FEDERAL UNIVERSITY OF RIO GRANDE DO SUL

INFORMATICS INSTITUTE

DEPARTMENT OF COMPUTER SCIENCE

HENRIQUE VALER

# XQUERY-BASED APPLICATION DEVELOPMENT

Graduation Thesis

Dipl.-Inf. Sebastian Bächle
Advisor

Prof. Dr. Renata Galante
Co-advisor

Porto Alegre, March 2010.

# Acknowledgements

First I would like to thank my family: my parents Loreci Carlos Valer and Dirlei Teresinha Lamonatto Valer, for the respect and care with which they always treated me and for the freedom of choices which was always given to me. An special acknowledgement to my dad, which role model I always try to follow throughout my life.

Next I would like to deeply thank Prof. Dr. Theo Härder, for the opportunity to work in his research group and realize most of this thesis efforts under Technical University of Kaiserslautern. I also would like to thank all participants of the DBIS group, especially Dipl.-Inf. Sebastian Bächle, which is co-advisor of this thesis, and was unbelievably helpful along this writing process.

I also would like to thanks Profa. Dra. Renata de Matos Galante for advising me in the rest of this work. Thanks also must be given to UFRGS as a whole, for the opportunity of studying in a university with excellence and international prestige, and completely out of costs. The lessons I learned in this place opened doors in my life and contributed immensely to my growth, not only professionally but also personally.

Finally, to all others who I may have forgot to mention, who have directly or indirectly helped me through this work.

# Table of Contents

# Glossary

| | |
|---|---|
| XML | Extensible Markup Laanguage |
| XTC | XML Transaction Coordinator |
| ASP | Active Server Pages |
| PHP | Personal Home Page |
| JSP | JavaServer Pages |
| LAN | Local Area Network |
| HTTP | Hypertext Transfer Protocol |
| J2EE | Java 2 Platform, Enterprise Edition |
| UI | User Interface |
| JDBC | Java Database Connectivity |
| W3C | World Wide Web Consortium |
| CRUD | Create, Read, Update and Delete |
| ACID | Atomicity, Consistency, Isolation and Durability |
| DOM | Document Object Model |
| SAX | Simple API for XML |
| FTP | File Transfer Protocol |
| ARPA | Address and Routing Parameter Area |
| HTML | HyperText Markup Language |
| URL | Uniform Resource Locator |
| MIME | Multipurpose Internet Mail Extensions |
| RFC | Request for Comments |
| MD5 | Message-Digest Algorithm 5 |
| CSS | Cascading Style Sheets |
| RAD | Rapid Application Development |
| UML | unified Modeling Language |
| REST | Representational State Transfer |
| AJAX | Asynchronous JavaScript and XML |
| AWS | Amazon Web Services |
| XQDT | XQuery Development Toolkit |

## List of Figures

## List of Tables

# Abstract

This thesis evaluates XQuery as a complete solution for data storage and processing, application logic and UI. To this end, I designed and implemented a simplified e-Commerce Web Application entirely with XQuery. I used XTC [8], a native XML database management system, as application server. This thesis also describes necessary updates and features related to the server, extending XTC to work as application server. The extension comprises an HTTP infrastructure and XQuery functions for the application development, not supported by the XQuery specification. The resulting application has shown that the use of XQuey and XML avoids the impedance mismatch between the data and the application logic and that it simplifies application development.

## keywords

XQuery, XML, web application, web development.

## 1 Introduction

Nowadays, more and more applications are built using web technologies. On the client side, we have HTML enhanced with Adobe Flash, Ajax and Javascript. On the server-side, PHP, ASP and JSP. These applications are accessed via web browsers and are usually referred to as web applications.

Their popularity comes from a series of advantages. Usually, a user can access an application through any web browser connected to the internet or LAN. Moreover, the application and its data are stored on the server-side, which increases reliability and makes it accessible from almost everywhere [17].

Besides that, web applications introduce advantages also for the developers. Applications can be developed, deployed and maintained without any interaction of a technically skilled user. In fact, users are not required to install or update the applications by themselves. Development and maintenance can be done remotely and the developed applications are platform independent. Any modern browser is able to display these applications.

The main cost for these features is complexity. While applications grow in size and features, development complexity grows even more. Some of the challenges that development of web applications need to address nowadays include:

**Availability**
Web applications need to run 24/7 and need to have fast response times. For that, the robustness of the code is essential.

**Scalability**
Because these platforms are accessible from almost anywhere, it is not surprising that the amount of users tend to grow with time. For that, large amounts of simultaneous connections need to be managed.

**Multi-user**
Web applications are not static. They work with dynamic data and changes come and go from user to servers quite regularly.

**Persistent data**
Most applications rely on persistent data storage, a characteristic of dynamic applications. Almost all kinds of applications store or load persistent data. A blog application, for example, must store comments and new postings somewhere.

**Accessibility**
The access to these applications is basically through a web browser. The communication between the client and the server occurs with HTTP, so the application server must support HTTP communication.

These challenges are typically addressed by the use of complex infrastructures and technologies. Of course, the system should obey common properties of any software, such as readability, maintainability, low complexity, composibility and reusability. Therefore, developers usually build their application on top of frameworks like Java Enterprise Edition (JEE), for example. JEE uses object orientation and a multi-layered approach to develop applications [4].

A web application can be typically divided into three logical layers: presentation layer, business layer and resource layer. The presentation layer provides the user interface (UI) and communicates with the business layer in order to process the application logic. The business layer processes these requests and exchanges data with the resource layer. The resource layer is responsible for the permanent storage of the data [19]. This approach often deals with a special problem: the impedance mismatch between the layers, commonly appearing between relational databases and object-oriented languages[20][21]. While the presentation layer usually uses HTML, the business layer uses Java language and the resource layer uses any kind of relational database to storage data.

We can better illustrate this problem with an example. Suppose we have to store a list of users in a database, consisting of an identifier, a name and the user's age. We want a list of all users with the name containing "Schwarzenegger" that are older than 50 years. In a Java application, we would use JDBC (a set of Java interfaces and classes allowing to read and write data from and to data sources - databases [38]) to access a relational database. The Java code (without any regard to security or best practices on accessing databases) would be something similar to the one shown in Fig. 1.

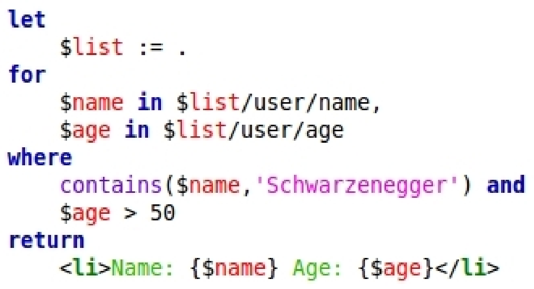
```
...
String query = "select ID, NAME, AGE " +
               "  from TESTTABLE t " +
               "  where t.NAME like 'Schwarzenegger' and " +
               "        t.AGE > 50";
Statement stmt;
stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery(query);
ResultSetMetaData rsmd = rs.getMetaData();

System.out.println("<ul>");
while (rs.next())
{
    System.out.println("<li>Name: " + rs.getString("NAME") +
                       " Age: " + rs.getInt("AGE") + "</li>");
}
System.out.println("</ul>");
stmt.close();
con.close();
...
```

Fig. 1: Query example using Java.

First, we declare a string variable to store the desired query. Then, an object of type Statement is used to execute the static query. After that, we begin the output, where the query results are displayed to the user. Data is displayed using a loop, creating an HTML list with results. Finally, we can release the resources used: the Statement and the connection to the database.

This thesis solves this task in a different way. Using technologies, such as XML and XQuery, the key idea is to build and deploy a whole web application using mostly XQuery programming language.

XQuery is a functional language designed to query and manipulate XML. It provides means to extract and manipulate data from XML. The XQuery language is derived from an XML query language called Quilt, which borrowed features from other languages, such as XPath, XQL, XML-QL, SQL and OQL [1]. Because XQuery speaks XML natively and sees output as a structured tree, not only string data, it is easy to create well-formed output with it. Thus, using XQuery and a native XML database, XTC [8] enables development of web-based application without facing the impedance mismatch problem. For example, the same query shown in Fig. 1 would be something similar to the one shown in Fig. 2.



```
let
    $list := .
for
    $name in $list/user/name,
    $age in $list/user/age
where
    contains($name,'Schwarzenegger') and
    $age > 50
return
    <li>Name: {$name} Age: {$age}</li>
```

Fig. 2: Query example using XQuery.

In this example, all we have to do is use a FLWOR expression. *For* binds the single items of a sequence to a variable; *Let* binds sequences to variables; *Where* filters nodes, based on a boolean expression; *Order by* sorts values; and *Return* gets evaluated once for every bound variable of the *For* clause. Both examples, the first in Java language, showed in Fig. 1, and the second in XQuery language, showed in Fig. 2, perform the same functionality. The result will be a list with all users whose name contains 'Schwarzenneger' and are older than 50 years. The XQuery implementation, however, is much easier to read and understand because it was designed for this purpose. The same situation will happen when compared to other programming languages, such as PHP, .Net, etc. [39].

The database used for this application is XTC[8]. XTC is an XML database management system. As such, it needs to be extended to work as an application server, supporting development of applications using XQuery. Its HTTP interface will be expanded and an entire structure for the development and deployment of applications with XQuery will be developed.

The remainder of this thesis is organized as follows. Section 2 describes the runtime environment, which is basically the application server, XTC, and its communication facilities. Besides that, it also describes how applications are deployed and structured in this approach. Section 3 introduces an XQuery application, developed and deployed on the server. Further details like application design, use cases and the necessary XQuery functions developed for general developing are presented. Section 4 compares, based on the already mentioned challenges, the development of applications with XTC against other known implementations of XQuery, like MarkLogic and Sausalito. Finally, section 5 concludes this thesis and discusses future work.

# 2 Runtime Environment

This chapter gives a brief introduction on the necessary infrastructure for the deployment of web applications. As already mentioned, XTC [8] is a database management system. Therefore, I had to extend it to work as an application server. For that, I focused mainly on the HTTP agent component, present in the server architecture. The main aspects that have been developed are: the creation of a pre-defined structure, a skeleton for developing applications on XTC; a structure built to handle HTTP requests; and extra features for extending the XQuery language, which, in the way it is defined, does not provide all necessary mechanisms for developing web applications. For example, XQuery does not provide mechanisms for accessing the database or mechanisms for gaining access and working with HTTP sessions. These features were developed when necessary. Some details about the server architecture and connectivity features are presented first.

## 2.1 XTC

The XML Transaction Coordinator (XTC) is a native XML database management system. Through extensive research in diverse areas, such as XQuery processing, XML storage, indexing techniques, transactions and self-tuning features, XTC allows to create, load and process XML files in a transactional environment.

It also provides several client interfaces with CRUD features - create, read, update and delete - on single and whole document collections, and primitives for ACID transactions, like “begin” and “commit”. Currently the system supports access via DOM, SAX, HTTP and FTP.

More details on particular features and actual research topics can be found in [8].

**2.1.1 Architecture** The architecture of XTC follows a five layer approach introduced by [7]. An overview of this architecture can be seen in Fig. 3.

The two bottom layers, the File Services layer (L1) and the Propagation layer (L2), manage external storage and buffers. The File Services layer works with data stored in containers (files arranged in a sequence of fixed-length blocks). This layer also provides read/write facilities between these containers and the main memory.

The Propagation layer implements buffers and assigns each container to a buffer manager (an array of pages). The buffer manager maps every block of a container directly to a page of the same size. This layer also handles page requests that come from the layer above.

The Access Services Layer (L3) implements the mapping from XML trees to the pages provided by the Propagation Layer, along with access and scan methods. The components responsible for it are: Index Services, responsible for storing records as key-value pairs; Catalog Manager, that keeps track of documents and meta-data collections, like document id and secondary indexes; Record Manager, that stores documents in a B-* tree index, called document index.

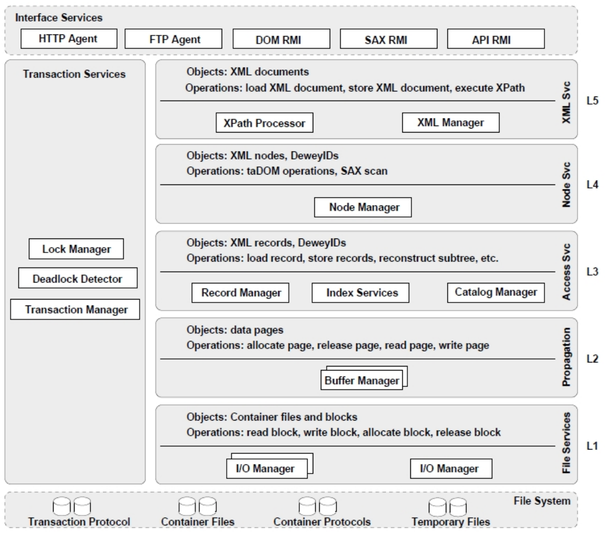


Fig. 3: XTC Structure [6].

The Node Services Layer (L4) serves as an interface provider for features of the Access Service Layer. Besides that, it also decodes internal records into external human readable XML and requests locks from the Lock Manager.

Finally the XML Processing Services layer (L5) provides high level functions like storage and reconstruction of documents and XPath evaluation.

This layer is also responsible for the metadata needed to manage XML documents and collections. The metadata mirrors the functionalities of a file system, enabling creation, removal, rename, etc, of directories and documents.

As can be seen in Fig. 3, the Transaction Services are connected to all layers. Its main responsibilities are transaction synchronization (i. e. lock manager and deadlock detector), logging for crash recovery and transaction rollback operations.

On top, the Interface layer provides a set of APIs, like HTTP, FTP, DOM, SAX and a proprietary library to interact with the system.

More details on the theoretical aspects of this structure can be found in [7].

## 2.2 XTC as Application Server

This chapter explains the activities that were necessary to transform XTC into an application server prototype. First I present the HTTP interface of the server. After that, I detail the used client-server model, then I give some explanation about the technology used to develop the internal structure. Ultimately, I present the used development patterns and the developed structure itself.

**2.2.1 HTTP Interface** The HTTP agent in the Interface Service layer is the central interface for this thesis. As already mentioned, users must be able to access the XQuery-based application somehow. Our server implements a client-server computing model and the HTTP Agent provides HTTP access to the application.

HTTP, Hyper Text Transfer Protocol, is the transport protocol used across the web. It specifies the message exchange between clients and servers. Its behavior is defined in RFC2616 [12]. The message structure follows the definitions of the old ARPA Internet Test Messages, defined in RFC822 [9].

With HTTP 1.0, a connection had to be established before a single request and response could be sent. Afterwards, the connection was released. This approach was adequate when pages were pure HTML text. Nowadays, every single web page consists of multiple content types, including images, style sheets, applets, etc. Establishing a connection to each of these objects is not appropriate.

With the arrival of HTTP 1.1, the support for persistent connections was added, that allows to send multiple requests over one single connection. It is also possible to pipeline requests; i.e, the second request can be issued before the first response arrives.

HTTP was designed more general than just to be used for the web, so methods were supported, instead of only requesting pages. The HTTP methods are GET, HEAD, PUT, POST, DELETE, TRACE, CONNECT and OPTIONS. This thesis gives most attention to GET and POST. While the first one requests some page information (it "gets" informations), the second sends data to be appended on the server-side (or "posts" data to the server).

### 2.2.2 Client-Server Model

In a client-server model, we have typically two actors: the client and the server. The client connects to the server and requests a resource, typically specifying a particular page. The server accepts the connection from the client, resolves the name of the requested resource to load, e.g, from the local disk and finaly returns the resource to the client and releases the connection.

Some of the responsibilities addressed by the our server are:

**Resolve requests**

A requested page might not be directly mapped to a file in the server. For example, when a user tries to access "http://localhost/apps/appName/default/" the application server may infer that the file "default.xq" is requested, even though its complete name is not in the request. Also, the resource location might not be directly mapped from the structure described in the URL to the physical storage. The server must resolve this name to the real path.

**Authentication and Authorisation**

The identity of the client should be validated. This step is necessary for access control to resources with confidential or private content.

Sometimes, not only the identity has to be validated, but also the place from where the request has been made. In other words, there might be restrictions on access from outside the company, and these accesses should be denied somehow.

**Resource management**
When a request reaches the application server, the server must, e.g, access its disks to retrieve resources and send them back to the clients. The server cannot serve more requests per second than it can make disk accesses. If the number of requests increases and exceeds the number of accesses the server can make, the subsequent requests will have to wait more and more for their responses. One simple improvement is to maintain a cache in memory, as already clarified in the XTC architecture 2.1.

**Determine the MIME type**
The acronym means Multipurpose Internet Mail Extensions, and the main idea is to define encoding rules for non-ASCII content. It is defined in RFC1341 [10] and RFC2045 [11] and is widely used, not only for Mail purposes. Web browsers, e.g, use it to display or output files that are not in HTML format. Because XTC provides storage for different files types, the right MIME type for the requested page must be specified.

**Miscellaneous**
Other kind of activities can be performed by the server, such as build user profiles or gather statistics, and so on and so forth.

To solve these server-side tasks in a simple, composable and reusable way, one good alternative is the use of Java Servlets and Java Development Patterns, because they provide component-based and platform-independent methods to develop web applications.

## 2.3 Servlets

A servlet is a Java class that aims to extend request-response server capabilities. Servlets are commonly used to build web servers.

The javax.servlet [31] and javax.servlet.http [32] packages provide classes and interfaces for writing servlets. All servlets implement the servlet interface [33].

I used a servlet-based architecture to manage the client-server communication. Servlets can be seen as an extension to a server and are also efficient, because after they are loaded, the server stores it as an object instance on memory. If multiple requests are needed, each request is managed by a different thread, so they are also scalable [15]. Servlets have also other advantages, like:

- Strong typing, inherited from the Java language
- Safe error handling, based on Java's exception handling mechanism
- Clean, object-oriented and modular coding

– Integration with the server let servlets address problems like file path translations, logging, check authorisations, map MIME types, among others.

This thesis focuses on the specialized HttpServlet class, that provides methods like doGet, do Post, doHead, doOptions, doPut and doTrace. These methods correspond directly to the HTTO protocol methods [14].

## 2.4 Development patterns

The design of this servlet-based structure I created is the most relevant aspect of the whole framework. Over the years, a series of guides concerning best practices related to architecture and design of applications with JEE technologies was created, like [2] and [23]. Strongly based on these design approaches, I use mainly two of the patterns which I describe now: the Front Controller and the Intercepting Filter pattern.

### 2.4.1 Front Controller

The main idea behind the Front Controller pattern concerns how requests will be handled. This pattern proposes a centralized way of solving this situation, a single entry point to the system. It achieves, among others:

– Centralized retrieval and manipulation of data to the presentation-tier [1] layer.
– Centralized handling of requests, easing the control and log of user's system usage.

The pattern proposes a single controller as the initial entry point for client requests. The controller manages common cross-cutting tasks like authentication, authorisation, delegation of processes and error handling. This centralized control promotes reuse across requests and increases maintainability. For example, control of authentication and authorisation does not have to exist on every single web page and whenever this control needs updates, the updates will be made only in one central place.

The pattern also provides a dispatcher component, which is responsible for navigation. Thus, it steers the control flow of the application, i.e., it chooses the next view for the presentation layer.

The Front Controller pattern suggests a centralized component for the management of requests.

The two other main components are the *View* and the *Helper*. The *View* represents the information presented to the user. The *Helper* is responsible for

[1] The presentation-tier is basically the user interface or topmost level of the application. The main goal is to translate results of request into something the user can understand.

helping a view or controller to complete their tasks. The *Helper* responsibilities include also data gathering, required by the view. The helper may adapt data and, afterwards, deliver this manipulated data to the view. For example, it can provide either access directly to the raw data or to data formated as a web content.

Fig. 4 shows an overview of this pattern main structural ideas.

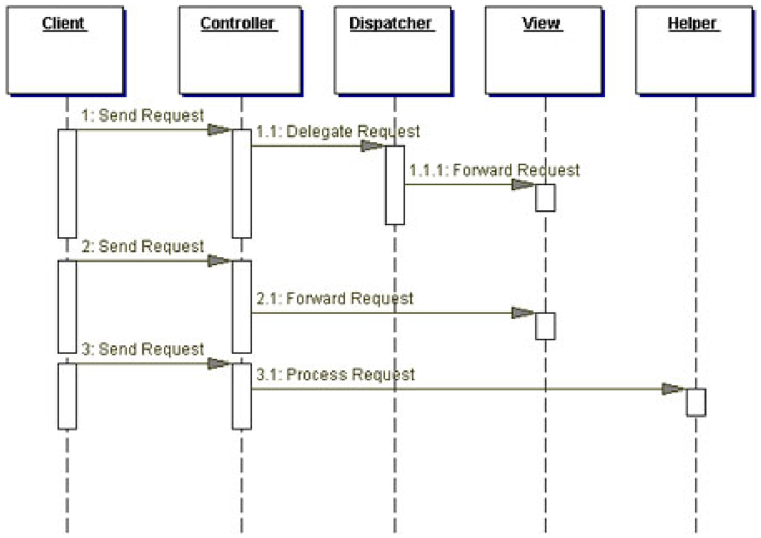


Fig. 4: Front Controller sequence diagram [2].

### 2.4.2 Intercepting Filter

The main concern of the pre- and post-processing of web requests. Usually, in any kind of web application, a request must pass some checks before it can be processed, for example:

- Authorisation
- Session validation
- Location validation (related to IP localization)

A naive solution for the realization of these checks would be a cascade of conditional expressions at the beginning of each operation. When a check evaluates to false, the processing stops or the user is displayed an appropriate error message. However, this solution leads to code fragility and a copy-and-paste style of programming [2].

The Intercepting Filter Pattern instigates the following idea: a simple mechanism to add and remove processing components, in a flexible and unobtrusive manner. Pluggable filters, that intercept requests and responses, may be added or removed unobtrusively, without changing the existing code.

When a user accesses a resource, the FilterManager intercepts this request and passes it to a FilterChain. A FilterChain consists of a series of filters, such as authorisation, location, internal dependencies, etc. You also define the filters processing order. Afterwards, the FilterManager forwards the request to its original target. Fig. 5 shows an overview of this structure.

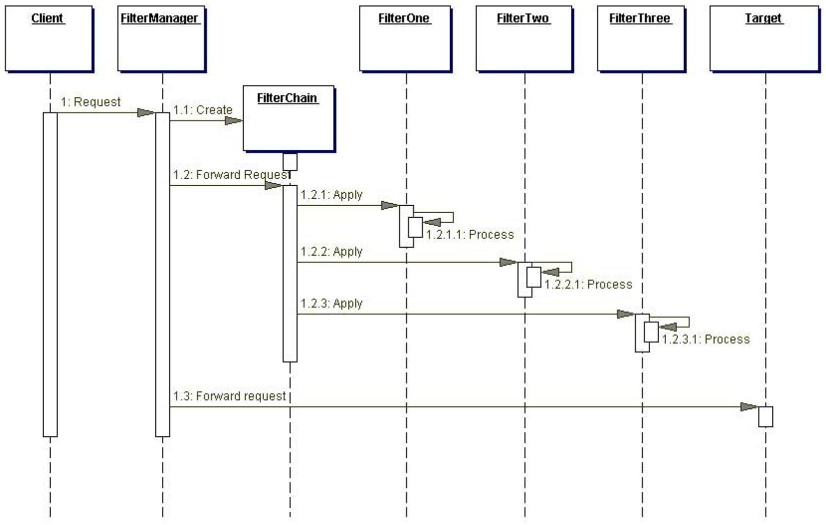


Fig. 5: Intercepting Filter sequence diagram [2].

### 2.4.3 Proposed Approach

The proposed approach assembles the best of both patterns. I implemented the structure proposed by the Front Controller pattern, showed in Fig. 4. Besides that, I incorporated the facilities introduced by the Intercepting Filter pattern, based on the principles I have already shown on 2.4.2.

The resulting architecture is composed of tree servlets, the Controller, the Dispatcher and the AppError, and of a filter, the AuthorisationFilter. Fig. 6 illustrates such architecture.

To identify the servlet responsible for a request, the server uses a mechanism based on the requested URL. Through a "longest match rule" we can decide which servlet will receive the request. Suppose we have the following two prefixes, registered to different servlets:

– Servlet A: http://localhost/app/*

– Servlet B: http://localhost/app/longerPath/*

In this example, the server routes a request for “http://localhost/app/longerPath/existingPage” to servlet B, because the longest complete match is with “http://localhost/app/longerPath/*”. Similarly, it routes a request for “http://localhost/app/otherPath/otherExistingPage” to servlet A.

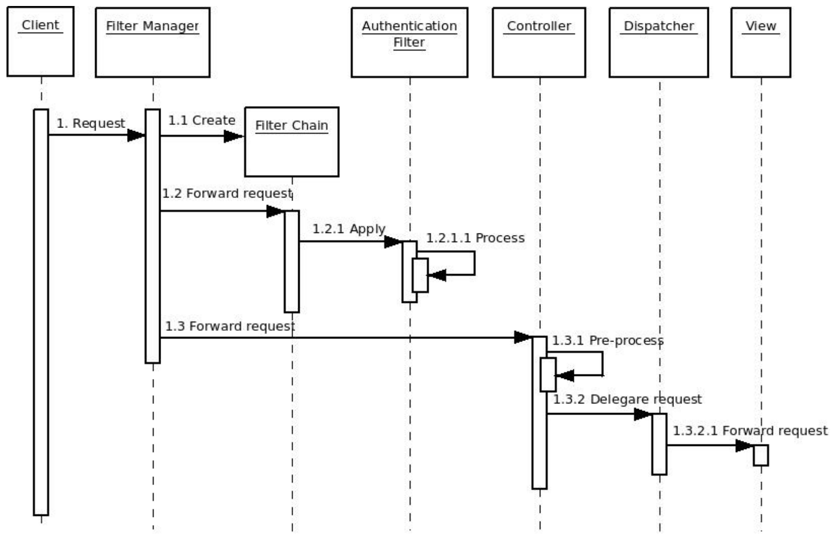


Fig. 6: Proposed Structure diagram.

An incoming request is first intercepted by the FilterChain. This mechanism is almost the same described in the Intercepting Filter Pattern, in section 2.4.2 and it allows processing components to be added and removed, in a flexible and unobtrusive manner. These filters are the first components to work with the request, which means that any undesirable behavior can be surpassed.

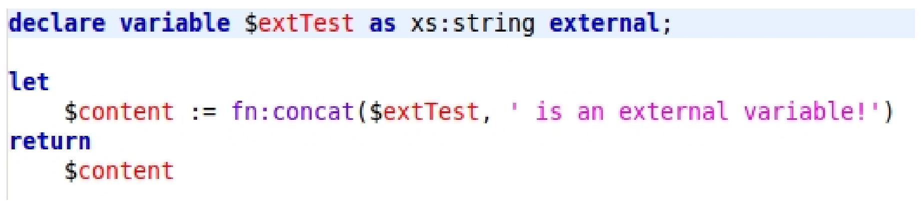
The request is then handled by the only filter present on the current version of the server, the AuthorisationFilter. It works as a wall against unauthorised users. While public resources do not require authorisation mechanisms, private resources actually rely on simple validation functions, such as: “checkUser(‘user’,‘pass’)”. This function would need to be called for every resource that needs access privileges to be reached. The AuthorisationFilter solves exactly this task. With a centralized approach, every resource under the private structure of the application is only accessible if an authentication is verified and the authorisation is granted. In other words, only authorised users access these resources.

After passing through the filter chain, the request reaches the Controller servlet. Its main purpose is to extract basic information about the request, such as the requested application name and the requested resource from the URI of the request. The servlet makes this information available for the further processing. The Controller servlet also sets the response content type ("application/x-html+xml; charset=UTF-8"). RFC 3236[13] gives more information about this content type.

Afterwards, the Controller passes the request to the servlet Dispatcher. The Dispatcher servlet receives the preprocessed request and chooses the right action to do with it. "Right action" means select the right resource from the server and bind all necessary external variables to it.

Till this point, I explained the path of the request through the framework structure. Now, the actual application comes into play. The requested resource is actually an XQuery page, stored in the application structure, which is described in 2.5.1. The Dispatcher selects all external variables in the XQuery page and checks all request parameters names for matches with these variables. The servlet takes into account only the variable name, which means that each variable declared as external in a XQuery page should have a corresponding parameter in the request with the same name. For example, the external variable declaration showed in Fig. 7 would be automatically bound to a request parameter called "extTest"[2]. In one hand we lose here a bit of flexibility, because the name of external variables must match the name of a parameter, but on the other hand, manual mechanisms to select HTTP parameters are not necessary. The automatic binding makes development easier and faster.

Finally, the Dispatcher evaluates the query and displays the result to the user.



```
declare variable $extTest as xs:string external;

let
    $content := fn:concat($extTest, ' is an external variable!')
return
    $content
```

Fig. 7: External variable declaration.

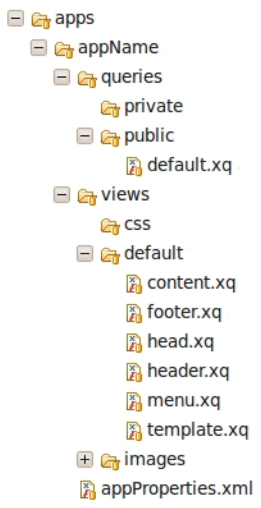
Besides that, application errors are handles by the AppError servlet, which is invoked, whenever an application component, e.g., a query, terminates with an exception. It shields the user from details of the error, like a stack trace, and displays appropriate error messages.

[2] Because XQuery allows declaration of external variables, binding them is necessary. Binding means that these variables must receive a value from external environment before the query can be evaluated.

## 2.5 Deployment

Each application developed for our platform follows a specific structure. This structure aims for simplicity: in one hand, design simplicity for new applications and in the other hand, easiness of deployment.

**2.5.1 Application Structure** Applications use a standard directory structure, shown in Fig. 8. This structure also ensures the existence of a root user, a manager, who has some access privileges. Its semantic is defined as: the user who owns the application. The resources under /appName/queries/private/ are only accessible after the manager is properly authenticated. The main structure is described below:



```
apps
  appName
    queries
      private
      public
        default.xq
    views
      css
      default
        content.xq
        footer.xq
        head.xq
        header.xq
        menu.xq
        template.xq
      images
    appProperties.xml
```

Fig. 8: Application structure.

**/appName**

The *appName* is the root folder of the application. The name of an application is the folder's name, in this case: appName. Users access applications using this name as final part of the URL, for example: "http://myServerLocation.com/apps/appName/resourceName".

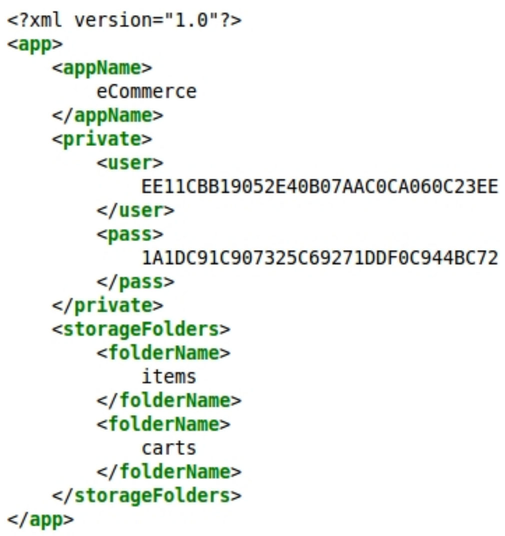
**/appname/queries**

The *queries* folder stores files containing the XQuery queries, used to define the business logic. Each file contains exactly one query. When the user accesses a query, it is executed by the server. Here, two different access types exist: private or public queries. Private queries can be accessed only by an authenticated user of the application and public queries are publicly accessed. Section 2.4.3 describes the mechanism for access restriction.

**/appname/views**

The *views* folder contains internal queries, images and other resources. Three basic folders coexist here: a “CSS” folder, where CSS style sheets are stored; an “images” folder, where image resources are stored; and a “default” folder, where internal queries are stored. These queries are located under a “default” folder and are internally accessed by predefined functions. Resources under the “CSS” and “images” folders are stored under XTC database at deployment time and are delivered as plain/uninterpreted binary content through HTTP request.

The structure contains also a configuration file, appProperties.xml. This configuration file stores information about the application, such as the manager username and password (both encoded as MD5 hash sequence [3]), the application name - which must match the main folder name - and the storage folders that need to be created under the XTC database. An example of configuration file is shown in Fig. 9.



```xml
<?xml version="1.0"?>
<app>
    <appName>
        eCommerce
    </appName>
    <private>
        <user>
            EE11CBB19052E40B07AAC0CA060C23EE
        </user>
        <pass>
            1A1DC91C907325C69271DDF0C944BC72
        </pass>
    </private>
    <storageFolders>
        <folderName>
            items
        </folderName>
        <folderName>
            carts
        </folderName>
    </storageFolders>
</app>
```

Fig. 9: Application properties file.

[3] MD5 - Message-Digest Algorithm 5 - is a cryptographic hash function with a 128-bit hash value. More details about it are available in [24].

The existence of this kind of structure simplifies application development, because the developer can focus solely on the application development. The developer also takes advantage of several predefined queries and resources.

The deployment of applications under XTC is rather easy. All that is necessary is the existence of a proper folder structure, as shown in Fig. 8. This folder must be placed under the "/apps/" folder. On startup, the server scans all applications underneath the "/apps/" folder, checks their structure, creates the necessary metadata, stores the necessary images and files and makes the application accessible.

So far, XTC provides only this deploying method. Some extra features are desirable, such as "hot deployment" which is the ability to deploy a new application, or re-deploy and existing one, without the need to stop the server. Nowadays, application servers like Tomcat [22] provides this extra feature. Tomcat allows local and remote deployment and does not require a restart.

# 3 XQuery application

This chapter focuses on the development of a web application, showing that the runtime environment is suited for developing web applications using XQuery. As showcase, it demonstrates how to create a simple e-commerce platform. First, I discuss some well-known techniques for developing software, whose teachings have provided good ideas and methodologies for software development. Then, I illustrate the business rules and use cases, which have to be modelled in the sample application. Furthermore, I introduce a series of extra functions, that enables the application to interact with the database and the servlet framework. Finally, I show some code snippets of the application, illustrating how the development is done using XQuery.

## 3.1 Software Development

Over the past years, several writers proposed development models that increase development success rate. This thesis gives a brief on three of them: the waterfall model, the spiral model and the RAD model. Other development models exist and are shown in [26].

The waterfall model is a sequential development approach. The model suggests a sequential organization of the development activities. Royce [25] introduced this approach, even tough he did not name it as waterfall. Its main phases are system and software requirement, analysis, program design, coding, testing and operation. More details about this approach are shown in [25][26].

The spiral model corrects the evident lack of flexibility in the waterfall model. The sequential order of the waterfall model is inconsistent with real situations, where typically activities need to be repeated two or more times [26]. Boehm [27] proposes a spiral model which is an incremental development process. More details about this approach are shown in [26][27]. An overview of this model is shown in Fig. 10.

The RAD model, Rapid Application Development, provides minimal planning in favor of rapid prototyping. Prototypes are typically samples, basis or primitive forms of the desired goal. Because prototypes are designed, the RAD model achieves fast development. It also emphasizes on completing the business need, in despite of technological or engineering excellence. More details about this model are shown in [28].

I used parts of these three developing models. I took advantage of the concepts of planning and designing, originally from the waterfall model. I also reorganize these ideas in parallel with the development, as in the spiral model. Besides that, I made the whole development and testing as prototypes, achieving fast development, as suggested by the RAD model.

## 3.2 Application Design

The developed application itself consists of an e-commerce web platform for a given market store, the Gaucho's Market. The platform allows clients to buy

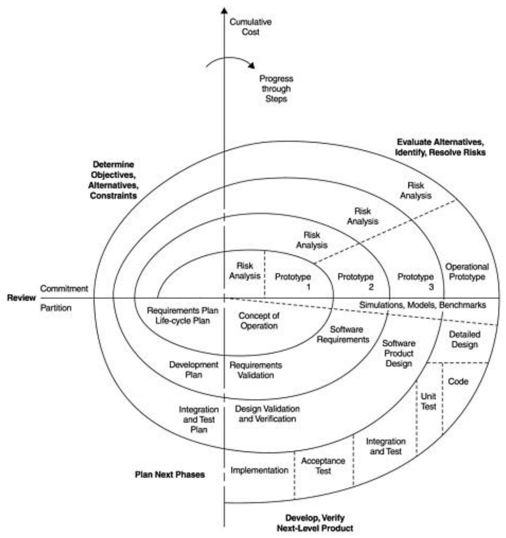


Fig. 10: Spiral Model introduced by [27].

items, which are offered by the platform manager. Because the application itself is not the focus of this thesis, more specific details are irrelevant.

The buying part is split up into the following use cases: Clients of the Gaucho's Market can search items [4], look items properties, add items to a private shopping cart and buy the items in the shopping cart.

The offering part, creates the following cases: The manager of the Gaucho's Market e-commerce platform can create, update, remove and search items.

I chose these business rules and use cases in order to address the widest possible range of challenges, from the point of view of storage and data access. Challenges, such as access, modification and removal of data from the database, in addition to transaction mechanisms, function calls and usage parallelism are addressed. An overview of this application use cases is shown in Fig. 11 and the use cases are shown in sections 3.2.1 to 3.2.9.

I chose UML to specify the use cases. I also represented every use case with an activity diagram. Activity diagrams are useful for presentations, fast to develop - because they are informal - and show a visual idea of activities, thus helping during the development process.

### 3.2.1 Search item

**Actors**

Manager, User.

[4] Items are a generalization of any kind of product that are sold online. Because the nature of the items is not relevant, I call them simply items.

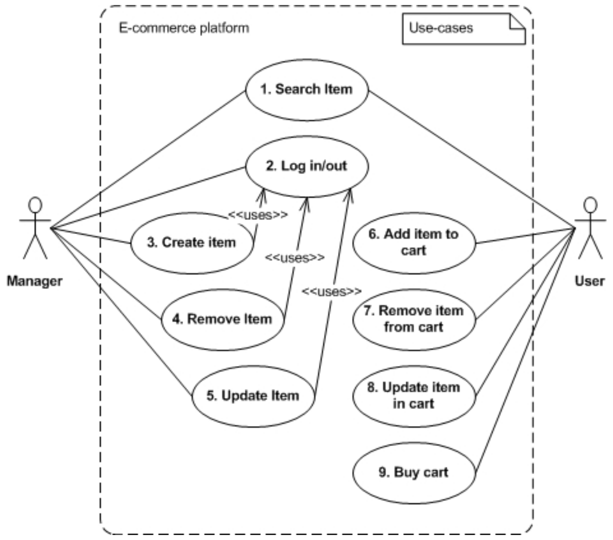


Fig. 11: Gaucho's Market use cases.

**Goal**
Provides a result list of items, according to the searched string key.

**Summary**
Provides basic search functionality.

**Dependencies**
None.

**Basic path**
1. The user accesses the search items page.
2. The search page displays a list of items, according to a predefined priority rule.
3. The user requests information about some specific item, through a string searching facility.
4. The application queries the data storage for the given string.
5. The server responds with the resulting data.
6. The application displays the resulting items to the user.

An overview of the search use case is shown in Fig. 12.

### 3.2.2 Login

**Actors**
Manager.

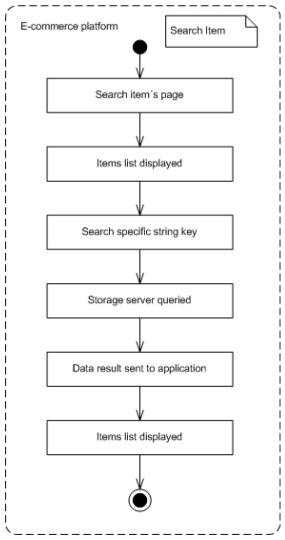


Fig. 12: Search Item activity diagram.

**Goal**

Establishes privileges and administrative access.

**Summary**

The Gaucho's Market is accessed by two different user groups: customs, that use the platform to buy, search, etc. items from the platform; and a manager, that manages the platform. The login mechanism authenticates and authorises managers, giving them access to resources with access restriction.

**Dependencies**

None.

**Basic path**

1. The manager accesses the login web page.
2. The manager enters username and password information.
3. The servers processes the login successfully.
4. The application displays login confirmation.

**Abnormal path**

1. The manager accesses the login web page.
2. The manager enters username and password information.
3. The servers processes the login unsuccessfully.
4. The application redirects the user to the login web page.

An overview of the login use case is shown in Fig. 13.

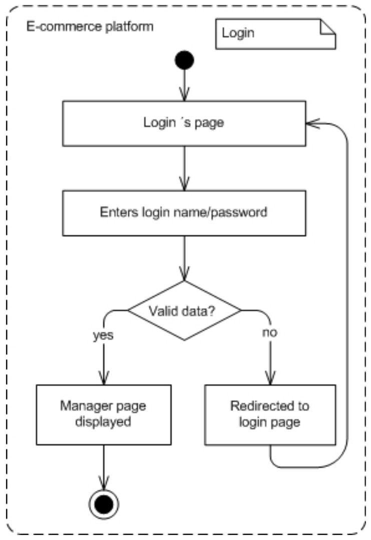


Fig. 13: Login activity diagram.

### 3.2.3 Create item

**Actors**

Manager.

**Goal**

Creates new items at the e-commerce platform.

**Summary**

Provides an item creation facility. Every item is defined by a name, a description and a price. The Manager is able to create different new items.

**Dependencies**

3.2.2.

**Basic path**

1. The manager accesses the create items web page.
2. The manager fills the item creation form information.
3. The manager sends the data to the application server.
4. The server validates the data successfully.
5. The server redirects the manager to the just created item page.
6. The application displays the new item.

**Abnormal path**

1. The manager accesses the create items web page.
2. The manager fills the item creation form information.

3. The manager sends the data to the application server.
4. Validation fails.
5. The application displays the invalid information.
6. The server redirects the manager to the create items web page.

An overview of the create item use case is shown in Fig. 14.

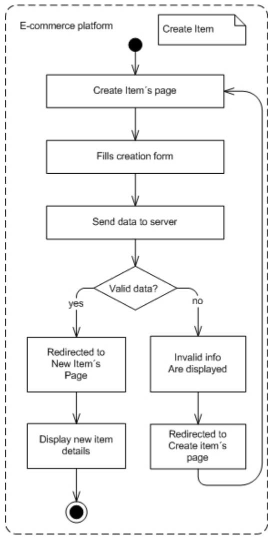


Fig. 14: Create item activity diagram.

### 3.2.4 Remove item

**Actors**

Manager.

**Goal**

Removes existing items from the e-commerce platform.

**Summary**

Provides an item deletion facility. The Manager is able to delete all existent items.

**Dependencies**

3.2.2.

**Basic path**

1. The manager accesses the remove items web page.

2. The application asks the manager confirmation for the deletion.
3. The manager confirms the deletion.
4. The server removes the item from the storage.
5. The server redirects the manager to the initial item listing page.

**Abnormal path**

1. The manager accesses the remove items web page.
2. The application asks the manager confirmation for the deletion.
3. The manager does not confirm the deletion.
4. The server redirects the manager to the initial item listing page.

An overview of the remove item use case is shown in Fig. 15.

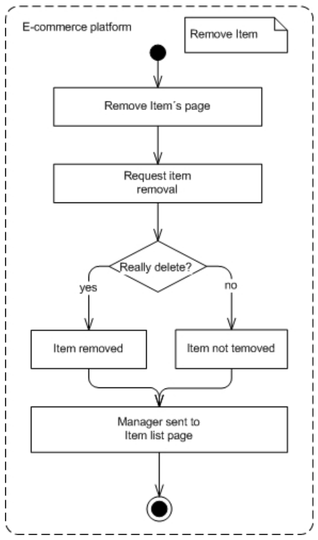


Fig. 15: Remove item activity diagram.

### 3.2.5 Update item

**Actors**

Manager.

**Goal**

Updates existing items in the e-commerce platform.

**Summary**

Provides an item update facility. The Manager is able to update all existent items.

**Dependencies**
3.2.2.

**Basic path**

1. The manager accesses the edit items web page.
2. The application displays the item details for edition.
3. The manager edits the item.
4. The manager sends the data to the application server.
5. The server validates the data successfully.
6. The server redirects the manager to the item's page.
7. The application displays the updated item's information to the manager.

**Abnormal path**

1. The manager accesses the edit items web page.
2. The application displays the item details for edition.
3. The manager edits the item.
4. The manager sends the data to the application server.
5. Validation fails.
6. The application displays invalid information to the manager.
7. The server redirects the manager to the edit items web page.

An overview of the update item use case is shown in Fig. 16.

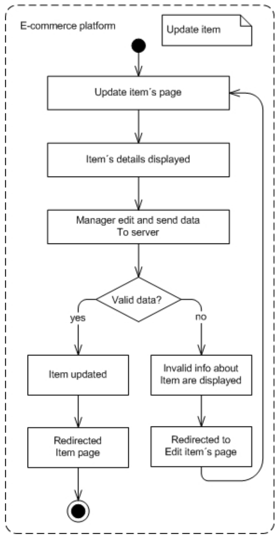


Fig. 16: Update item activity diagram.

### 3.2.6 Add item to cart

**Actors**
User.

**Goal**
Adds the desired item to the current client shopping cart.

**Summary**
A shopping cart defines the set of products to buy. The add to cart activity adds the desired item to your current shopping cart.

**Dependencies**
None.

**Basic path**
1. User accesses the item's detailed web page.
2. Application displays item's details.
3. User adds the item to the cart.
4. Application displays message about the item added to the cart.
5. Server redirects user to the item list page.

An overview of the add item to cart use case is shown in Fig. 17.

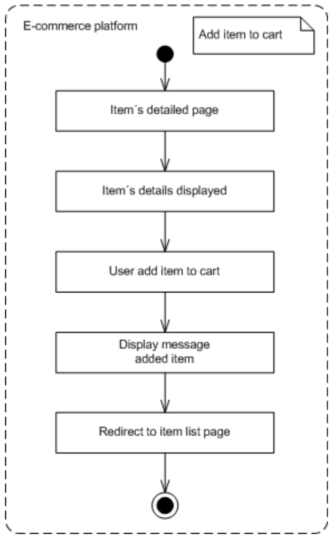


Fig. 17: Add item to cart activity diagram.

### 3.2.7 Remove item from cart

**Actors**
User.

**Goal**
Removes an existing item from the cart.

**Summary**
Provides an item removal facility. The user is able to remove all existent items from the shopping cart.

**Dependencies**
None.

**Basic path**
1. User accesses cart list items web page.
2. Application displays cart items.
3. User removes item from the cart.
4. Application asks confirmation for the deletion.
5. User confirms deletion.
6. Application removes item from cart.
7. Server redirects the user to the cart list items page.

**Abnormal path**
1. User accesses cart list items web page.
2. Application displays cart items.
3. User removes item from the cart.
4. Application asks confirmation for the deletion.
5. User does not confirm deletion.
6. Server redirects the user to the cart list items page.

An overview of the remove item from cart use case is shown in Fig. 18.

### 3.2.8 Update cart item

**Actors**
User.

**Goal**
Updates an existing item in the cart.

**Summary**
Provides an item updating facility. The user is able to update all existent items in the shopping cart.

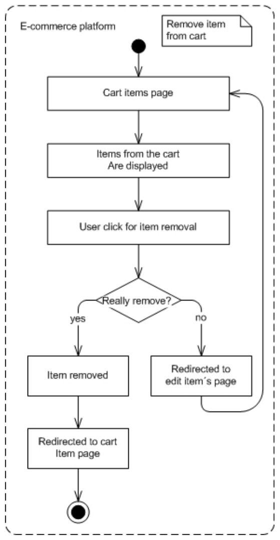

E-commerce platform
Remove item from cart
Cart items page
Items from the cart Are displayed
User click for item removal
Really remove?
yes
no
Item removed
Redirected to edit item´s page
Redirected to cart Item page

Fig. 18: Remove Item from cart activity diagram.

**Dependencies**

None.

**Basic path**

1. User accesses edit item in cart web page.
2. Application displays item's details.
3. User edits item.
4. User sends data to the server.
5. Valid data.
6. Server redirects user to the cart list items web page.
7. Application displays updated item's information

**Abnormal path**

1. User accesses edit item in cart web page.
2. Application displays item's details.
3. User edits item.
4. User sends data to the server.
5. Data not valid.
6. Application displays invalid information to the User.
7. Server redirects user to edit item in cart web page.

An overview of the update item in cart use case is shown in Fig. 19.

### 3.2.9 Buy cart

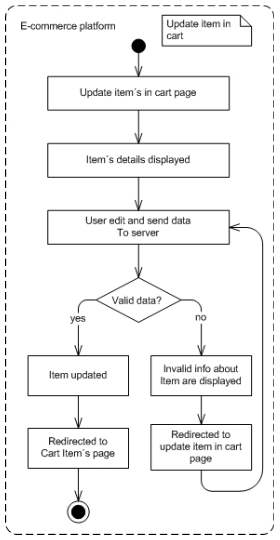

E-commerce platform
Update item in cart
Update item´s in cart page
Item´s details displayed
User edit and send data To server
Valid data?
yes
no
Item updated
Invalid info about Item are displayed
Redirected to Cart Item´s page
Redirected to update item in cart page

Fig. 19: Update cart item activity diagram.

**Actors**
User.

**Goal**
Buys the items in the cart.

**Summary**
Provides a buying cart facility. The user is able to buy the items of its shopping cart.

**Dependencies**
None.

**Basic path**
1. User accesses buy cart web page.
2. Application displays current cart item's, together with a user information form.
3. User fills in the form with his personal information.
4. User sends data to the server.
5. Valid data.
6. Server redirects user to the cart list payment options.
7. User chooses among different payment options.
8. Valid payment options.
9. Server registers the buying for future processing.

**Abnormal path**

Abnormal paths, in this case, consist of validation errors in multiple paths. These situations are better viewed on the activity diagram of this use case, shown in Fig. 20.

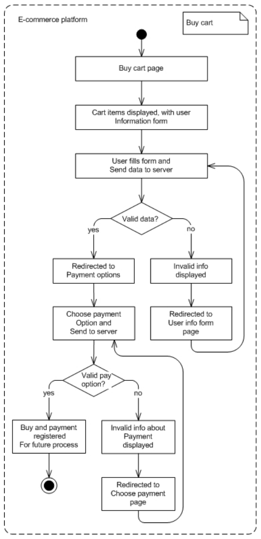

E-commerce platform
Buy cart
Buy cart page
Cart items displayed, with user Information form
User fills form and Send data to server
Valid data?
yes
no
Redirected to Payment options
Invalid info displayed
Choose payment Option and Send to server
Redirected to User info form page
Valid pay option?
yes
no
Buy and payment registered For future process
Invalid info about Payment displayed
Redirected to Choose payment page

Fig. 20: Buy cart activity diagram.

## 3.3 Application Implementation

Based on the use cases shown in sections 3.2.1 to 3.2.9, this section sketches the implementation of the Gaucho's Market application in XQuery. First, it gives an overview of the application, than it shows some extra functions necessary during the development and finally it shows some code snippets, in order to get a taste of the implementation style.

**3.3.1 Overview** The structure of the application is shown in Fig. 11. Each of these use cases is translated into one or more queries formulated in XQuery. For more complex cases, I used more than one query, aiming for simplicity, instead of larger queries with more complex code. These cases are the ones where interaction with the user is necessary, e.g., item creation and buying the shopping cart.

The development of Gaucho's Market application involves 17 different queries, besides some default queries shown in the default structure in Fig. 8. These 17 queries are divided in two groups: 6 are private and 11 are public. Private queries concern application management, used by the Manager, such as creation, removal and update of items. Public queries concern application usage, such as adding items to cart, searching for items and the actual purchase of items.

During the development phase, some extra features were needed, because XQuery itself does not provide certain functionalities, such as document management and metadata creation in the XTC database, dynamic evaluation of queries or interoperability with HTTP objects.

**3.3.2 Application server API** This section shows suplementary XQuery functions that were created to allow the XQuery application to interact with the runtime environment, i.e., the application server. The functions have been created based on needs that emerged during the XML development. For better organization of the tasks, I used the concept of namespaces, which are predeclared prefixes [1], achieving better classification of these extra features. I created the following namespaces: *http*, *xtc* and *util*. The *http* namespace comprises functions related to the HTTP protocol extended by XTC. The *xtc* namespace unifies functions related specifically to the usage of the XTC database and the *util* namespace groups miscellaneous utility functions.

*3.3.2.1 Document Storage*
The *storeFile* function stores a document in XTC. It takes as parameters the full path name of the resource to be stored and the content of this resource, and returns the stored XML document. The specified content must be a well-formed XML node.

```
xtc:storeFile($name as xs:string, $content as node()) as node()
```

*3.3.2.2 Document Removal*
The *deleteFile* function removes a document from the XTC database. It takes the full path name of the resource, deletes it from the database and returns true in case of success and false otherwise.

```
xtc:deleteFile($name as xs:string) as xs:boolean
```

*3.3.2.3 Document Access*
The *loadFile* function accesses an already existent document from the file system, not from the XTC database. It receives the full path name of the resource to be loaded and returns the file as a string result.

```
xtc:loadFile($name as xs:string) as xs:string
```

*3.3.2.4 Dynamic Evaluation*
The *eval* function is used to execute a dynamically constructed XQuery expression, inside a running XQuery script. It is very useful when, for example, an XQuery page generates queries based on HTTP parameters passed by the user. The eval function receives a query and returns the result of this query.

```
xtc:eval($expr as node()) as node()
```

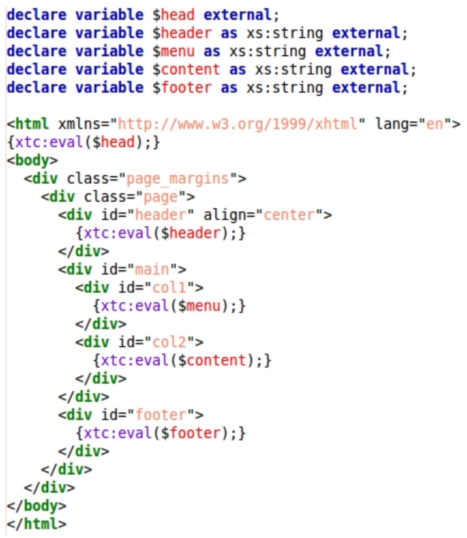
#### *3.3.2.5 Directory Creation*

The *makeDirectory* function creates a virtual directory in the metadata catalog of XTC, as already explained in section 2.1.1. It takes the full directory path name and returns true in case of successful directory creation and false otherwise.

```
xtc:makeDirectory($fullPathName as xs:string) as xs:boolean
```

#### *3.3.2.6 Template Creation*

The dynamic generation of HTML code is a key task in web applications. The problem is that, most of the times, the same HTML code is repeated in the development of different pages. The *template* function aims to resolve this problem, increasing reusability of HTML code. The function *template* works with a predefined XQuery page and renders five different external variables, head, header, menu, content and footer. This query can be seen in Fig. 21.



```
declare variable $head external;
declare variable $header as xs:string external;
declare variable $menu as xs:string external;
declare variable $content as xs:string external;
declare variable $footer as xs:string external;

<html xmlns="http://www.w3.org/1999/xhtml" lang="en">
{xtc:eval($head);}
<body>
  <div class="page_margins">
    <div class="page">
      <div id="header" align="center">
        {xtc:eval($header);}
      </div>
      <div id="main">
        <div id="col1">
          {xtc:eval($menu);}
        </div>
        <div id="col2">
          {xtc:eval($content);}
        </div>
      </div>
      <div id="footer">
        {xtc:eval($footer);}
      </div>
    </div>
  </div>
</body>
</html>
```

Fig. 21: Default query used by template function.

The function has two different signatures: the first receives five parameters, head, header, menu, content and footer. Here, all parameters will be bound to the template query variables, as shown in Fig. 21. All parameters must be well-formed XQuery expressions so the function can work properly. The second signature receives only the main content of the page, and displays it according to the already shown query. The other variables, $head, $header, $menu and $footer are bound to defautl resources. These resources are *head.xq*, *header.xq*, *menu.xq* and *footer.xq* respectively, and are present under the */views/default* folder, shown in Fig. 8.

The usage of the function is as follow.

```
xtc:template($content as node()) as node()

or

xtc:template($content as node(),
             $head as node(),
             $header as node(),
             $menu as node(),
             $footer as node()) as node()
```

*3.3.2.7 Session Attributes*

The HTTP protocol is by definition stateless, but usually state information must be kept between requests. The HTTP session stores this kind of information, but the XQuery standard does not provide any mechanism to use or access it[1]. Therefore, the creation of functions to support session features became necessary. The basic functions are create, read and remove of session attributes.

The *setSessionAtt* function creates a new session attribute. It takes the attribute name and content. It returns true in case of success, false otherwise.

```
http:setSessionAtt($name as xs:string,
                   $obj as  node()) as xs:boolean
```

The *getSessionAtt* function reads a session attribute. It takes the attribute name and returns the session attribute, if it exists.

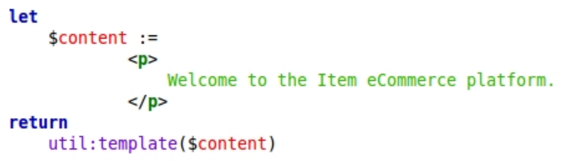
```
http:getSessionAtt($name as xs:string) as node()
```

The *removeSessionAtt* function removes an attribute from the session. It takes the attribute name and returns true when the attribute is successfully removed, false otherwise.

```
http:removeSessionAtt($name as xs:string) as xs:boolean
```

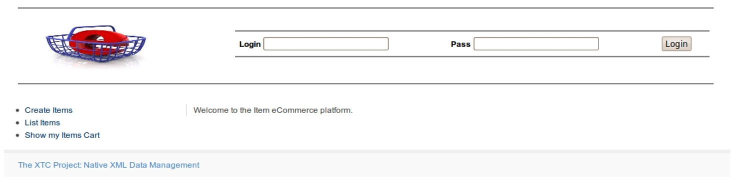
**3.3.3 Examples** To better understand how the above presented functions, as well as the development of applications using XQuery, this section shows some of the developed code. It provides a step by step guide on how to create an item, illustrating the running XQuery code, along with the result of each XQuery, i.e., the HTML screens displayed to the user.

First, the user accesses the e-commerce platform using the URL `http://localhost:24280/app/eCommerce/default/`, the server executes the respective query *default.xq*, and, as a result, it shows the home page of the store. Fig. 22 shows both, the query and its result.



```
let
    $content :=
            <p>
                Welcome to the Item eCommerce platform.
            </p>
return
    util:template($content)
```

(a) XQuery code.



- Create Items
- List Items
- Show my Items Cart

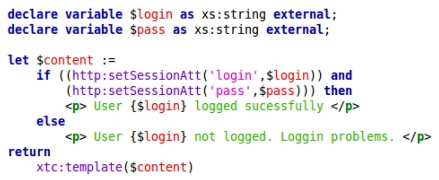
Welcome to the Item eCommerce platform.

The XTC Project: Native XML Data Management

(b) HTML result.

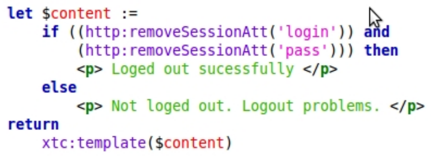
Fig. 22: Default application page.

Fig. 22 also presents the template function in action, which simplifies development. Managers must log in before they can access item maintenance functions. Ergo, the login form is present on the default page. The code for the application login and logout is showed in Fig. 23.



```
declare variable $login as xs:string external;
declare variable $pass as xs:string external;

let $content :=
    if ((http:setSessionAtt('login',$login)) and
        (http:setSessionAtt('pass',$pass))) then
        <p> User {$login} logged sucessfully </p>
    else
        <p> User {$login} not logged. Loggin problems. </p>
return
    xtc:template($content)
```

(a) XQuery login code.



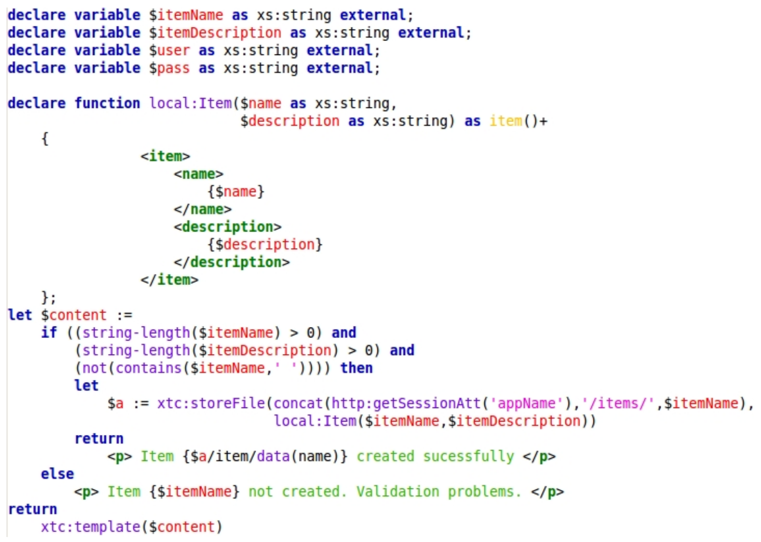
```
let $content :=
    if ((http:removeSessionAtt('login')) and
        (http:removeSessionAtt('pass'))) then
        <p> Loged out sucessfully </p>
    else
        <p> Not loged out. Logout problems. </p>
return
    xtc:template($content)
```

(b) XQuery logout code.

Fig. 23: XQuery code for login and logout features.

The login and logout queries show, also the usage of session control functions, such as add or remove attributes. After logging in properly, the user proceeds to the item creation page. This page requests the items name and description. The user fulfills and submits the form, allowing the server to create the item. Finally, fig. 24 presents the executed XQuery code, to process user input and to create a new item in the database.

In this example, I presented definition and usage of local functions, as well a new function to access session attibutes, *getSesstionAtt*. This example also demonstrates how to store documents using *storeFile*.



```
declare variable $itemName as xs:string external;
declare variable $itemDescription as xs:string external;
declare variable $user as xs:string external;
declare variable $pass as xs:string external;

declare function local:Item($name as xs:string,
                            $description as xs:string) as item()+
    {
                <item>
                    <name>
                        {$name}
                    </name>
                    <description>
                        {$description}
                    </description>
                </item>
    };
let $content :=
    if ((string-length($itemName) > 0) and
        (string-length($itemDescription) > 0) and
        (not(contains($itemName,' ')))) then
        let
            $a := xtc:storeFile(concat(http:getSessionAtt('appName'),'/items/',$itemName),
                                local:Item($itemName,$itemDescription))
        return
            <p> Item {$a/item/data(name)} created sucessfully </p>
    else
        <p> Item {$itemName} not created. Validation problems. </p>
return
    xtc:template($content)
```

Fig. 24: XQuery items creation code.

# 4 Related Approaches

This section analyzes other application servers with XQuery capabilities. The selected solutions are MarkLogic [36] and Sausalito [35]. Both are commercial products, which makes the comparison more interesting, because it shows how academic work is capable of reaching a commercial systems level. The considered aspects are the respective architecture, and how XQuery was adapted as programming language.

## 4.1 MarkLogic

MarkLogic Server is an XQuery- and XSLT-driven database server. It fuses database functionality with search and indexing features and application server logic together into a unified system [36].

It uses XML documents as its core data model, supports the ACID properties, search features, and stores both structure and unstructured information.

**4.1.1 Architecture** MarkLogic follows a three level architecture composed of a Data Layer, an Evaluation Layer and Application Services. The later handles access through a variety of technologies, such as HTTP, HTTPS, XDBC, WebDAV, REST and AJAX. The architecture is shown in Fig. 25.

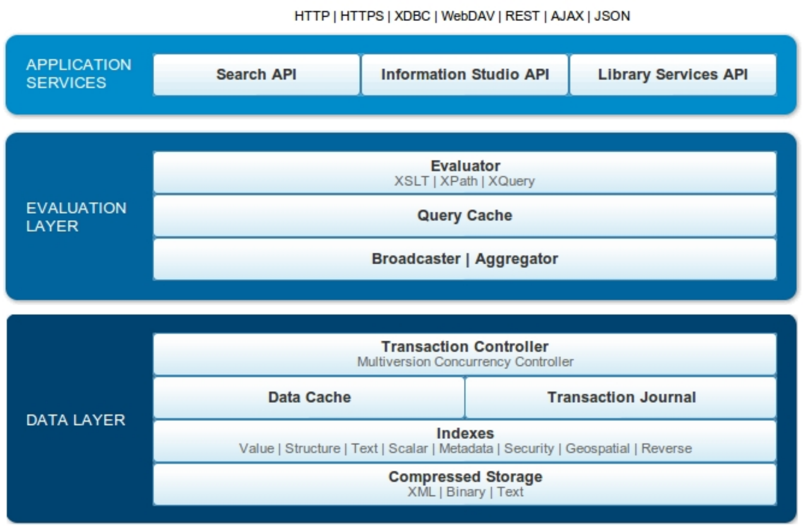


Fig. 25: MarkLogic server structure [36].

The Data layer handles data by compressing XML documents into binary fragments, provides supplementary indexes and cache mechanisms, and supporting transactions. On top of the Data layer, the Evaluation Layer evaluates all requests to documents.

Finally, the Application Services layer provides client APIs, such as Search, Information Studio (simplifies document handling) and Library Services (document management, version control mechanisms).

#### 4.1.2 Application Development

MarkLogic provides some built-in functions, aiming to fulfill the gaps for document loading and plain text search existent in XQuery [36]. It provides one store document and one search function. The store function, *xdmp:document-load('docPath')*, loads a document into the database. The search function, *search:search('key')* searches documents from the database, matching a given key.

This string search feature is certainly an extra provided by MarkLogic. They create different indexes, such as a word index and various phrase indexes, which are used depending on the search string. The search process consists of analyzing the query, deciding which index to use, selecting a first group of solution candidates, and finally filtering such candidates, confirming the result.

### 4.2 Sausalito

Sausalito is a scalable XML database and provide a set of tools that allow you to write, test, and deploy XQuery web-based applications.

#### 4.2.1 Architecture

Sausalito is an application server integrated with a database system and a web server. This section shows an overview of the whole Sausalito environment. Clients can connect to Sausalito using technologies, such as REST and HTTP. An overview of this structure is present in Fig. 26.

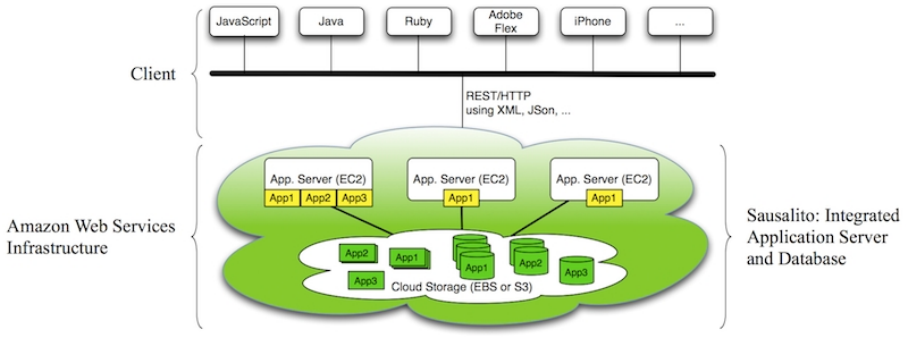

JavaScript
Java
Ruby
Adobe Flex
iPhone
...
Client
REST/HTTP using XML, JSon, ...
App. Server (EC2)
App1
App2
App3
Amazon Web Services Infrastructure
Cloud Storage (EBS or S3)
Sausalito: Integrated Application Server and Database

Fig. 26: Sausalito environment structure [35].

**4.2.2 Application Development** Like the solution proposed in this thesis, Sausalito uses a default skeleton structure for the development of applications. This structure may contain XQuery code, static files (e.g. HTML, JavaScript, or images), XML schema (to validate data), seed data (to test projects locally), configuration resources or testing information. The specific folder structure is showed in Fig. 27. In General, Sausalito provides three module types: handlers, libraries, and external modules. These modules are stored under *handlers*, *lib* and *external* folders, respectively.

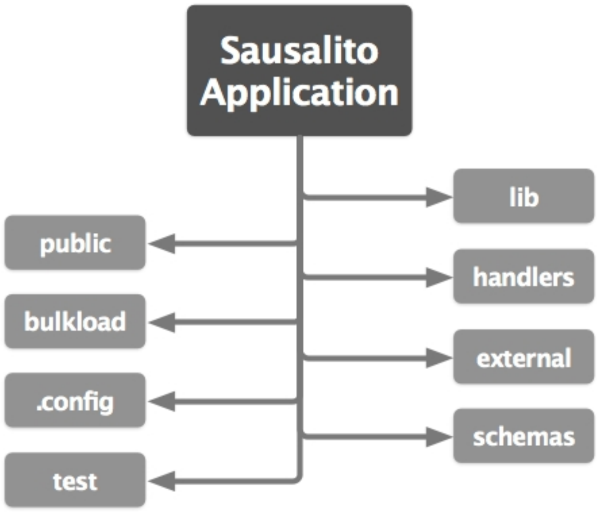


Fig. 27: Directory structure of Sausalito projects [35].

Handlers are XQuery functions that build the interface of the applications. Each function can be called by HTTP request with a path component identified by the module's name and the function's name. For example, the function named listModules in the module declared in the resource handlers/testModules.xq can be called by making one HTTP call with path */testModules/listModules*.

The lib directory contains XQuery library modules. The functions contained within these library modules can only be imported by other modules (handler modules or other library modules). That is the main difference between libraries and handlers. These libraries are not directly exposed and accessible through HTTP calls. An arbitrary number of nested directories within the lib directory is allowed.

The external directory stores third-party XQuery modules. To use these modules, an import statement is provided. More information on how to use this features can be found in [35].

Besides these folders, the *public* folder stores static files, such as images, JavaScript or HTML. The access of /public/test.jpg, for example, is possible through http://localhost/appName/test.jpg. The *test* folder contains logs and persisted data. The *bulkload* folder contains XQuery programs that can be used to load initial data into the collections declared in a project. The *.config* directory contains the sausalito.xml file. This file stores configuration details about the application. The *schemas* directory contains schemas to validate data within a project.

Sausalito also provides extra functions, with the same purpose of MarkLogic: fill the existent gaps of XQuery, concerning web application development. It provides HTTP modules, with support to session, request and responses, very useful when working with web applications.

## 4.3 Comparison

The similarities between XTC, MarkLogic and Sausalito are evident. Therefore, comparing these commercial solutions with a prototyped application server becomes interesting. It is motivating to notice how much an academic system such as XTC can achieve.

MarkLogic provides different access protocols, along with a consistent and well-organized application server structure, expanded features implemented as predefined functions and deployment services. Sausalito provides cloud support, RESTfull services, plugins for Eclipse development, namespaces support and a predefined structure for application development.

A better comparison between these approaches is shown in table 1.

| **Feature** | **XTC** | **MarkLogic** | **Sausalito** |
|---|---|---|---|
| Application deployment services | OK | OK | OK |
| Application access methods | OK | OK | OK |
| Application development structure | OK | - | OK |
| Detailed application server architecture | OK | OK | - |
| Extra predefined functions | OK | OK | OK |
| Cloud support | - | OK | OK |
| RESTfull services support | - | - | OK |
| Development plugins | - | - | OK |
| Binary files under database storage | OK | - | OK |

Table 1: Comparison between approaches

# 5 Conclusions

This thesis showed the development of a whole web application using the XQuery programming language. This technology has shown to be suited for this development and to have significant advantages over common application development paradigms (J2EE, .Net, etc), avoiding the impedance mismatch between application layers and ensuring simpler code.

XTC has shown to be a good application server. The functionalities developed within this work achieved the thesis objectives: the server provides HTTP access to resources managed by the database; provides authentication and authorization methods; and deals with different MIME types properly. The Java part of the server, consisting of the servlets already explained in section 2.3, is minimal and very functional. It contains only code that would take longer to be developed within XQuery and is much simpler within Java servlets environment.

The application development itself also showed to be flexible and simple. The standardized directory structure helped significantly to increase development efficiency.

The deployment mechanism also showed to be necessary. Even though XQuery pages are accessed directly from the application structure, thus making new changes to these files automatically accessible, new resourses that need to be stored in XTC, i.e., new images, are only available after re-deployment.

Summarizing, this thesis proposed a first sketch of web application development using XQuery successfully. In the future, however, more experience with this kind of development is needed.

## 5.1 Future Work

After developing this application using XQuery, and all necessary server-side development, some problems emerged as challenges for the future.

**Development Plugins**

Plugins for development tools (like Eclipse), are a good idea. Sausalito provides a tool called XQuery Development Toolkit (XQDT) [40]. XQDT supports XQuery inside Eclipse. It provides features such as XQuery 1.1, XQuery scripting, code completion and code templates, launching and debugging support for Sausalito and Zorba, as-you-type validation and semantic checking.

**Data Validation**

Concerning data consistency, the creation of relational database concepts like *keys* is often necessary. For example, when a new item is created within the presented application, it has an identification number. When the shopping cart containing this item is bought, the item number is used to reference it in the XTC database and to create a buying order. The relation between the item and the buying order is that an item cannot be bought when it does not exist within the database. In a relational database, *referential key constraints* address this kind of problems. Constraints make possible to restrict

the domain of data values. We could address a specific integer column of a table to accept only values between 1 and 99 using a constraint. The same concept is valid for referential constraints. The values of a column can be restricted to another table column using a primary key. In the item example, the item identification number would be the *primary key* of the table. Whenever a buying order is placed, the item to be bought must exist within the item table, otherwise the buying order would not be placed.
There are techniques for data validation within the XML world, such as XML Schema or Document Type Definitions (DTD). A schema validation step can check if a XML file is well-formed, validate data types and data structures.

**Application Monitoring**

Another good idea is a kind of application monitor, as the one present in MarkLogic. This application monitors all applications deployed under the server, and shows statistics and management options. More details are shown in [36].

**Deployment Improvements**

The XTC server automatically deploys all applications under its *apps* folder. However, especially in the development phase, simple and fast mechanisms to (re-)deploy an application partially or as a whole are desirable. That is why a "hot deployment" mechanism would be a good alternative for the future, along with other optimizations and improvements concerning the deployment mechanism.

**Scripting Facilities**

XQuery provides an extension called XQuery Scripting Extension 1.0 [41]. Most programmers, nowadays, are used to imperative programming (programming in statements that change the program state). Thus, the functional programing is not very common in practice. With the XQuery Scripting Extension, expressions can be evaluated in a specific order, with later expressions seeing the effects of the expressions that came before them. These support of the XQuery Scripting Extension in XTC might be a good idea.

**Plain Text Search**

XQuery expresses rich and powerful queries over structured data, but it expresses very poor queries over plain text data [42]. A text search extension might be a good idea. Some approaches already exist, like the one proposed by MarkLogic [36] and other by TeXQuery [42].

## References

1. W3C XQuery Specification, http://www.w3.org/TR/xquery/
2. Core J2EE Patterns: Best Practices and Design Strategies Deepak Alur, John Crupi and Dan Malks Publisher: Prentice Hall / Sun Microsystems Press ISBN:0130648841; 1st edition (June 26, 2001)
3. Microsoft Deployment Patterns http://msdn.microsoft.com/en-us/library/ms998478.aspx
4. Java Language Specification, 3rd Edition.z/OS V1R7.0-V1R11.0 TSO/E Guide to Server-Requester Programming Interface, SA22-7785
5. Leonardo Andrade Ribeiro, A Framework for XML Similarity Joins, Ph.D. Thesis, University of Kaiserslautern, Verlag Dr. Hut, Mnchen, October 2010
6. Christian Mathis, Storing, Indexing, and Querying XML Documents in Native Database Management Systems, Ph.D. Thesis, University of Kaiserslautern, Verlag Dr. Hut, Mnchen July 2009
7. Theo Hrder and Andreas Reuter. Concepts for Implementing a Centralized Database Management System. In Symposium on Application Systems Development, pages 2860, 1983.
8. "The xtc project: Native xml data management," http://wwwlgis.informatik.uni-kl.de/cms/dbis/projects/xtc/.
9. RFC 822, http://www.ietf.org/rfc/rfc822.txt
10. RFC 1341, http://www.ietf.org/rfc/rfc1341.txt
11. RFC 2045, http://www.ietf.org/rfc/rfc2045.txt
12. RFC 2616, http://www.ietf.org/rfc/rfc2616.txt
13. RFC 3236, http://www.ietf.org/rfc/rfc3236.txt
14. A. Tanenbaum, AS ; Computer networks; Prentice Hall PTR; 2003
15. Jason Hunter, William Crawford; Java servlet programming; O'Reilly Media, Inc., 2001
16. HTML 4.01 Specification, http://www.w3.org/TR/html4/
17. Vora P.; Web application design patterns; Morgan Kaufmann, 2009.
18. Loudon, K.; Developing Large Web Applications: Producing Code That Can Grow and Thrive; O'Reilly Media, Inc., 2010
19. Taylor, A.; J2EE and beyond; Prentice Hall PTR, 2002
20. G. Copeland, D.Maier; Making Smalltalk a database system. SIGMOD 1984, Boston, USA.
21. Ralf L&#228;mmel and Erik Meijer. 2006. Revealing the X/O impedance mismatch: changing lead into gold. In Proceedings of the 2006 international conference on Datatype-generic programming (SSDGP'06), Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring (Eds.). Springer-Verlag, Berlin, Heidelberg, 285-367.
22. Brittain, J.; F. Darwin, I; Tomcat: the definitive guide; O'Reilly Media, Inc.; 2007
23. Eckstein, R.; Java Enterprise best practices; O'Reilly Media, Inc.; 2002
24. RFC 1321, http://www.ietf.org/rfc/rfc1321.txt
25. W. Royce, Managing the Development of Large Software Systems; Proc. Westcon, IEEE CS Press, 1970, pp. 328-339.
26. Casteleyn, S; Daniel, F; Dolog, P; Matera, M; Engineering Web Applications; Springer, 2009
27. B Boehm, A spiral model of software development and enhancement, ACM SIGSOFT Software Engineering Notes, v.11 n.4, p.14-24, August 1986
28. Martin, J; Rapid application development; Macmillan Pub. Co., 1991

29. Goel, R; E-Commerce; New Age International, 2007
30. Web-Based Application Development: Look for servlets information
31. javax.servlet, http://download.oracle.com/javaee/1.3/api/javax/servlet/package-summary.html
32. javax.servlet.http, http://download.oracle.com/javaee/1.3/api/javax/servlet/http/package-summary.html
33. Servlet interface, http://download.oracle.com/javaee/1.3/api/javax/servlet/Servlet.html
34. HttpServlet, http://download.oracle.com/javaee/1.3/api/javax/servlet/http/HttpServlet.html
35. Sausalito Project Web Page, http://www.28msec.com/home/index
36. Hunter, J; Inside MarkLogic Server; Revision 7; MarkLogic Co.
37. Application Builder Developer's Guide; MarkLogic Co., 2009.
38. Bales, D; JDBC pocket reference; O'Reilly Media, Inc., 2003.
39. M. Kaufmann and D. Kossmann; Developing an Enterprise Web Application in XQuery. http://www.28msec.com/tech_reading.html, 2008.
40. XQDT, http://www.xqdt.org/.
41. XQuery Scripting Extension 1.0, http://www.w3.org/TR/xquery-sx-10/
42. S. Amer-Yahia et al; TeXQuery: A Full-Text Search Extension to XQuery; In WWW 2004.