leads to the conclusion that they operate in a common way, which is generally true. What unites them is the usage of a `Builder` instance directly in the parsing process. They always read sequentially the content, which is taken from the input stream of a file (in case of `IOFile`) or an HTTP connection (in case of `IOUrl`), and send events to the builder,

which itself constructs a `Data` object corresponding to the XDM representation for the coming data. Figure 5.6 gives a general overview of this process.

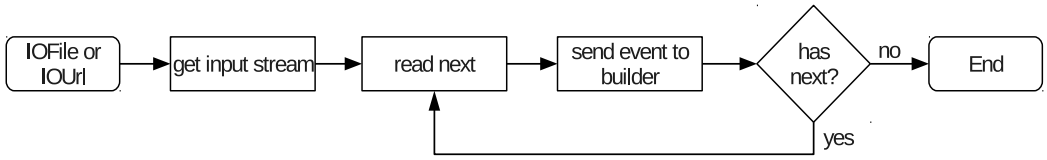


Figure 5.6.: BaseX: Parsing via sending events to a builder

However, not all non-XML parsers implement this event-based approach, which results in an inconsistency in the overall way data is parsed. Thus, for example, `JSONParser` and `HTMLParser` are not direct descendants of `SingleParser` and there is a reason for this. A closer look at their implementation reveals that what they actually do is to first parse the data, then cache it into a new `IOContent` instance and finally handle it to the built-in XML parser in order its builder to construct the XDM internal representation. In the case of `JSONParser` the process is even more complicated since first an instance of `ANode` is created, which after serialization to XML is passed to the XML parser.

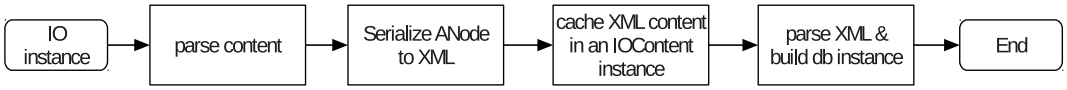


Figure 5.7.: BaseX: JSON parsing in case of database creation

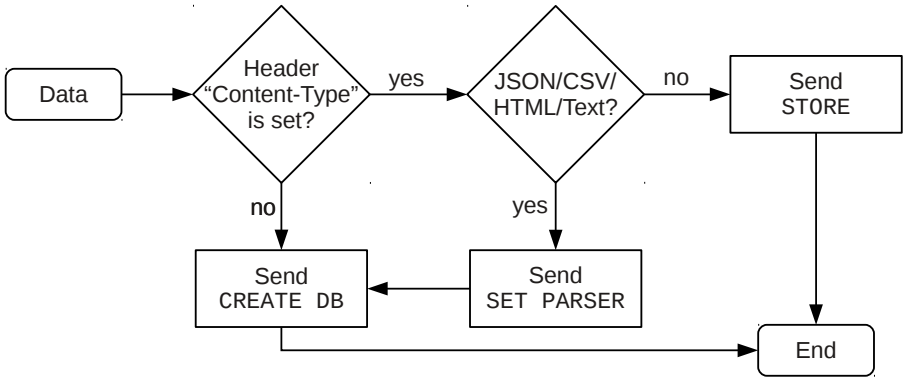
Clearly, using directly the `Builder` class to parse non-XML data is the better approach. The reason for this is that it reads the incoming content directly from the input stream of the source. Thus, parsing large resources cannot cause lack of main memory. A disadvantage, which can be avoided by some rethink of the implementation, is that currently this way of parsing produces `DBNode` instances while the XQuery functions in BaseX work mainly with `FNode`. That is why there are no XQuery functions for pure parsing available at present, which correspond to the above described parsers. This situation can be recognized in Table 3.3 where it is visible that the formats supported for input by the GUI,

the commands and the REST API are not supported in the case of XQuery. Only JSON makes an exception and the reason for that was explained above. The good about the JSON parsing approach is that it is more consistent because it uses the `JSONConverter` class, which is dedicated only to converting JSON to XML and nothing else. This makes it more flexible and reusable. However, the disadvantages coming with this are more complex processing in case of database creation and what is worse – parsed content is cached which makes main memory a bottleneck.
Now, when we have an idea how operate the parsers for single resources, we can step one level above in the input process and see how they are actually called. BaseX is able to work both in local and in client-server mode. In the first case the possible interfaces for interaction with the system are GUI, command line and XML:DB. All three of them use the command interface internally when it comes to creating a new database or adding documents to an existing one. Starting point (except by XML:DB) is the directory parser `DirParser`. Figure 5.8 gives a generalized view of the process.

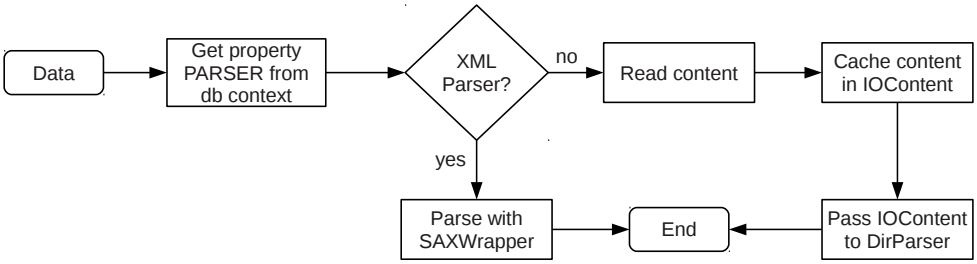
Figure 5.8.: BaseX: Creating a database in local mode

As it can be seen whenever data enters the database, a corresponding `IO` instance is created which is then passedto a `DirParser`. The latter one calls the relevant single parser. One disadvantage of this approach is that although the logic in `DirParser` is mainly responsible for traversing a file hierarchy, it is classified as a content parser along with the single parsers, which process individual files. The result of this is that `DirParser` accepts parsing properties for a single content type and thus only collections consisting of files with one and the same content type can be imported. If other files are present, they can be stored as raw data. This is a shortcoming because a user cannot import a directory full of HTML and CSV file at once, for example. First they have to import the HTML files and then - the CSV or vise versa. One way to improve the current situation may be to isolate the logic for walking through a file hierarchy in a separate class, which at least is not classified as a parser. Furthermore, it can be checked right at the beginning of the input process if the source is a file or directory in order to decide whether to call directly a corresponding single parser or to use the `DirParser`.
When BaseX is used in client-server mode there are three ways to communicate with it – via commands sent through the BaseXClient interface, via REST requests and through a WebDAV client. The first approach is trivial since with it a user can directly execute the available commands through a BaseXClient and they result in what was already described above. This is not the case with the RESTful API and WebDAV. They also use the commands in the end but before that some preprocessing steps take place which influence the further execution. Figure 5.9a shows what are the general steps when sending

a REST *Put* request and what happens when the `CREATE DB` command is executed on the server.



(a) BaseX: REST Put Request



(b) BaseX: `CREATE DB` Command

Figure 5.9.: BaseX: Sending a REST PUT request and handling it on the server

An application can make clear what is the content type of the data it sends to the database server by setting the “Content-Type“ header in the HTTP connection. When it is set, the REST API first sends a `SET PARSER` command to the BaseX server in order to announce what parser must be used further. If this header is not set, it is assumed that the data is XML and the default XML parser is used. If the header is set but there is no available parser for the content type, then the data is stored raw in its original format. One problem caused by this workflow appears when non-XML data is sent to the server but the “Content-Type“ header is not set. In this case it is assumed that the data is XML and on the server it is directly passed to the `SAXWrapper` to handle it. Of course, this results in an error since this parser expects exclusively XML data. The second problem

appears when the header is set. It comes from the fact that currently BaseX does not have a functionality for working with stream data and the only remaining option in such a scenario is to cache the incoming content in an `IOContent` instance. Then it can be passed to the corresponding parser. Again, the same problem occurs as by the HTML and JSON parsing in local mode – main memory lack in case of large resources.
By WebDAV the behavior is different – whenever a new database is created or a resource is added to an existing one, it is checked if it is XML. If this is not the case, it is directly stored as raw content. In BaseX raw files are kept in a system-specific sub-directory. Their import has no complex logic since no parsing takes place and the data is directly read from the input stream. This is why this functionality is accessible from all available interfaces.
The last data channel which remained is XQuery. Currently BaseX does not offer any special XQuery functions for parsing data with content type different from XML, except JSON. The reasons for that were already explained above. An exception makes only the `db:add()` function, which works with `DBNode` instances but it is aimed to add documents to a database and not for pure parsing. In this case the parser to be used can be set in the prolog of the corresponding XQuery expression via `declare option db:parser`.

5.2.2. Output

The output in BaseX is organized more consistently in comparison to the input. This can be observed in Table 3.3 – the output formats supported by XQuery, commands, GUI and REST API are almost the same. By WebDAV and XML:DB the situation is different but this is because of their different interface. Figure 5.10 gives an overview of the currently available serializers.

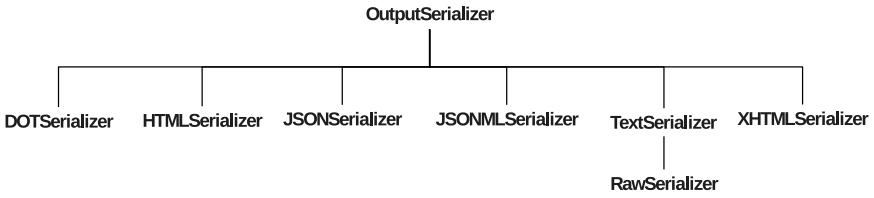


Figure 5.10.: BaseX: Serializers

The good about them is that they have a common interface, which is defined by the `OutputSerializer` class. Furthermore, the logic they encapsulate is organized consistently and handles only the serialization process. This is feasible thanks to the internal XDM representation in BaseX. In the previous section it was explained that it can be

distinguished between disk-based and memory-based nodes(`DBNode` and `FNode` respectively). They, however, have a common parent – `ANode`, which provides an abstract method for serialization. It accepts always a serializer instance as an argument. Thus, where needed this method is implemented. The result is a common way to serialize both XML data coming from the database (`DBNode`) and such which was passed directly to an XQuery function (`FNode` and its descendants).

5.2.3. Options

Currently, in BaseX it is possible to use various input and output options. The input options can be divided into content parsing and directory parsing ones. The first ones specify what is the content of the incoming data as well as properties referring to the corresponding parser. The second are settings indicating how to parse the files in a directory or archive, which is added to a database or from which a new database is created. Among these are flag for skipping corrupt files, flag for adding files which are in sub-archives, etc. The output options in BaseX are available in the form of serialization parameters. They include not only the standard ones but also such that are specific to BaseX. When it comes to output, there are two possibilities – either to serialize a document or an XQuery result or to export all documents in a database. In both cases the available serialization parameters can be used to control the process.

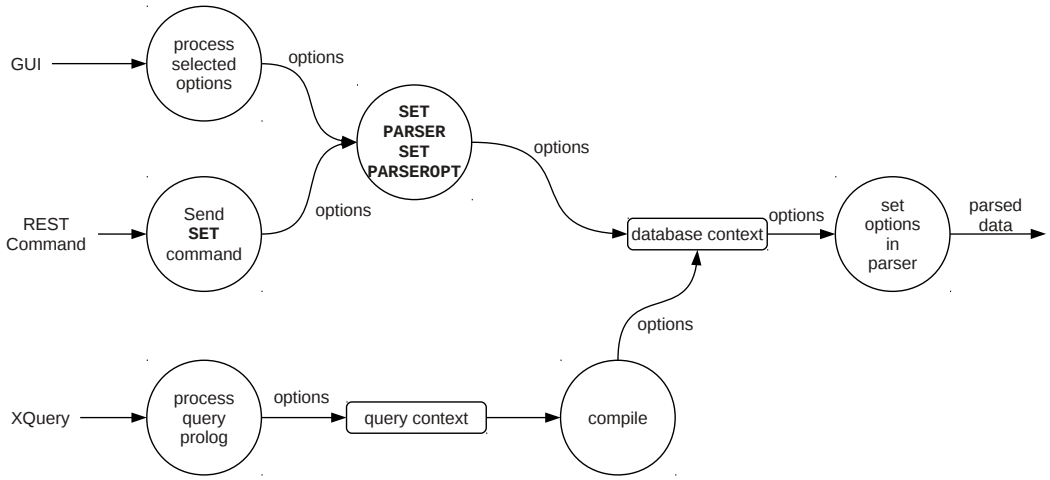


Figure 5.11.: BaseX: Setting Parser Options

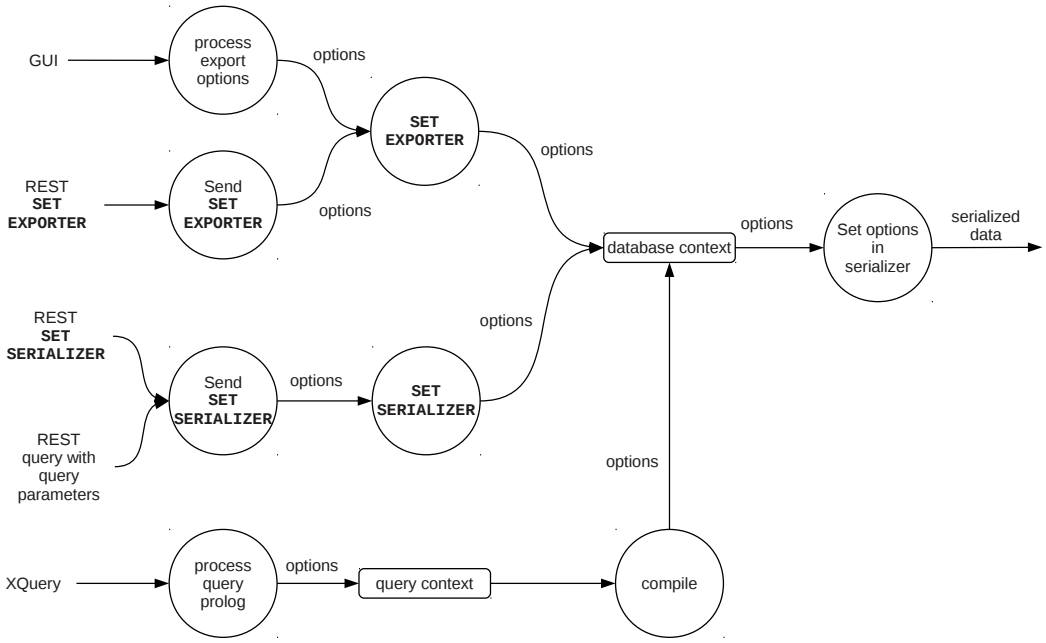


Figure 5.12.: BaseX: Setting Serializer Options

The input and output options in BaseX can be set in diverse ways depending on the used channel. Their final destination, however, is always the same – the database context. From there they are taken and passed to the relevant parsers and serializers, which use them in the data processing. As it is impossible to specify options through XML:DB and WebDAV, only the rest three channels are considered – GUI, REST API and XQuery. Figures 5.11 and 5.12 show how the input and output parameters are managed by them. In the GUI the user specifies the necessary options and the underlying logic uses the `SET` command to set them in the database context. Through the REST API an application can set parser options using the `SET PARSER` and `SET PARSEROPT` commands. In case of output, again the `SET` command can be sent directly to the server or the needed serialization parameters can be set as query parameters and the REST API sends them via the `SET SERIALIZER` command to the server. When XQuery functions are executed, the preferred options can be set in the prolog of the XQuery expression. In that case the XQuery processor sets them first in the query context and after that, when the query is compiled, they are set in the database context. The database context always holds a reference to a `Prop` instance, which assembles properties used throughout the whole project. This instance has fields for parser, serializer and exporter options, which are set in the above described cases.

5.3. Improvement

In the previous section we made a short analysis of the current input and output data flow in BaseX. Both advantages and disadvantages have been found. Here we are going to see how the existing functionality can be used to implement the generic architecture presented in Chapter 4.

5.3.1. Input and Output Management

Presently, BaseX offers various options for managing the input and output of data. Some of them refer to concrete parsers and serializers while others are related particularly to importing data from directories and archives. All of them can be specified through almost all available data channels. Figure 5.13 gives an overview how a user can set diverse input options through the GUI. Although the current approach covers most of the user's needs, it is not consistent enough and can be confusing. In this section we will show how this disadvantage can be eliminated by introducing a completely new way for manipulating input and output options.

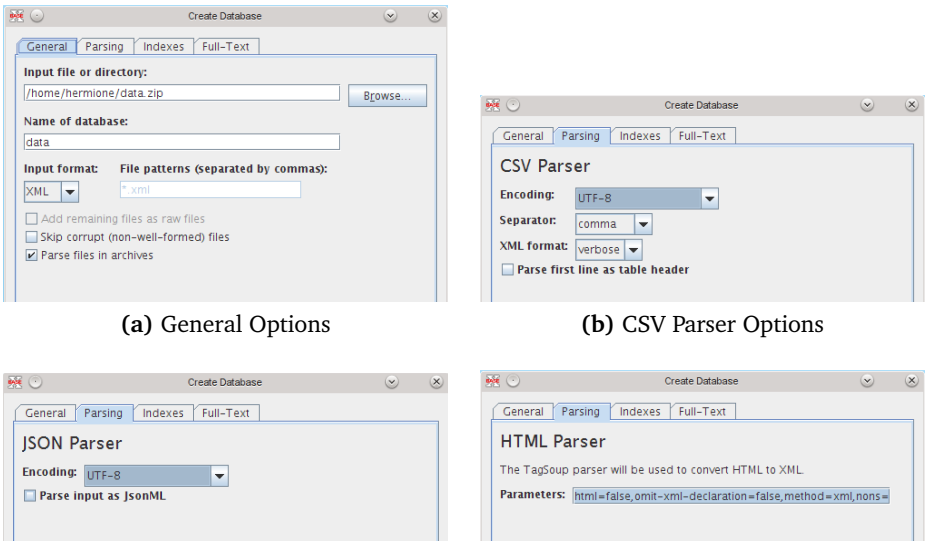
5.3.1.1. Configurations

In Chapter 4 was presented the concept of input and output configurations, which we are going to apply in BaseX. As it was explained there, the way content types are treated by input and output can be easily defined using XML. This is a convenience when it comes to XML databases and XQuery engines because one way to manipulate the content of the above mentioned XML files is to keep them in a special administrative database within the system.

In this way they can be easily accessed, queried and updated by using XQuery statements. We will follow this approach in BaseX and will introduce a dedicated system-specific database called *system-config*. Four documents will initially always reside in it:

- `input-options.xml` specifying how each content type shall be processed and the corresponding parser
- `parser-options.xml` specifying the avaialable options for each parser
- `output-options.xml` specifying how the serialization to a given content type shall happen and the corresponding serializer

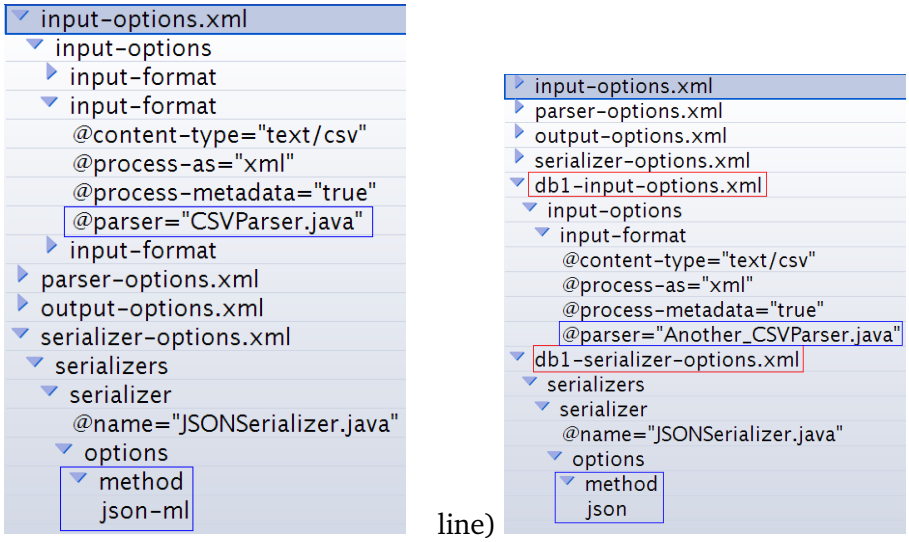
- `serializer-options.xml` specifying the available options for each serializer



(a) General Options **(b)** CSV Parser Options

(c) JSON Parser Options **(d)** HTML Parser Options

Figure 5.13.: BaseX: Input and Parsing Options



line)

(a) Global Options **(b)** *db1* specific options

Figure 5.14.: BaseX: Folder View of *system-config*

Of course, configuring a content type for a specific database will also be possible. In this case in the *system-config* database will be created a version of the corresponding document starting with the name of the database and holding only the difference with

the original one. Figure 5.14 shows a simple example. Thus, BaseX will have a global configuration represented by the four XML files from above and many database-specific ones. The global configuration will be applied whenever documents enter a database for which their content type is not specifically configured.

5.3.1.2. Configuration Management

Having the input and output options in an XML database is quite convenient since they can be easily manipulated with XQuery. However, an XQuery module which cares only for configuration-related tasks is even more convenient as it can save a lot of work. Furthermore, it can be implemented entirely in XQuery. We define such a module for BaseX. The functions included in it are listed below:

- **cfg:list-input-content-types**
  `cfg:list-input-content-types() as` *element(input)**
  This function returns all supported input content types.

- **cfg:input-content-type**
  `cfg:input-content-type($content-type as` *xs:string*`)`
  `cfg:input-content-type($content-type as` *xs:string*`,`
  `$database as` *xs:string*`) as`
  element(input)
  This function returns the input options for the content type `$content-type` in the global configuration. If a database name is specified, the settings in the database-specific configuration are returned.

- **cfg:parser-options**
  `cfg:parser-options($content-type as` *xs:string*`) as`
  element(parser)
  `cfg:parser-options($content-type as` *xs:string*`,`
  `$database as` *xs:string*`)as`
  element(parser)
  This function returns the parser options for `$content-type` from the global configuration. If a database name is specified, the parser options from the database-specific configuration are returned.

- **cfg:list-output-content-types**
  `cfg:list-output-content-types() as` *element(output)**
  This function returns all supported output content types.

- **cfg:output-content-type**
  `cfg:output-content-type($content-type as` *xs:string*`)`

```
cfg:output-content-type($content-type as xs:string,
                        $database as xs:string) as
                        element(output)
```

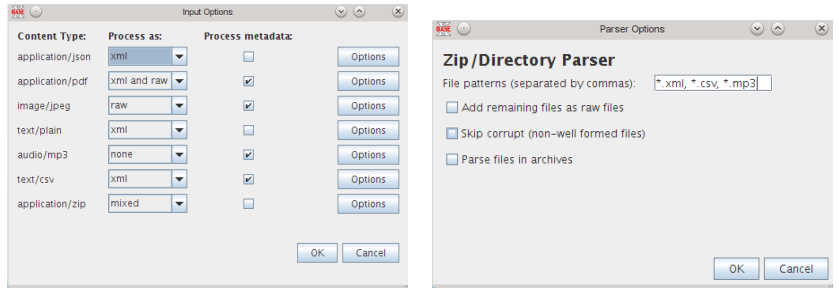
This function returns the output options for the content type `$content-type` in the global configuration. If a database name is specified, the settings in the database-specific configuration are returned.

- **cfg:serializer-options**

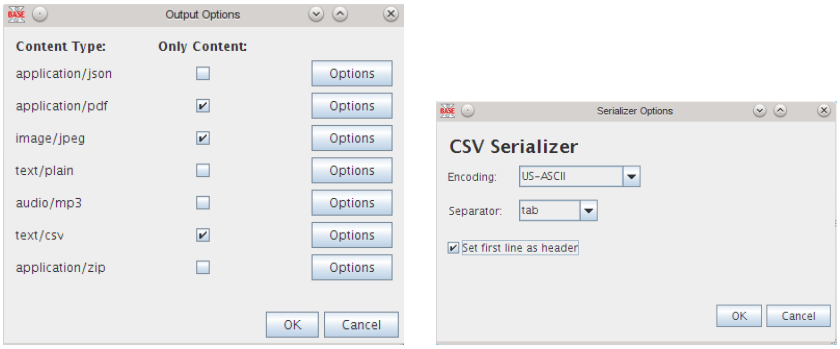
```
cfg:serializer-options($content-type as xs:string) as
                       element(serializer)
cfg:serializer-options($content-type as xs:string,
                       $database as xs:string)as
                       element(serializer)
```

This function returns the global serializer options for `$content-type`. If a database name is specified, the serializer options from the database-specific configuration are returned.

Apart from offering an intuitive way to manage the input and output configurations, the defined XQuery module can be easily used through most of the other data channels in BaseX since the command interface is able to communicate with the XQuery engine via the `XQUERY` command. It will not be possible to control the options through WebDAV or XML:DB because their interfaces do not allow such actions but the important thing is that whenever documents enter a database through one of them, the global or the corresponding database-specific configuration will be applied. This is possible as now all data channels work with a `DataSource` instance which content type can be recognized and based on it the relevant configuration can be detected as shown on Figure 4.7.
What may be interesting is how the input, output, parser and serialization options will be presented and managed by the GUI in BaseX once the above concept is applied. Figure 5.14 shows some sample mockups.



(a) Input Options **(b)** Zip/Directory Parser

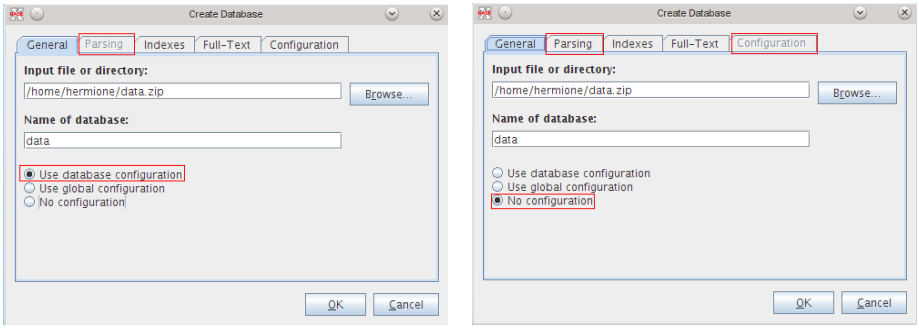


(c) Output Options **(d)** CSV Serializer

Figure 5.15.: BaseX: New Input and Output Options

The current BaseX GUI can be easily extended with a menu for configuring the global input and output options. The *Options* Button against each input/output content type navigates to the corresponding parser/serializer options. Of course, this visualization is not the most optimal one since it will become quite inconvenient if a wide range of formats is supported. Maybe the easiest solution to this problem is to add a search field which will allow to look for and directly navigate to a concrete content type. In case of database-specific configuration the very same visualization can be used but it will be accessible through an additional tab *Configuration* in the dialog for creating a new database. Whenever the settings for a given content type are updated in the GUI, the corresponding XQuery function from the above module will be called to retrieve the relevant record from the configuration and the changes will be persisted in the *system-config* database using an XQuery update statement on the returned record.
It is important to note that the new way for controlling the data processing in BaseX will not replace or eliminate the already existing one. It can be viewed more as a useful addition to it. Specifying the options through the GUI, command line, XQuery prolog or REST will still be possible but it will make sense to use it only when it is explicitly stated that no present configuration shall be used. In the GUI this can be done as shown on Figure 5.16. “Underneath“, the three settings can be managed by the `SET` command and three new corresponding options for them. Thus, they can be also controlled through

the REST API.



(a) Use database configuration **(b)** Do not use any configuration

Figure 5.16.: BaseX: Configuration Usage

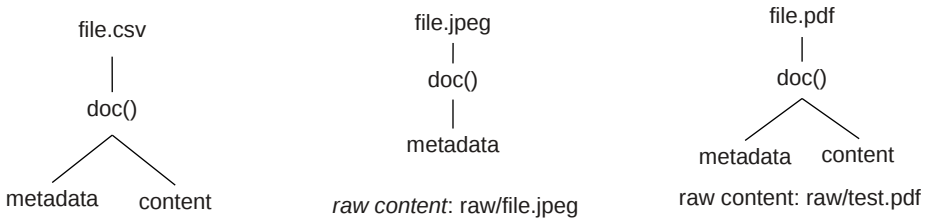
As it was explained earlier, no matter how the parser and serializer options are specified, they arrive at the end in the `Prop` instance in the database context from where they can be reused. This approach has a good workflow but is not consistent enough because these options are not isolated in any way from the rest of the options used in the system. Apart from them the `Prop` class holds also database-specific ones, operating system-specific, etc. The new way for managing options will eliminate this inconsistency. The database context will be extended with two more fields corresponding to the input and output options which are to be used in the current session. These will be instances of the `InputConfiguration` and `OutputConfiguration` classes. They will be initialized only in case it is specified that no existing configuration shall be used. Otherwise, the settings will be read dynamically from the configuration.

5.3.2. Content and Metadata

Currently, BaseX does not support separate storage of metadata and content. In previous versions there were parsers for specific metadata containers such as ID3 and EXIF but at that time binary data could not be kept in the database and thus the content of such files was stored on the file system, “outside“ the database. In this section we are going to see how we can use BaseX in its current state to provide also functionality for working separately with metadata and content.

5.3.2.1. Storage

BaseX stores binary content as raw in a special sub-directory, called *raw*, which resides within each database. Clearly if metadata is to be stored, then it shall be kept as XML since otherwise it is not useful. A natural question which arises when we talk about separation of content from metadata is how will be kept the relation between them once they are stored in the XML database or in other words how will it be known which metadata passes to which content. Since we deal with two types of data – such that can be converted to XML and such that cannot, the way metadata is maintained for them is also different. When it comes to XML, BaseX is flexible and allows things from which we can benefit. Thus, for example, presently it is able to store more than one root node under a document node and this solves half of our problem. When data is to be saved as XML, its metadata can be kept as a second root node under the corresponding document node. Thus for each document there will be two root nodes – one representing the content and another one – for the metadata. In case of binary data, only the metadata node will be present. The content itself will be stored in the *raw* sub-directory but with exactly the same path as the corresponding XML document holding the metadata. As in the database cannot reside two XML documents with one and the same path ,as well as in a directory, the path will serve as a key for finding matching binary content and metadata. Figure 5.17 shows how content and metadata will be stored in the three different cases allowed by the input configuration.



(a) Process as: XML **(b)** Process as: raw **(c)** Process as: mixed

Figure 5.17.: BaseX: Content and Metadata

5.3.2.2. Metadata

The notion of metadata can have sometimes an ambiguous meaning since diverse properties can be treated as metadata. Furthermore, some files have specific metadata containers while by others it can be determined based on analysis of the content. Here we are going to define how a `metadata` element shall look like in BaseX.

Generally metadata can be classified in two types – such that is file system-specific and each file, no matter what is its content type, can have it. Date of last modification or size are examples for such data. The second type is the data which is determined by the content type. Thus, for instance, in a JPEG file the metadata is in EXIF format and contains such information as model of the used camera, matrix sensitivity, focal length, etc. Based on this division, we can define a simple structure for the `metadata` element. First, it shall have an attribute for the content type of the data. Second, there must be a child holding file-system information. Let us call it `fs-info`. Last, the content type-related metadata must be listed as key-value pairs where key will be the name of the element and value – its text value. Thus, for example, the `metadata` element for a JPEG file will have the following structure:

```
<metadata content-type='image/jpeg'>
  <fs_info>
   <size>902KiB</size>
   <last-modified>2011-05-18</last-modified>
   <original-path>/home/pictures/pic.jpeg</original-path>
  </fs-info>
  <make>EASTMAN KODAK COMPANY</make>
  <model>KODAK DX6490 ZOOM DIGITAL CAMERA</model>
  <lens/>
  <aperture>F2.8</aperture>
  <focal>6.3mm</focal>
  <sensitivity>140/1ISO</sensitivity>
</metadata>
```

Content Type	Metadata
application/xml	encoding
text/plain	encoding, language, byte order mark, end of line
text/csv	header, end of line, record delimiter, column separator
application/pdf	various document, properties such as author, title, creator,etc.
application/vnd.openxmlformats-officedocument. wordprocessingml.document(docx)	template, pages, words, paragraphs
audio/mp3	ID3 tags such as, artist, year, album, etc.
video/avi	RIFF tags such as genre, location, starring, etc.

Table 5.1.: Examples for metadata

Table 5.1 contains some more examples on metadata. For some content types metadata is clearly defined and kept in a specific container, e.g. ID3 tags by MP3, RIFF tags by AVI

and the folder *docProps* in case of *docx* files. In other cases determining what is metadata is quite subjective and can strongly depend on the user's and application's needs. Thus, for example, by XML one can treat the encoding as metadata if it is necessary later by the serialization of this data to their original encoding. By CSV the header line can be considered metadata and kept separately in order to prevent a user from treating it as content.

5.3.3. Input

5.3.3.1. Data Sources

In section 5.2.1.1 we acquainted ourselves with the three classes in BaseX responsible for the input and output of data – `IOFile`, `IOUrl` and `IOContent`. For now they are able to do most of the necessary work when it comes to input but a drawback which was observed is that currently they do not offer a way to work directly with stream data unless it is cached. This becomes even a bigger problem when we have to implement the generic architecture because the parsers in it work with an input stream. Since the `IO` class delivers functionality which is used in other places in the project (for example when building a database) except within the parsers, it is not a good option to work directly with `java.io.InputStream` within the new parsers because needed information will not be present. Thus, the logical solution of the problem is to add one more inheritor to the `IO` class in BaseX which works only with stream data. Let us call it `IOStream`.
With the `IOStream` class the picture is already complete and we can continue with the real part of our work, namely the implementation of `DataSource`. This will not be a difficult task because most of the needed functionality is present. Thus, the simplest approach would be to create a class encapsulating the logic of the classes for input and output, which is necessary to implement the methods of `DataSource`. The missing functionality will have to be added. Table 5.2 and Figure 5.18 show how the existing methods of the `IO` inheritors can be reused and how the new class hierarchy for input and output will look like.

DataSource Method	IO Method
`getContentType()`	to be implemented
`getName()`	`name()`
`getData()`	`inputStream()`
`getDataSize()`	to be implemented
`getEntries()`	to be implemented
`isCollection()`	`isDir()`, `isArchive()`

Table 5.2.: `DataSource` Methods vs. `IO` Methods

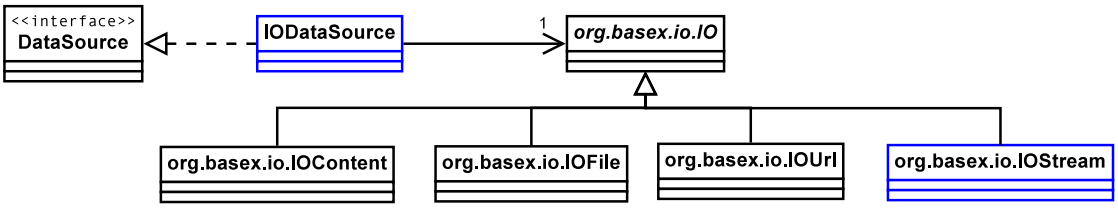


Figure 5.18.: BaseX: `DataSource` Implementation

5.3.3.2. Parsing

The next step is to see how the existing parsing functionality in BaseX has to be modified so that it can be applicable in the generic architecture. As it was already presented most of the parsers work hand-in-hand with a `Builder` instance by sending events to it. Although this is a good approach, it makes the `Parser` and `Builder` classes too dependent on each other. Figure 5.19 shows this dependency more clearly. As it can be seen a `Builder` instance is always created with a `Parser` although the parser is the one that uses the builder to convert data to XML. The problem appears when it comes to importing data from "complex" data sources, i.e. archives and directories. In this case, as it was shown in Figure 4.10, for each data source will be instantiated the corresponding parser according to its content type. In case of directories/archives with too many files with diverse formats this will result in creating too many `Data` instances (one for each file) which may lead to lack of main memory. However, with the current implementation this is not a problem since a `DirParser` instance uses always just one builder and consequently creates just one `Data` instance for all included files.

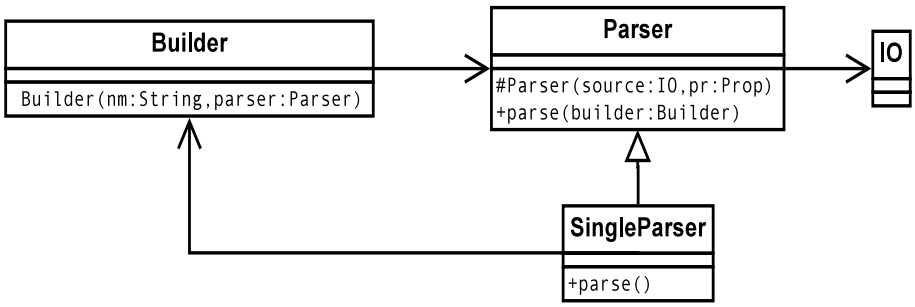


Figure 5.19.: BaseX: `Builder-Parser` Relationship

Thus our task is to adapt the existing logic in such a way that we can avoid this problem when parsing directories and archives with the new input functionality. After a closer look at the implementation of `Builder` it becomes clear that currently it is responsible for two main tasks– preparing and finalizing the internal representation and listening to

events coming from a parser. For the first task it needs information about the input which can be taken only from the `IO` instance referenced by the parser. The second one is absolutely independent from the parser. This leads to the conclusion that a `Builder` can be created using only an `IO` instance. Furthermore, since a parser always uses only the logic for event listening, it can be decoupled from the `Builder` class and called separately. In that way we can achieve a more flexible and modular implementation, which will be able to serve our needs. Figure 5.20 shows the corresponding UML diagram. As it can be seen, there is already a dedicated class for event listening – `ParserListener`. Since the events are interpreted in a different way when a disk-based and when a memory-based database is created, there are two separate listeners for each of the cases. What is more important, however, is that a `Builder` now can take the necessary information directly from an `IO` instance and does not need a `Parser` to be instantiated. All parsers now will send events not to a builder but to an event listener.
We can continue with the parsers themselves. Since the current ones in BaseX work quite well and are consistent enough, the only thing we have to consider is how to reuse them by the implementation of the generic architecture. This does not appear to be a challenge when we compare what a BaseX parser needs in order to function properly and what the generic parser from Chapter 4 needs for its goals. Both have something in common and this is the input. Since in BaseX the input is represented by an `IO` instance, this will be the first argument in the constructor for the abstract `Parser` class. Two more things necessary for a BaseX parser are parsing properties, which have to be applied and a listener, which will listen to the events sent from the parser and write in the corresponding database. The parsing logic is already present and can be reused within the `parseContentAsXml()` method. As far as the metadata is concerned, it will be parsed to XML, too. Raw content will always be returned as an input stream from where it could be directly read. Based on these reflections the new parser hierarchy in BaseX will look like as shown on Figure 5.21.

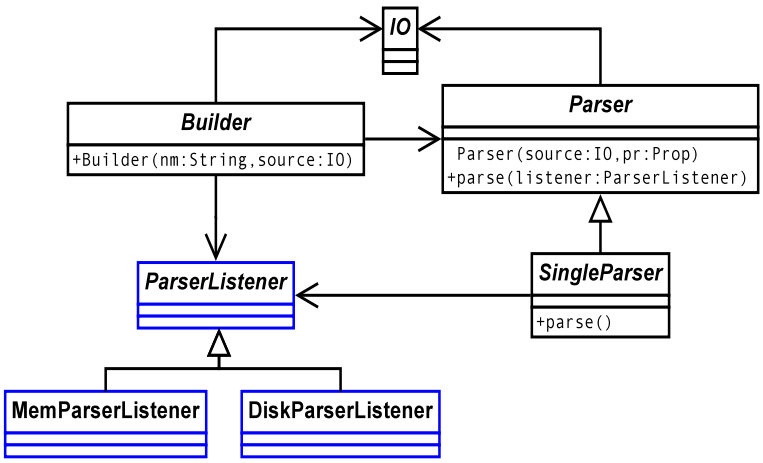


Figure 5.20.: BaseX: New `Builder`-`Parser` Relationship

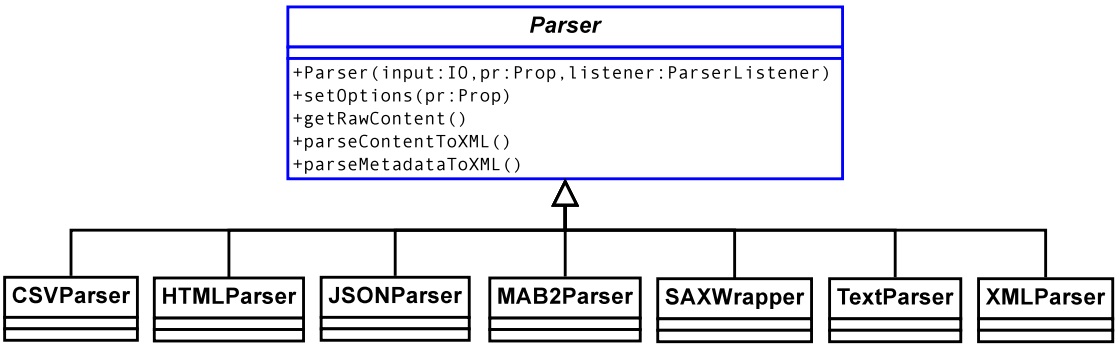


Figure 5.21.: BaseX: New parser hierarchy

With the `DataSource` and `Parser` implementations now the input functionality in BaseX has a more ordered and logical flow – there is unified access to data sources no matter what is their nature – a single file, a directory or a stream, and furthermore – parsers are dedicated to one single task, namely the conversion of non-XML data to XML. One last feature that is needed in order to achieve the architecture from Chapter 4 is the "automated behavior" mentioned there, which is delivered by the `InputProcessor` class. Since it is based on the interplay between data sources and parsers, it can be already realized in BaseX as shown with the flowcharts on Figure 4.9 and Figure 4.10. Its UML diagram is depicted on Figure 5.22.

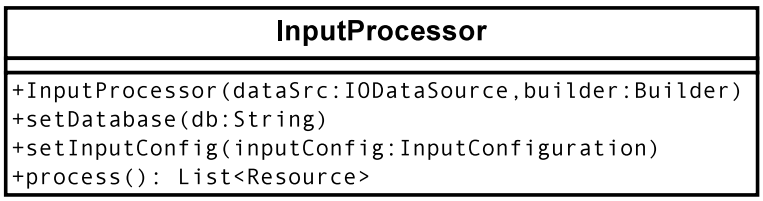


Figure 5.22.: BaseX: Class `InputProcessor`

As it can be seen, `InputProcessor` is always instantiated with an `IODataSource` and a `Builder`. The latter is needed to prepare and finalize the database instance and to provide a `ParserListener`, which has to be passed to the corresponding parsers. `setInputConfig()` is used to set an input configuration containing input options set by a user or an application. In case it is stated that an already existing configuration shall be used, then this method is not called. The `process()` method does the actual data processing. Based on the input configuration, it decides how to process it – as XML, as raw or as mixed and what to parse – content, metadata or both. Once this is determined, the relevant parser is instantiated and the corresponding methods are called. Figure 5.23 summarizes the new workflow within the `CREATE DB` command by processing of a single file.

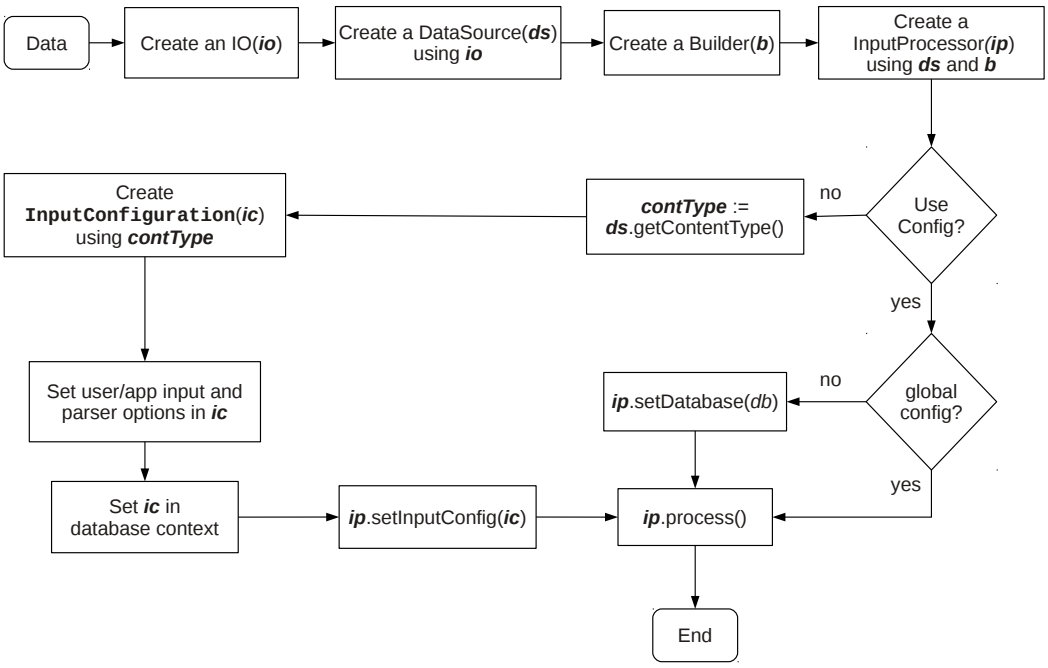


Figure 5.23.: BaseX: New `CREATE DB` implementation

The `process()` method of `InputProcessor` returns `Resource` instances, which hold the already parsed data. Since this functionality shall be used also by the XQuery engine, it is essential to get the metadata and XML content from a `Resource` in a form acceptable by the XQuery implementation, too. This is why the methods `getXMLContent()` and `getMetadata()` will return an instance of `ANode`.
With this last piece from the generic input architecture, it is possible to extend BaseX even with a more general XQuery function for input parsing, similar to the `resource()` function mentioned in Chapter 4. In that way the functionality shown on Figure 5.23 will be applicable in all available input "areas" in BaseX. Furthermore, since we have achieved modularity by isolating the parsing logic, now it is possible also to define finer granulated XQuery functions for parsing specific content types, e.g. `parse-csv()` for CSV data or `parse-html()` for HTML data. This was not possible with the previous state of the input functionality.

5.3.4. Output

In the end we are going to discuss how the output in BaseX can be modified in order to be in accordance with the generic architecture. In this section we will explain some changes which shall be introduced in the current serializer classes and we will concen-

trate especially on the improvement of the `EXPORT` command.

5.3.4.1. Serializers

A closer look at the current serialization logic in BaseX shows that the changes which it has to underlie in order to be applicable in the context of the generic framework are not that many. Presently, each serializer class is initialized with an output stream and serialization properties. This corresponds to some extent to the way a generic serializer is created. By the latter, however, always a `Resource` instance is passed as a parameter. The main purpose of the `Resource` class is to serve as a convenient encapsulation of content and metadata after they have been parsed. Such an encapsulation is needed in order to maintain the relation between these two. In BaseX, however, we can leave this representation aside because content and metadata stay always together under one and the same document node or in case of raw data – the path to metadata coincides with this to content. Thus, the only thing which has to be done is to add two new abstract methods to the `OutputSerializer` class – `serializeOnlyContent()` and `serializeContentWithMetadata()`. They will accept an `ANode` instance as an argument and will be implemented by the diverse specific serializers. The logic in `serializeOnlyContent` is straightforward – either the `serialize()` method of `ANode` is called in case of XML content or the corresponding binary file is read in case of raw content. The workflow in `serializeContentWithMetadata()` is depicted on Figure 5.24.

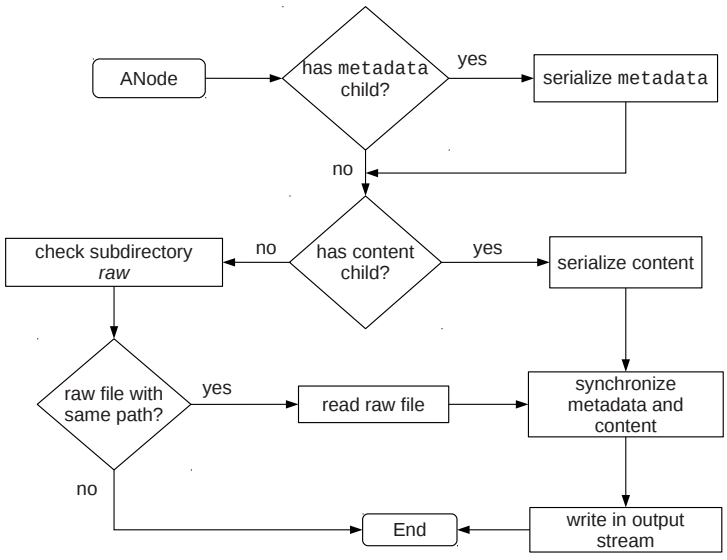


Figure 5.24.: BaseX: `serializeContentAndMetadata()`

5.3.4.2. Serialization and Export

Currently in BaseX there are two ways to output data – by serializing it and by exporting it. Serialization can be applied both on data stored in the database and on data which is passed to an XQuery function as an XML fragment, for instance. Apart from the standard serialization parameters, BaseX provides its own specific, too. Especially the `method` parameter is extended with additional values such as JSON and raw. Analogous to the parser options, the serializer parameters can be set either by the `SET SERIALIZER` command or in the prolog of an XQuery expression in case the usage of an existing output configuration is not needed. Another possible way is to provide them as an argument either to `fn:serialize()` or to `file:write()`. Whenever an output configuration is to be used, the output options and serialization parameters will be read from it. The behavior is similar to the one explained for the input.
The second way to output data is to use the `EXPORT` command. It allows to export all resources of a database at once. Although this approach is convenient, it is not flexible enough. The reason for this is that all documents from a database are exported to one and the same format, which is restricted to the values available for the `method` serialization parameter. Currently it is not possible to export a single document or to export documents to their original content type. This comes from the fact that until now the original format of data was not stored in any way. With the new output processing, this can be realized. The component which will care for this work is the `OutputProcessor` class. Its UML diagram is shown on Figure 5.26.

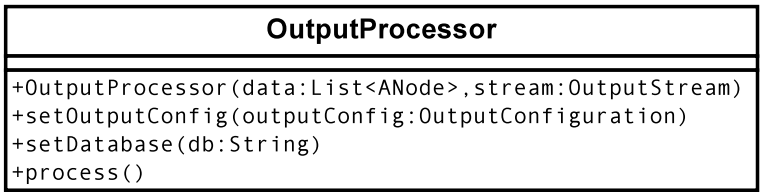


Figure 5.25.: BaseX: Class `OutputProcessor`

As it can be seen `OutputProcessor` is always instantiated with a list of `ANode` instances and an output stream where the result from their serialization will be written. The `setOutputConfig()` method can be used to set an output configuration, which contains output and serializer options defined by a user or an application. If it is not used, the `process()` method reads directly the existing configuration for the target content type and uses the options from it. In order `OutputProcessor` to be usable within the `EXPORT` command, its current implementation has to be a little modified. Presently, when a user wants to export the resources from a database, it is iterated over the *pre* values of the document nodes within it and for each one is called the `node()` method of the corresponding serializer. Binary resources are read from the *raw* sub-directory and written directly into the output stream. This functionality is replaced now with the

`process()` method of `OutputProcessor`. The flowchart on Figure 5.26 shows how it works for a single `ANode` instance.

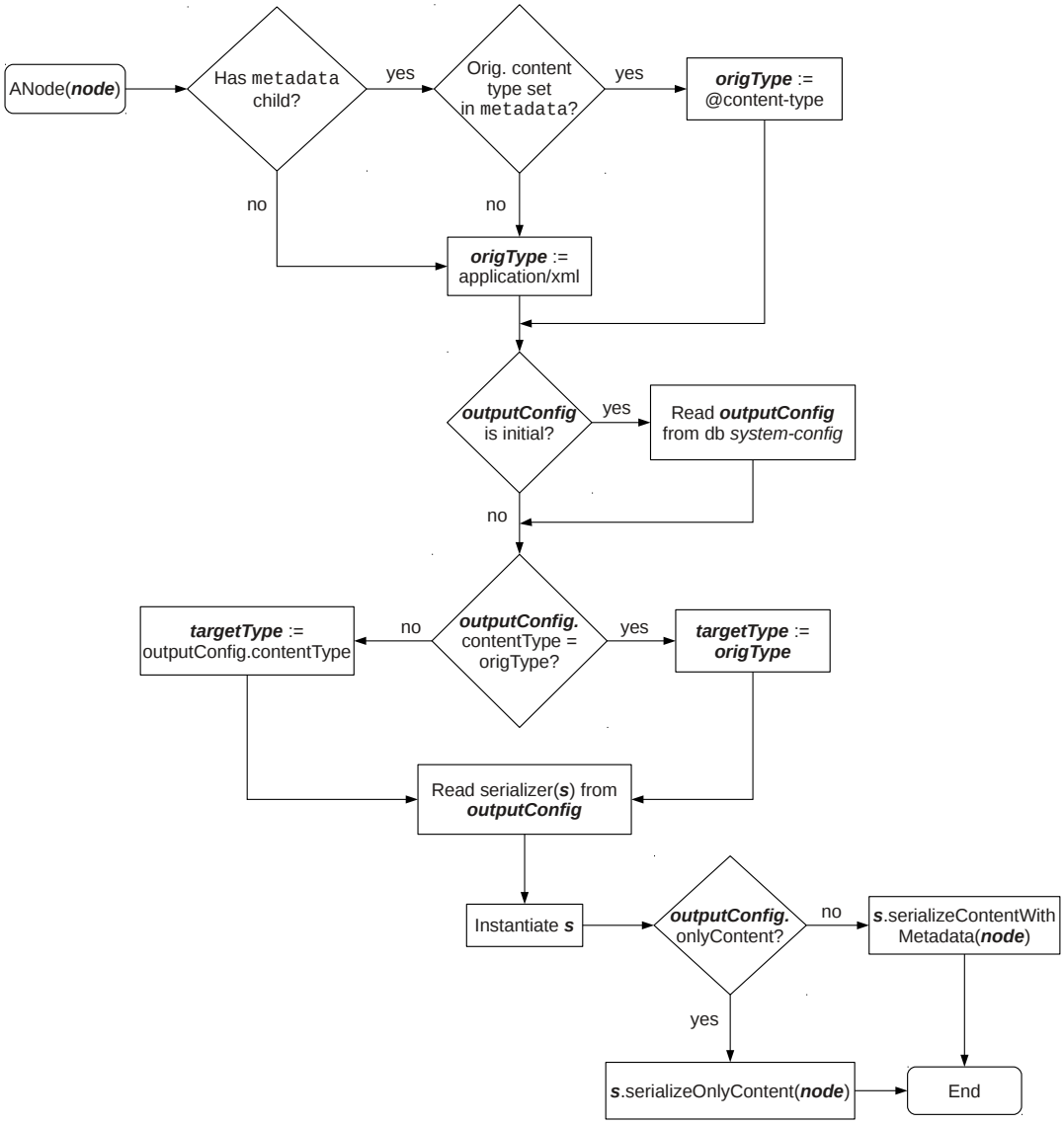


Figure 5.26.: BaseX: Exporting a single resource

The documents are first collected into a list of `ANode` instances and after that this list, together with the target output stream, are passed to the constructor of `OutputProcessor`. If it is specified that no output configuration must be used, the one which is initialized

with the user parameters is read from the database context and set via the `setOutputConfig()` method. Otherwise, it is proceeded with the next step, namely calling the `process()` method, which takes care for the rest of the export. The `EXPORT` command is now able to output the resources from a database in their original format. Of course, this is possible only when this format is present in the metadata. Otherwise, *application/xml* is used as a default output format. Furthermore, the command can be easily enhanced to allow export of single resources identified by their database path.

5.4. Conclusion

In this chapter we demonstrated how the proposed generic framework can be applied in an existing XML database. As a result the input and output in BaseX have been improved. We introduced a completely new approach for controlling the data processing. Since it is entirely based on XML and XQuery, BaseX does not need to be extended with additional logic in order to support it. Another new feature is the separate handling of content and metadata. It brings more flexibility to the existing functionality and expands the usage scenarios for BaseX. Last but not least, parsers and serializers have now clearly defined interfaces and adding support for new content types is straightforward. Furthermore, the input and output processor classes offer a convenient way to handle data independent of its format.

6. Future Work

Purpose of the generic architecture for input and output is to propose a modular architecture for data processing within an XML database. The aimed separation of concerns is essential not only to provide a clear workflow and flexibility. It also opens new doors for future improvements. Here we are going to discuss two of them.

6.1. Streamable Data Processing

In Chapter 4 the class `InputProcessor` was introduced, which takes a `DataSource` as input and parses it according to its content type. The result is always a list of `Resource` instances encapsulating the parsed metadata and content. By this approach the resulting `Resource` instance always holds the whole parsed document. In many cases, however, not the entire data is needed but just pieces of it. Thus, for example, a user may want to iterate over the records in a CSV file until they find such that meets a certain predicate. In this case it will suffice to parse just a single record per iteration but not the whole file at the beginning. This brings the requirement for stream-wise processing of data. The current model of the generic architecture for input and output does not allow this. It, however, can be modified in such a way to make it possible. The first component, which has to be taken into account, is the `Parser` abstract class. It has to be changed in such a way that it provides an iterative parsing of data. For example, there can be two methods – `hasNext()`, which checks if there is a next record and if yes – parses it, and `getNext()`, which returns the next record. The second component that has to be adapted to these changes is the `Resource` class. It can hold a reference to a corresponding parser instance and instead of the two methods for getting the whole parsed metadata and content, it can have two new ones returning iterators for stream-wise reading of metadata and content – `getMetadataIterator()` and `getContentIterator()`. With these modifications, the architecture will allow parsing of data on demand. It will not be needed to process a whole document when just parts of it are actually needed.

6.2. Relational Databases

The framework proposed in this thesis is designed to use exclusively files and streams as data sources. Another large source of information, which was not discussed until now, are the relational databases. The reason for this is that they have other characteristics and their content requires quite a different approach to be read and imported into an XML database. However, the reflections on streamable data processing in the previous section throw some light on the subject. The data in a relational database is organized in tables and when retrieved it is returned as a sequence of rows from one or many tables. This is what makes it incomparable to reading data from files. Nonetheless, if an XML database supports streamwise reading of data, parsing the records from a relational database to a specific XML representation can be done as explained above. For instance, a dedicated table parser may return a parsed entry each time its `getNext()` method is called. In that way despite the different nature of relational data, it will be treated in the same way as the data coming from other data sources.

7. Conclusion

Finding a unified way to organize the input and output in an XML database can be a challenging task. In this master thesis we proved that in spite of this, it is achievable. We started with three major use cases, which outline the requirements for generalized data processing. In Chapter 3 their coverage in several existing XML databases was investigated and two main conclusions were met. First, input and output are often inconsistent with one another – data can enter a database in one format through one channel but cannot leave it in the same format through the same channel. Second, in most cases a user has little or no control on data processing.

Based on these observations, in Chapter 4 we defined a generic framework that can be used as a model by the design of data input and output in an XML database. The benefits it brings are consistency, modularity, flexibility, easy enhancement. Adding support for new input and output formats has a clear workflow. The separate parsers and serializers can be used individually as well as through more general processors which are able to orchestrate them. The latter owe their "intelligence" to the presence of input and output configurations pointing the rules for data processing. These rules can be easily managed by the end users and applications.

In Chapter 5 we described how the proposed framework can be integrated in BaseX. Two new features were introduced – management of the input and output options based on configurations kept in an administrative database, and separated handling of content and metadata. The existing parsers and serializers were remodeled in order to be more consistent and reusable. Consequently, the existing input and output functionality has become more structured and easier to expand. As a final result, BaseX meets the requirements for the three use cases defined at the beginning of our work.

List of Figures

List of Tables

Listings

A. Appendix

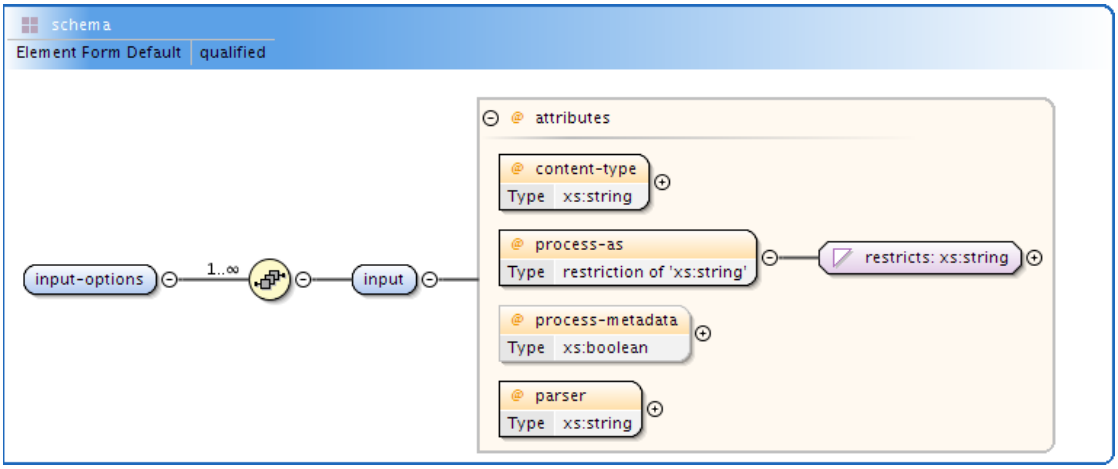


Figure A.1.: XML Schema: Input Options

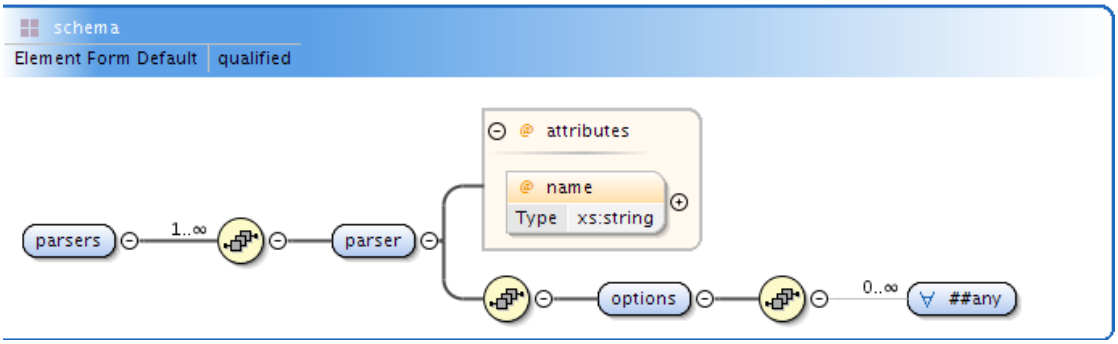


Figure A.2.: XML Schema: Parser Options

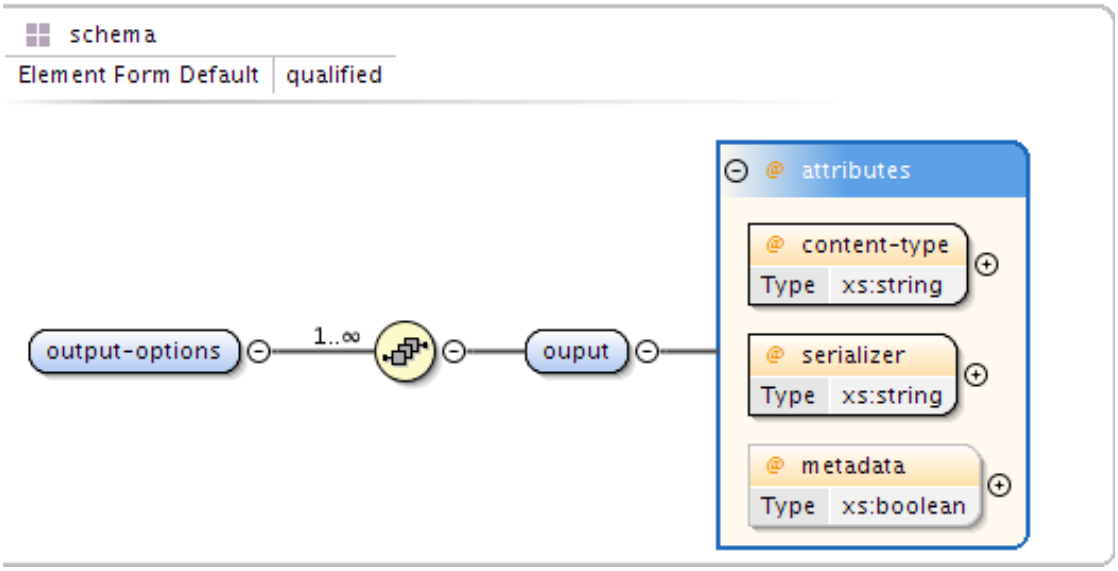


Figure A.3.: XML Schema: Output Options

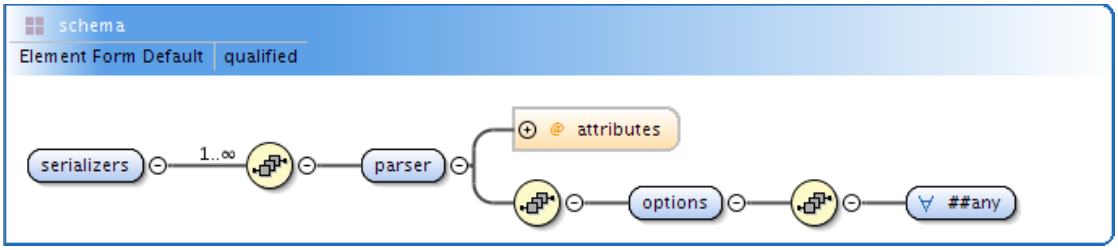


Figure A.4.: XML Schema: Serializer Options

Bibliography

[BAS] BaseX. http://basex.org/.

[BBB+09] Roger Bamford, Vinayak Borkar, Matthias Brantner, Peter M. Fischer, Daniela Florescu, David Graf, Donald Kossmann, Tim Kraska, Dan Muresan, Sorin Nasoi, and Markos Zacharioudakis. XQuery Reloaded. *Proc. VLDB Endow.*, 2(2):2, August 2009.

[BCF+] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language.

[Boua] Ronald Bourret. Going native: Use cases for native XML databases.

[Boub] Ronald Bourret. XML and Databases.

[EXIa] eXist. Content Extraction Module. http://exist-db.org/apps/wiki/blogs/eXist/ContentExtraction.

[EXIb] exist-db. http://exist-db.org/exist/index.xml.

[FIL] EXPath File Module. http://expath.org/spec/file.

[HTT] EXPath HTTP Module. http://expath.org/modules/http-client.

[MLA] *MarkLogic Application Developer's Guide*.

[MLC] *Mark Logic Content Processing Framework Guide*.

[MLI] *MarkLogic Information Studio Developer's Guide*.

[MOD] Modular Architecture. http://www.webopedia.com/TERM/M/modular_architecture.html.

[QIZa] Qizx. http://www.xmlmind.com/qizx/.

[QIZb] Qizx Product Description. http://www.xmlmind.com/qizx/product.html.

[SQL] EXPath SQL Module. http://expath.org/spec/sql.

[XDM] XQuery 1.0 and XPath 2.0 Data Model (XDM).

[XQA] Five Practical XQuery Applications. http://www.devx.com/xml/Article/15618.

[XQF] XQuery 1.0 and XPath 2.0 Functions and Operators. http://www.w3.org/TR/xpath-functions.

[XQS] XSLT 2.0 and XQuery 1.0 Serialization. http://www.w3.org/TR/xslt-xquery-serialization/.

[ZOR] Zorba xsl-fo module. http://www.zorba-xquery.com/html/modules/zorba/data-formatting/xsl-fo.