

Developing an Enterprise Web Application in XQuery

Martin Kaufmann, Donald Kossmann

ETH Zürich, Systems Group,
Universitätsstrasse 6, 8092 Zürich, Switzerland
{martinka, donaldk}@ethz.ch

Abstract. XQuery is a declarative programming language which can be used to express queries and transformations of XML data. It was designed by the W3C according to a series of use cases in the area of Web data management. The goal of this paper is to explore the expressive power of XQuery as a general-purpose programming language. To this end, this paper describes how to build an entire enterprise web application in XQuery. This paper first identifies the general requirements of a web application and then describes an appropriate implementation in XQuery. It is shown that it is actually possible and quite effective to implement a web application entirely in XQuery and that there are several advantages in doing so. The resulting code has proven to be very concise and elegant. More importantly, the use of XQuery simplifies the overall application architecture and improves flexibility.

Keywords: XQuery, Enterprise Application, Web Application, XML, Browser, Script, JavaScript, Programming, HTML

1 Introduction

Today, enterprise web applications play an important role for e-business software and data integration. Companies have demand for software which is well designed and structured to allow short training periods for new developers. The software should be easy to maintain by separation of concerns and extensible for future scaling.

Several technologies and frameworks have been proposed in recent years to meet these demands. One approach is to use scripting languages like PHP, Python, or Perl combined with a relational database. The advantage of this approach is that considerable parts of an application can be prototyped in a short time. On the negative side, such scripting languages are typically not appropriate in order to develop large-scale and complex applications. Furthermore, the resulting software exhibits the well-known impedance mismatch because the data models of the business logic and database do not match. In addition, the same tasks must be carried out several times at the different application tiers. For instance, errors in the SQL code must be handled in the application tier (using the scripting language) in order to provide meaningful error messages to the end user. Other examples include logging, checking of integrity constraints, and security.

In order to develop large-scale enterprise web applications, the state-of-the-art is to use frameworks such as the Java Enterprise Edition (J2EE) or Microsoft's .Net. These frameworks take advantage of the vast experience of object-oriented software development with programming languages like Java and C#. Using J2EE, for instance, Java objects can be mapped to a relational database using an object/relational persistence and query service like Hibernate. An application server (e.g., JBoss, WebLogic, or WebSphere) can report errors in HQL [5], an SQL like query language, during the deployment process. In the .Net framework LINQ [7] can be used in order to embed SQL-style database access into the object-oriented application code.

Obviously, both J2EE and .Net are better suited than scripting languages like PHP in order to develop large-scale applications. On the negative side, these frameworks require substantial training of developers. Even though technologies like Hibernate and LINQ try to remedy this problem, both J2EE and .Net suffer from the impedance mismatch between different data models used at different application tiers and from repeated work for the same tasks at different application tiers (e.g., error handling, etc.). The core of this problem stems from the fact that all these frameworks actually make use of a mix of technologies. This paper argues that a unified technology stack can result in simpler, more flexible, and more efficient application code.

The impedance mismatch between relational databases and object-oriented programming has been identified already more than twenty years ago [3]. A result of this observation was the development of object-oriented database systems. These systems were popular in the Nineties and have received revived interest recently. A recent example is the DB4o system for Java [4]. The use of DB4o mitigates the impedance mismatch problem by providing a uniform (object-oriented) data model. DB4o offers the SODA query API, which makes it possible to write queries in pure Java code on the notion of a query graph. The following Java code snippet shows how to retrieve all authors which have either the name "Don Chamberlin" or whose age is between 20 and 30 using the SODA API of DB4o.

Code 1: Query example in Java

```
Query query=db.query();
query.constrain(Author.class);
Query ageQuery=query.descend("age");
query.descend("name").constrain("Don Chamberlin")
.or(ageQuery.constrain(new Integer(20)).greater()
.and(pointQuery.constrain(new Integer(30)).smaller()));
ObjectSet<Author> result=query.execute();

for(author: result) {
    System.out.println("<b>" + author.name + ": " + author.age
+ "</b>");
}
```

This paper explores a new approach to develop enterprise web applications. The key idea is to leverage the (unified) technology stack developed by the World Wide Web Consortium (W3C). Over the years, the W3C has standardized in a bottom-up way all building blocks which are needed to develop enterprise web applications. The W3C has developed data formats (i.e., XML, HTML, and XHTML), a schema language (i.e., XML schema), protocols for distributed and service-oriented computing (i.e., HTTP, REST, and the Web Service family of recommendations), and programming languages to write application code (i.e., XPath, XSLT, and XQuery). Even though all these technologies were designed individually having a series of (small) use cases in mind, all these technologies fit together well and can be integrated into a single and uniform technology stack. The main contribution of this paper is to demonstrate that such a uniform technology stack based entirely on W3C standards can be used in order to build and deploy large-scale enterprise web applications.

A central piece of this new approach to build web applications using W3C standards only is the use of the XQuery programming language [18]. XQuery was initially designed to query and transform XML data. XQuery is an extension of XPath [17] which was developed in the late Nineties for XML processing. Furthermore, XQuery is related to XSLT which was specifically designed for XML transformations. A few years ago, the W3C has started to develop XPath, XSLT, and XQuery in concert in order to provide a uniform and powerful technology for any kind of processing of data on the web (including, but not necessarily restricted to XML and (X)HTML). In addition to constructs for powerful query and transformation, this family of W3C recommendations has been extended by languages for updates (the XQuery Update Facility [14]), scripting (the XQuery Scripting Extension [18]), and full-text search (XQuery Full-text [17]). Furthermore, XQuery has been extended with built-in support for RESTful services and Web Services and processing of data streams (e.g., RSS aggregation) [18]. In the remainder of this paper, we refer to this whole family of languages and extensions simply as XQuery because they all fit together and are implemented together so that the developer can use all features in a uniform way, without being aware that these are different languages and recommendations.

Using XQuery, the simple example from above that retrieves all authors whose name is “Don Chamberlin” or whose age is between 20 and 30 can be expressed as shown in Code 2. Rather than storing the authors in a relational database, the authors are stored in an XML collection which may or may not be stored in a relational database (the specifics of the database system are encapsulated, thereby providing an additional level of data independence):

Code 2: Query example in XQuery

```
for $author in $coll/author[name = 'Don Chamberlin'
  or (age > 20 and age < 30)]
return
  <b>{$author.name}<b>: { $author.age}</b>
```

Both code snippets (Code 1 and 2) perform the same functionality and return a list of authors and their age as a result. Obviously the XQuery implementation is much more concise and easier to read. The main reason for the Java code being so verbose is that Java simply was not designed for that purpose. Similar observations can be made for the other alternatives to develop web applications (e.g., PHP, .Net, etc.). XQuery, on the other hand, was specifically designed for these tasks. In addition, XQuery was extended to become a general-purpose programming language for web applications. Again, the goal of this paper is to explore whether XQuery lives up to this expectation.

The remainder of this paper is organized as follows: Section 2 defines the requirements of an enterprise web application and introduces the PubZone application, the running example used throughout the remainder of this paper. Section 3 describes a possible system and software architecture. Section 4 sketches the implementation of the PubZone application in XQuery. Section 5 compares the XQuery and Java implementations of PubZone and discusses our experience in the development of PubZone with XQuery. Section 6 discusses related work. Section 7 contains conclusions and avenues for future work.

2. Requirements

An enterprise application is software which performs business tasks like management of a customer database, control of production or banking applications. Often a large number of users access the application at the same time using the company network or the public internet. The application may be hosted on a central server, but can also be distributed over a network.

This section describes common requirements of enterprise web applications and introduces the running example used throughout the remainder of this paper; i.e., the PubZone application. We hint why XQuery (or more precisely, the use of W3C standards) is a good match to implement these requirements; a more complete discussion is given in Section 5.

2.1 General Requirements of Web Applications

/R1/ Persistence and database. An enterprise web application needs a way to store data persistently. The database must support many users which access the application concurrently and comply with the well-known ACID paradigm. Data must be accessible from the database using a declarative query language that supports operators such as joins, aggregation, and sorting natively.

/R2/ Programming Model for Millions of Lines of Code. In order to cope with the complexity of a large application, it is necessary to decompose compound functionality into distinct functions. Each of these functions performs a definable task which ideally can be reused.

/R3/ Error Handling. If an error occurs in a subordinate function (callee), it must be possible for the superordinate function (caller) to catch the error which has been caused by the callee. The caller must be able to handle the error correctly in order to ensure an appropriate behavior of the application.

/R4/ Session Management and Security. It must be possible to correlate several user requests. For instance, a user may first login to the application and then execute several functions of the application without the need to provide the login credentials with every access. Correlating user requests is typically carried out by the means of a Session ID which is allocated as part of the first (i.e., login) request. Session IDs are communicated from the user's browser to the application server either by cookies or URL rewriting. For security reasons, it is important that unauthorized users are not able to forge cookies or URLs with Session IDs.

/R5/ GUI. The GUI is represented as an (X)HTML page (possibly with embedded JavaScript) which is generated by the web application. Each page contains a hierarchical menu which is generated dynamically. Functions in the menu are displayed according to the privileges of the user. The menu is updated in an event-based way whenever a user selects a different function from the menu.

/R6/ Modules to Facilitate Recurring Tasks. In a web application, there are several recurring tasks like building the headers and footers of (X)HTML pages in a specific layout, the position of the menu etc. The input of the user is done by means of HTML forms which have to be pre-populated whenever existing content is edited. Another example for a recurring task that can be found across applications is user management (e.g., the credentials of users such as user names and passwords) and role-based authorization (e.g., access privileges of groups of users). A framework to develop web applications must provide strong support for such recurring tasks.

/R7/ REST and Web Services. Often it is required for a web application to communicate with other network services. Therefore it has to provide an open interface to the outside world. More specifically, standards such as REST [11] and Web Services must be supported. In many ways, modern web applications can be seen as a set of services which are exposed to the outside world and which are composed of other services (internally and externally). Support for exposing application code as a service and integrating other services via REST and Web Services protocols must be natively supported by a programming framework for web applications.

2.2 PubZone Application

As a running example, this paper uses the PubZone application. PubZone is a web based repository for scientific publications. It is going to be used for the first time for publications of the ACM SIGMOD 2009 conference, and it is likely to be adopted by several venues and journals of the life science research community. In a nutshell,

PubZone provides a Wiki (or blog) for every publication. That is, it provides a discussion forum for each publication and a way to upload supplemental information such as experimental data and results, source code of implementations of algorithms, and PowerPoint presentations or reports from, e.g., students who have studied the publication as part of seminars. Furthermore, PubZone allows the publication of (anonymized and unanonymized) reviews, author feedback, reviewer discussions, and ratings of papers and reviews and supplemental material; all these features have been inspired by Amazon's open reviewing platform in which any user can write a review about any work. PubZone also has a powerful search interface (by author and publication) and ways to compute statistics (e.g., most controversial papers, publication venue that has attracted the most discussions, authors whose reviews have the highest ratings). The design of PubZone was discussed over several months within the ACM SIGMOD Executive Board and in the EMBO Board (European Molecular Biology Organization).

Obviously, the PubZone application involves all the requirements listed in Section 2.1. Furthermore, the PubZone application needs to be very flexible and customizable. For instance, after discussions within ACM and EMBO, it became clear that the biologists favor a much more open reviewing scheme than the computer scientists.

The source code of the PubZone application is available for download. There are two versions of this source code: XQuery [10] (based on W3C standards only) and Java [9] (using the J2EE framework). The XQuery version is described in Section 4. A comparison of the two versions is given in Section 5. For SIGMOD 2009, the Java version of PubZone will be used because the J2EE application server (i.e., JBoss) is more stable than our XQuery application server.

3. Architecture

3.1 System Architecture

For this work, we adopt the traditional three-tier architecture for enterprise web applications. As shown in Figure 1(a), this three-tier architecture is typically implemented using different languages and data models at each tier. As shown in Figure 1(b), we propose to use, as much as possible, one programming language and one data model throughout the entire application stack. The advantage of this approach is that the architecture becomes more flexible, more efficient, and simpler.

The architecture of Figure 1(b) is more flexible because it allows for moving code from one tier to another. This migration is possible because the same data model, data format, and programming language is used at all tiers. There are many reasons why it might be necessary to move code to a different tier; in the past, varying trends with regard to cost (e.g., offloading servers), security and system administration have led to different instantiations of "fat client" and "thin client" architectures. Currently, there seems to be a trend towards "fat clients" for Web 2.0 applications using AJAX

technologies and “thin clients” for mission critical applications, but these trends can easily swing back in the future. Actually, Figure 1(b) does not show the use of the XQuery programming language at the top (presentation) tier because JavaScript is the predominant programming language to program the browser today; however, all modern web browsers already support XPath (a subset of XQuery) in order to more elegantly navigate the internal XML representation of a web page (exposed in JavaScript using the DOM interface). Furthermore, a full-fledged XQuery plug-in for Microsoft’s Internet Explorer has already been released and an XQuery plug-in for Firefox is under development [15].

Simplicity is achieved by applying the same framework (and programming language) to all three application tiers. As a result, tasks such as logging, error handling, and checking of integrity constraints must be implemented only once. In fact, the whole application can be implemented in a uniform way, thereby appearing as a single-tiered application. Obviously, this simplicity increases maintainability and reduces the size of the code; an example of this effect is given in the introduction. We will quantify this effect in Section 5, which compares the lines of code of the XQuery and Java implementation of the PubZone application.

Efficiency can be improved in three ways with the help of the architecture of Figure 1(b), as compared to the architecture of Figure 1(a). First, effort for data marshalling is reduced because the same data model and format is used in all tiers. Second, repeated work for logging and checking of integrity constraints is avoided by the simplified “single-tier” view of application development, as explained in the previous paragraph. Third, the whole application can be optimized globally, rather than optimizing, say, each database query individually. The efficiency advantages of the architecture of Figure 1(b) cannot be exploited today because the XQuery application servers needed for this architecture are not mature enough products yet. In fact, one of the goals of this paper is to encourage vendors of application servers (e.g., Oracle and IBM) to improve their product offerings in this regard.

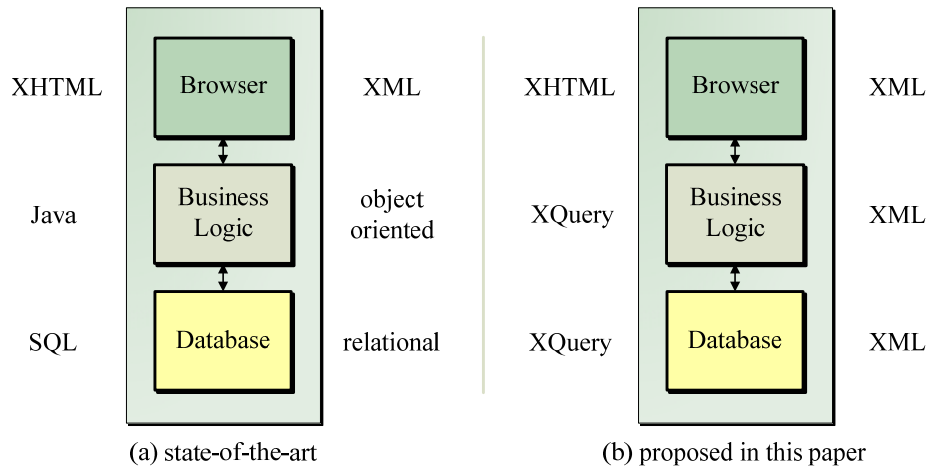


Fig. 1. Three-tier Application Architecture

For the implementation and deployment of the PubZone application, we used the Sausalito XQuery application server [12]. Sausalito is a product of 28msec Inc., a start-up based in Palo Alto and Zurich. Sausalito integrates an XML database system, the Zorba XQuery engine [19], and the Apache Web Server into a single XQuery application server. Deployment is done on top of Amazon Web Services (in particular, S3 and EC2). Figure 2 depicts the way Sausalito handles requests from Internet users. The Sausalito product is currently in an early alpha release and tested with a number of reference applications such as PubZone. An alternative would be to use the Mark Logic server [8]. Mark Logic also provides a complete framework to develop enterprise web applications using XQuery. In all, we expect that more products will appear on the market place and that large vendors will also support the architecture of Figure 1(b). Oracle, IBM, Microsoft already support XQuery as part of their database products and in various places as part of their middleware products.

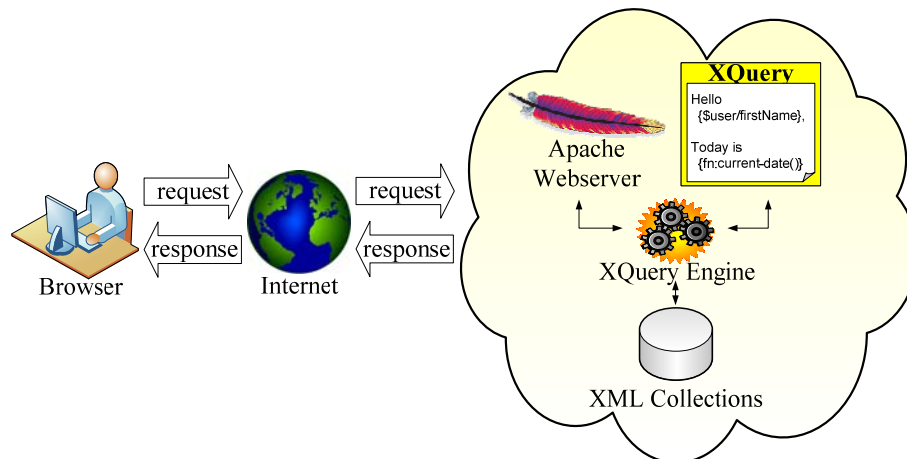


Fig. 2. XQuery engine integrated into a web server

3.2 Software Architecture

According to best practices, we adopt the MVC pattern in order to separate business logic from program control flow and presentation layer. In our implementation of the PubZone application, the *Model* contains the business logic of the application. Two instances of the MVC pattern are used to separate generic components from application-specific components, as shown in Figure 3. Generic components are used by many applications. Examples for generic components are user management or for the generation of (X)HTML in different layouts. Application-specific components call generic components; a typical example is the PubZone user management which is based on the generic user management and which has specific implementations for authors and administrators.

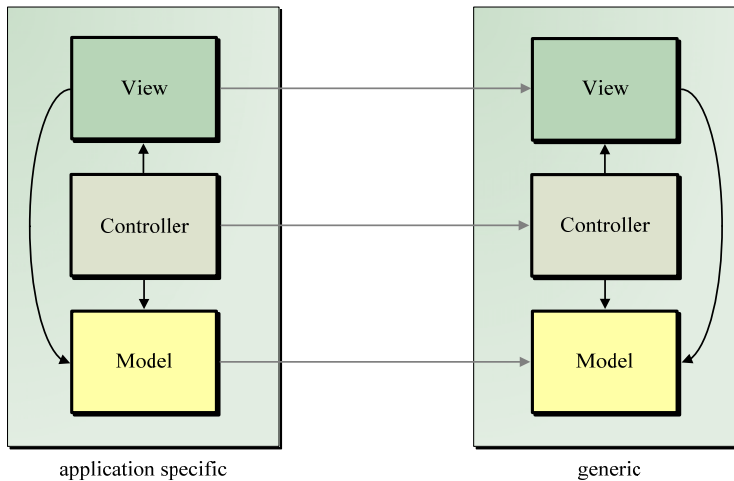


Fig. 3. Two instances of the MVC pattern

4. Implementation

Based on the software architecture given in Section 3.2, this section sketches an implementation of the PubZone application in XQuery. An overview is given and example code snippets are presented in order to get a picture of the entire implementation. A basic familiarity with XQuery is assumed in order to understand the code snippets. The interested reader is referred to [18] for a tutorial and detailed description of XQuery.

4.1 Overview

The structure of the PubZone application can be visualized by the means of a UML class diagram. Figure 4 shows a fraction of the PubZone class diagram; this fraction represents the classes involved in the PubZone user management. Each UML class is

implemented by an XQuery module. In XQuery, a module consists of a collection of functions which can be called directly by another XQuery program or can be exposed via REST or as a Web Service. The implementation of the whole PubZone application involves 64 XQuery modules of which only a fraction is depicted in Figure 5. Unlike object-oriented programming languages, XQuery does not bundle behavior and state. As a result, XQuery modules cannot be instantiated (there are no objects in XQuery) and there is no inheritance. While this restriction sounds prohibitive, it turns out that XQuery is nevertheless a good match to implement enterprise web applications like PubZone.

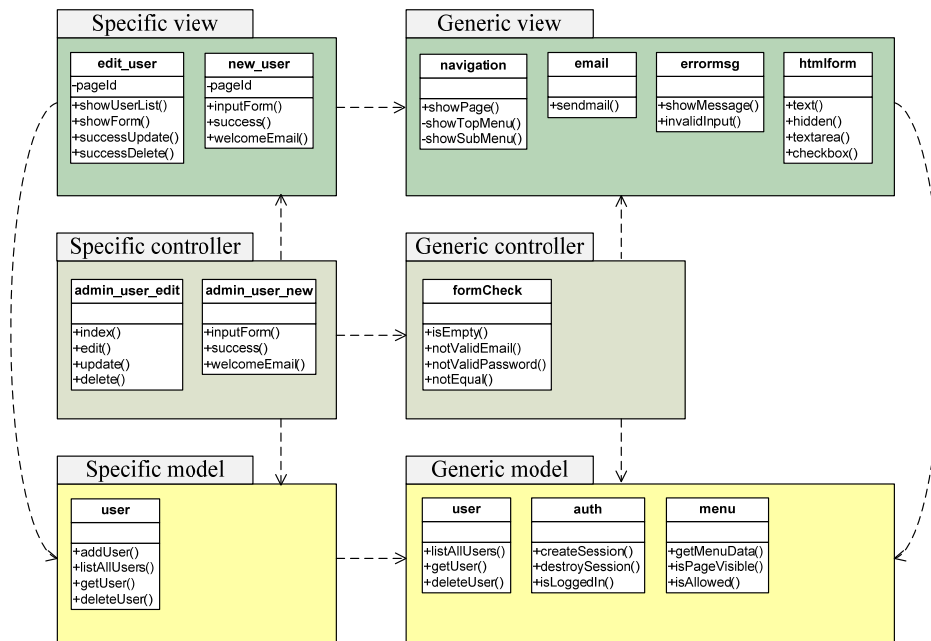


Fig. 4. UML diagram of the PubZone User Management

Using XQuery, state is kept in *collections*. A collection contains a sequence (i.e., list) of items (e.g., XML elements). Each XML element which is a member of a collection can have sub-elements and attributes, just like any other XML element according to the XML standard. Elements and their sub-elements and attributes can be typed. Furthermore, a collection is uniquely identified by a URI. Conceptually, a collection is equivalent to a table in a relational database; elements correspond to tuples and sub-elements correspond to fields of tuples. The set of all collections corresponds to the whole database.

The implementation of the PubZone application has five collections in order to store all registered users, groups of users, sessions, menus, and publications, as shown in Figure 5. The Java implementation of PubZone which is based on a relational database involves 17 tables; using XQuery and an XML database, less collections are needed because N:1 relationships can be modeled more naturally. For instance, the

“author” information of a publication can be nested inside each publication. In Figure 4, such multi-sets are marked with a “+” sign, indicating that one or more sub-elements of this kind are possible. For simplicity, the complete type information is omitted in Figure 4, but it should be straight-forward to deduce.

Table 1. XML Collections of the PubZone Application.

Collection name	Attributes
pubzoneUserDB	username, firstName, lastName, email, password, groupId
pubzoneGroupDB	groupName
pubzoneSessionDB	userName, sessionId, time
pubzoneMenuDB	title, pageId, groupId+
pubzonePublications	title, keyword+, description, author+

In order to access collections, the *fn:collection()* function of XQuery [18] is used. The next subsection gives an example code snippet for the use of this function. This function is called like any other function as part of an XQuery program and it can be fully composed with any other expression of the program. That is, database access is seamlessly integrated into the programming language with XQuery. Logically, the *fn:collection()* function returns the whole collection which can be huge (just like a relational table). In practice, however, the XQuery optimizer optimizes a whole program and, e.g., pushes down predicates, carries out access path selection (i.e., indexes) and join ordering in the same way as a traditional database optimizer does. As stated in Section 3.1, optimization is one reason why an application written entirely in XQuery is potentially faster than an application written, say, in Java with SQL or XQuery embedded into the Java code: If the whole application is written in XQuery, then the whole application code (or at least whole modules) can be optimized, thereby having knowledge of the complete access pattern of the application to the database and possibly exploiting multi-query optimization and common sub-expressions across queries. In the traditional approach, each query is optimized individually, thereby missing important opportunities for (multi-query) optimization.

4.2 Example Code Snippets

Code Listings 3, 4, and 5 contain the PubZone application-specific functions of the *View*, *Controller*, and *Model* components of adding a new user. (For presentation purposes, the definitions are slightly changed from the original code.) Code 3 shows the `inputForm()` function of the *View* component which generates the HTML form that is used in order to register new users. Implementing this function in XQuery is straight-forward because XQuery was designed to generate XML (including XHTML) output. In addition to simple XHTML construction of form fields, this example also shows how some more sophisticated logic can be implemented in XQuery: Here, the list of all possible user groups is generated (as a pull-down menu in the GUI) by calling the `listAllGroups()` function from the *Model* component (i.e., database) and processing the result of that function.

Again, the `inputForm()` function of Code 3 only specifies application-specific behavior: Generic functionality such as embedding a form into a page layout and generating headers and footers is implemented in the `showPage()` function of the generic *View* implementation which is called in the last line of the `inputForm()` function. The `showPage()` function takes as a parameter a `$page_id` (in this example, the id for the `new_user` form) in order to adjust the menu selection.

Code 3: `inputForm()` Function of the `new_user` Module of the *View*

```
declare function def:inputForm () {
  let $title := "Add a new user"
  let $text :=
    <form method="post" action="/user_new/submit">
      <p>Username: { htmlform:text("uid", 25) }</p>
      <p>E-Mail: { htmlform:text("email", 25) }</p>
      <p>Group:
        <select name="groupId" size="1">
          { htmlform:option("", "Please select...") }
          { for $group in groups:listAllGroups()
            return htmlform:option($group/@id,$group/name) }
        </select></p>
      <input type="submit" value="Save" />
    </form>
  return navigation:showPage($title, $text, $def:PAGE_ID)
};
```

Code 4 shows the application logic to process a new user in the *Controller*. As a parameter, the *Controller* receives an XML input generated from the HTTP request of the web client. Sausalito has pre-defined XQuery modules in order to process these HTTP requests (not shown here). The *Controller* carries out several checks (i.e., duplicate users) and, if successful, the new user is recorded in the database. Again, it should become clear that XQuery is a nice fit to implement this kind of functionality.

Code 4: `addUser()` Function of the `new_user` Module of the *Controller*

```
declare sequential function def:addUser($new_u as element)
                                     as element() {
  if(fn:isEmpty($new_u/uid)) then
    exit with errormsg:invalidInput("No username!")
  else if(modelUser:exists($new_u/uid)) then
    exit with errormsg:invalidInput("Username is taken!")
  else
    return modelUser:addUser($new_u);
    exit with viewNewUser:success();
};
```

In order to record the new user into the database and check whether a user with the requested name already exists, the *Controller* calls the *Model* component; the `modelUser` module in this particular case. The `addUser()` function of this module is

shown in Code 5. Again, XQuery is a perfect match to implement functionality at this level because database collections can naturally be accessed and manipulated with XQuery. As stated in Section 4.1, the `fn:collection()` function is used for this purpose and the full power of XQuery (and the XQuery Update language) is applicable to directly manipulate items in the database. In this particular example, the `pubzoneUserDB` collection (Table 1) is modified. The URI of this collection is declared as a constant referred to by the `COLL_USER` function in the `const` XQuery module.

Code 5: `addUser()` Function of the `modelUser` Module of the *Model*

```
(: Insert a new user :)
declare function def:addUser($new_u as element) {
  insert node $new_u into fn:collection(const:COLL_USER())/a
};
```

5. Discussion

This section discusses whether our implementation of the PubZone application in XQuery meets the requirements of web based applications listed in Section 2. Furthermore, this section contains a comparison between the XQuery and Java implementations of PubZone.

5.1 Are the requirements met?

The requirements from Section 2.1 can be classified into three groups. The first group of requirements are fulfilled natively (or automatically) by XQuery and the XQuery application server so that the application programmer does not need to bother about them. The second group of requirements are not fulfilled automatically so that they must be implemented by the application programmer for each application individually. Like the second group, the third group of requirements are not fulfilled automatically, but these requirements could be fulfilled by providing an extended XQuery programming framework (i.e., generic XQuery modules for the MVC pattern) so that they would be implemented only once for a large range of applications.

5.1.1 Requirements Met by XQuery or an XQuery Application Server.

[R1] As stated in Section 4, XQuery includes the concept of collections to store data. An XQuery application server like Sausalito or Mark Logic stores collections persistently so that the persistence requirement is fulfilled natively.

[R3] Error handling mechanisms are provided by XQuery by means of `try` and `catch` expressions whose semantics are similar to the Java analogon. Thus, a caller can catch an error caused by a callee function. The construct can also be used to

handle errors caused by functions that have been called. This enables a callee to move the error handling to the caller which simplifies the code. In all, this requirement is met by XQuery as well as by object-oriented programming environments like Java or C#/.Net.

[R6] The XQuery module concept was designed to create libraries of functions. So, the concept of implementing recurring tasks only once is naturally supported with XQuery. In the implementation of the PubZone application, this feature was beneficial and ensured that a great deal of code is part of generic components which can be re-used for other applications.

[R7] Any XQuery module can be exposed as a RESTful service or Web Service. As a result, this requirement is automatically met.

5.1.2 Requirements Met by the Application-specific Implementation.

[R2] The XQuery implementation of the PubZone application involves 64 modules and about 3500 lines of XQuery code. While this is a moderately small project, the experience shows that the modularization works quite well with XQuery. We expect that the module concept of XQuery would scale to large applications, too.

A point of criticism is that XQuery does not support *private* and *protected* functions. All functions defined in a module are visible to everybody. Furthermore, XQuery does not support the notion of a *package* which is often handy in object-oriented programming in order to bundle a set of class definitions.

[R5] The GUI is implemented by means of an HTML output. We have provided a function that renders the page and automatically inserts the menu for navigating between the different pages. Using XQuery to provide (X)HTML output is clearly more elegant than using, say, servlets because the same programming style is used. Separating the *View* from the *Controller* and *Model* as mandated in the MVC pattern is possible in an XQuery implementation just as well as with Java or .Net or any other programming style.

5.1.3 Requirements met by an Extended XQuery Framework

[R4] The Sausalito XQuery application server has built-in support for session management. The PubZone implementation does not use this built-in support for session management because that feature was added only recently and because that feature is not standardized (i.e., the support varies between different XQuery application servers). As a result, we provided our own generic implementation of session management; this implementation could easily be integrated into the XQuery application server, once a standard API for session management is found.

In summary, all requirements could be met, but there is room for improvement as XQuery development frameworks evolve and more XQuery libraries become available.

5.2 Experience

Coding style: Java vs. XQuery. We have been developing enterprise Java applications for many years. For an XQuery newcomer, it is unfamiliar to write business logic in XQuery. A new way of thinking is required for an XQuery application programmer because XQuery is a functional programming language. Nevertheless, the XQuery code of the PubZone application [10] ended up to be elegant und much shorter than the corresponding Java implementation [9].

Table 3. Lines of Code (LOC) of PubZone Implementation in Java and XQuery.

	J2EE / LOC	XQuery / LOC
Model	3100	generic: 210 specific: 240 total: 450
View	4100	generic: 330 specific: 1500 total: 1830
Controller	900	generic: 30 specific: 1180 total: 1210
Total code length	8100	3490

Table 3 shows the lines of code necessary to implement PubZone in Java and in XQuery. For the Java implementation, we did not differentiate between generic and application specific code. As expected, the XQuery implementation was more compact; this is particularly (and not surprisingly) true for the *Model* and the *View* components. In both of these components, the Java implementation suffers from the technology discontinuity and impedance mismatch between Java and relational databases (on the one side) and Java and XHTML presentation data (on the other side). In the *Controller*, Java was actually more concise but only marginally. The *Controller* is shorter using J2EE because of the use of the Struts Frameworks which allows the *Controller* to be configured by means of an XML configuration file.

Features and Performance of the Sausalito Application Server. Currently, the Sausalito application server is in an early alpha release of its first version. Clearly, it is not yet mature. (Mark Logic is better in this regard.) At the time the PubZone application was developed, Sausalito did not provide several important features such as session management, sending of emails, generation of random strings. (Actually, at the time of writing this paper, these features have been added, but the overall statement of lack of maturity is still relevant.) Furthermore, we experienced some performance problems at the beginning which have been fixed in course of this project. In all, time will prove whether XQuery application servers mature; we tend to be optimistic as XQuery gains more and more attention.

6. Related Work

As mentioned in the introduction, there are several alternative ways to implement enterprise web applications today. Most of these involve a mix of technologies; at the very minimum a query language for database access (e.g., SQL or XQuery), a programming language for the application logic (e.g., Java, .Net, PHP), and a framework to provide REST services (e.g., Java Axes) and serve web pages (e.g. servlets or ASP.Net). The goal of this work was to give an alternative based entirely on W3C standards and use XQuery as the one and only programming language to implement database access, application logic, and presentation logic.

Obviously, there has been a great deal of work on XQuery already. As mentioned before, XQuery has been implemented by all major database vendors. Furthermore, XQuery is a popular tool for various tasks in the middleware; for instance, BEA's Enterprise Information Bus is based on XQuery and BEA's ALDSP server for enterprise information integration is based on XQuery [1]. XQuery is also used for data stream processing such as RSS streams [2], [6]. There are also a number of development tools emerging for XQuery development emerging on the market place including Eclipse Plug-ins and debuggers (e.g., StylusStudio, XML Spy, Oxygen, and XQDT [16]).

7. Conclusion

This paper reported on our experience to develop a complex and customizable enterprise web application entirely using the XQuery programming language and using related W3C recommendations (e.g., XML as a data format, XML Schema to describe data types, and HTTP and REST for remote communication). Overall, the conclusion is that the W3C family of standards is very well suited for this task and has important advantages over the state-of-the-art (e.g., J2EE, .Net, or PHP). Most importantly, using XQuery and W3C standards only ensures a uniform technology stack and avoids the technology jungle of mixing different technologies and data models. As a result, the application architecture becomes more flexible, simpler, and potentially more efficient. Today, the biggest concern in adopting this approach is that there are no mature application servers available, but we believe that the situation will change soon in this regard.

Obviously, this work was only a first step in order to develop complex and large-scale enterprise web applications entirely in XQuery with the help of W3C standards. In the future, more experience with other applications is needed. One important avenue for future work is to develop a methodology to develop such applications with XQuery.

References

1. BEA AquaLogic Data Services Platform, <http://edocs.bea.com/aldsp/docs30/>
2. I.Botan, P.Fischer, D.Florescu, et al.: Extending XQuery with Window Functions. VLDB 2007, Vienna, Austria.
3. G.Copeland, D.Maier. Making Smalltalk a database system. SIGMOD 1984, Boston, USA.
4. Native Java & .NET Open Source Object Database, <http://www.db4o.com/>
5. HQL: The Hibernate Query Language, http://www.hibernate.org/hib_docs/reference/en/html/queryhql.html
6. C.Koch, S.Scherzinger, N.Schweikardt, B.Stegmaier. FluXQuery: An Optimizing XQuery Processor for Streaming XML Data. VLDB 2004.
7. LINQ Project, <http://msdn.microsoft.com/en-us/netframework/aa904594.aspx>
8. MarkLogic Server, <http://www.marklogic.com/>
9. M.Kaufmann et al.: PubZone Java implementation, <http://java.pubzone.org/>
- 10.M.Kaufmann: PubZone XQuery implementation, <http://xquery.pubzone.org/>
- 11.Leonard Richardson, Sam Ruby. RESTful Web Services. O'Reilly 2007.
- 12.Sausalito, XQuery Application Server, <http://sausalito.28msec.com/>
- 13.D.Chamberlin et al.: W3C XML Query. W3C Recommendation, 2007.
- 14.D.Chamberlin et al.: XQuery Update Facility 1.0, W3C Candidate Recommendation, 2008.
- 15.G.Fourny, D.Kossmann, T.Kraska, M.Pilman, D.Florescu: XQuery in the browser. SIGMOD 2008, Vancouver, Canada.
- 16.G. Petrovay: XQuery Development Tools, <http://www.xqdt.org/>
- 17.Sihem Amer-Yahia et al.: XQuery and XPath Full Text 1.0, W3C Candidate Recommendation, <http://www.w3.org/TR/xpath-full-text-10/>
- 18.D.Chamberlin et al.: XQuery 1.1, W3C Working Draft, <http://www.w3.org/TR/xquery-11/>
- 19.The Zorba XQuery Processor, <http://www.zorba-xquery.com/>