Dbox Online Resource Management System: an application based on HTTP and RESTful architecture

Computer Network Fundamentals
Spring 2011
Version: 0.1

Jong H. Lim Andreas Terzis
Department of Computer Science
Johns Hopkins University
Baltimore, MD
Email: {ljh,terzis}@cs.jhu.edu

February 8, 2011

1 Introduction

The Web has become an indispensable element of our everyday lives, as more and more people use it for sharing information. The reason behind the Web's success lies on its simplicity. Locating, accessing, and retrieving information become easy through the use of URLs. Furthermore, application developers can use HTTP, the Web's popular communication protocol, to develop their applications without worrying about low-level communications details (e.g., packet retransmission and delimiting messages). Moreover, existing HTTP protocol extensions support additional features such as security and state management.

The goal of this project is to implement *DBox*, a Dropbox-like (see `http://www.dropbox.com/`) online file management system. The implementation will use the protocols and interfaces upon which existing Web services rely heavily—HTTP and XML. Specifically, the server side is an application running on a PC that handles HTTP requests from multiple clients and generates appropriate HTTP responses (likely to contain XML documents). The client side is an application running on an Android device (Motorola Droid phone). Upon user request, the application lists the user's resources (i.e., files and directories), downloads them from, uploads them to, or deletes them from the remote server(s). All operations are performed through XML messages exchanged over HTTP.

HTTP and XML by themselves are not enough to design Web services because they are only a communication protocol and data representation format, respectively. For this reason, multiple methods have been proposed for implementing Web services, including SOAP and CORBA. Among them, we choose REST (REpresentational State Transfer) as the way of building this project's Web services.

1.1 Educational goals

The educational goals of this project are:

- Teach students the design of REST-based Web services.

- Familiarize students with the HTTP protocol, XML, and web programming.
- Teach students how to develop networked applications for the Android platform.

2 Architecture

The goal of this section is to introduce the Web design philosophy underlying this project —REST (REpresentational State Transfer) [15]— and present the Web services architecture following the REST paradigm.

2.1 RESTful Design

One of the most popular Web service architectures available today is SOAP+WSDL, an example of RPC-style (Remote Procedure Call) architectures. As the RPC name suggests, a client in this architecture calls a procedure defined by the server side to perform the operation that it desires. For instance, consider a client that wants to read the contents of a file. Then, the client describes the name of a function to be called within an XML message and sends this message through an HTTP POST request. While this approach achieves the client's goal, it does not use the HTTP POST request in the way that it was originally designed. (According to the original HTTP design, the GET request should be used to 'read' the contents of a resource). Moreover, the client uses the same URL (e.g., `http://api.google.com/search/beta2`) for every API call, ignoring most features of the HTTP protocol and good Web design principles.

On the other hand, architectures following the REST paradigm are similar to Object Oriented (OO) design. In OO designs, a programmer defines a class and class methods describe what class instances can do. Similarly, a RESTful design, which is resource-oriented, defines resources and describes the actions that these resources perform when standard HTTP requests (e.g., GET, POST, PUT, DELETE) are called against them. Furthermore, distinct URIs are used to name and locate all the resources defined in a RESTful design.

Irrespective of whether a Web Service follows a RESTful or RPC-style architecture, it is built using two components: resources and URIs (Uniform Resource Indicator). A resource is anything that is important enough to be referenced as a separate entity. Usually, a resource is something that can be stored on a computer and represented as a stream of bits: a document or a row in a database, but software components as large as a search engine are also considered as resources. Then, a URI is the name and address of a resource.

The Resource-Oriented Architecture (ROA), described in [10], presents one set of best practices for designing given resources and URIs into RESTful Web Services. In short, it suggests that a RESTful design include the following components:

- **Addressability:** every interesting piece of information the server can provide should be exposed as a resource, and given its own URI. RPC-style architectures also define resources and URI, but it is often the case that one URI represents multiple of resources.
- **Uniform Interface:** resources should expose a subset of the standard, uniform HTTP interfaces to be operated on.
- **Representation:** any information about the current state of a resource. For example, a file could be represented as a set of metadata, such as file name, size, and modification date. At the same time, the contents of the file are also a representation of the file.

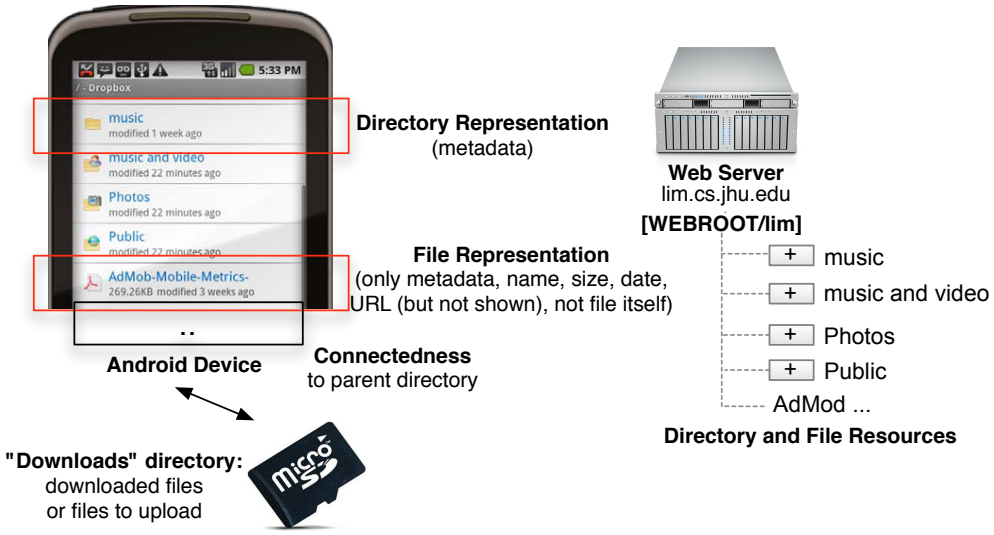

Figure 1: Dbox architecture for file management. The GUI on the Android device may vary, but the screen shows how the REST concepts are embedded in the project's implementation (Sec.3 describes the uniform interface).

- **Connectedness:** the quality of having links to connect resources and for clients to change application states. As resources become better-connected, the relationships between them becomes more obvious. This also gives applications (clients) a handle to move across resources.

For more information about RESTful designs, the interested reader can refer to: [4, 5, 6, 10, 15].

2.2 Web Service Architecture of the DBox Project

The design of the Dbox project follows the ROA principles described in Sec.2.1.

Resource: There are three types of resources: files, directories, and user accounts. We use MIME types to describe the different supported file types [14].

Addressability: Each resource has a unique name in the form of a URL.

Uniform Interfaces: File resources expose the HTTP GET (download), PUT (upload), and DELETE (delete) methods. Directory resources expose the HTTP GET (retrieve the entries of a directory), PUT (create a directory), and DELETE (delete a directory) methods. Finally, user accounts support the HTTP PUT method for changing a user's password.

Representation: File and directory resources have two representations: metadata and actual resource (that provides the resource's contents). The resources on the directory listing displayed on the screen shown in Figure 1 have not been downloaded on the device. Instead, the client has downloaded and displays only the metadata about these resources. Only after the user requests (clicks on) a file to be downloaded, the actual contents are transferred to the device. We note that the contents of a directory are the metadata of the files and sub-directories contained in that directory. The difference between requesting the contents of a file and a directory is that the downloaded content of a file is stored in a *pre-defined* directory on the device's SD card (See Sec.3.5), whereas the contents of a directory are only shown on the screen.

The metadata associated with a file resource are its name, size in bytes, modification date, and resource type, which is 'file'. Likewise, the metadata associated with a directory resource are its name, the number of entries it contains, modification date, and resource type, which is 'directory'. Finally, the metadata of a user account resource are its user name, ID, password, and resource type, which is 'user account'. We emphasize that the password is never sent to the client, but rather stored at the server and used to authenticate user requests.

Connectedness: Each resource has a URL tag when represented in XML format (See Sec.3.1.2). Furthermore, as Figure 1 suggests, directories include a link to the parent directory (equivalent to the '..' convention in Unix-like environments. See also Sec.3.1.2).

3 Implementation

This section describes the tasks that the DBox server and client must perform.

At a high level, the server maintains resources in the form of a tree of directories. For example, when a client accesses a resource through the URL `http://lim.cs.jhu.edu/lim/music`, it refers to a resource (either file or directory) located at lim/music under the root directory of the DBox server (see Figure 1). In other words, there is a one-to-one correspondence between URLs and the directory structure of the web server. The client responds to user requests, to list and update directory resources, upload and download files, and possibly delete file and directory resources.

3.1 Web Server

The Web server is a process running on a PC that acts as a file repository. The server processes the clients' HTTP requests and replies with appropriate HTTP responses. HTTP responses can include specific response codes that the server generates while processing a request. For any client-side errors that are not described in the following sections, the server returns *HTTP 400 "Bad Request"*. On the other hand, for the server side errors (i.e., when the server catches any error while processing a valid user request), the server returns to the client an *HTTP 500 "Internal Server Error"* response and possibly the reason for the failure in the body of the response. We list the tasks the web server performs in what follows.

3.1.1 Authentication

The server maintains a cache of valid users and their passwords. Upon startup, the server populates this cache by reading the contents of the `.passwd` file located in the server's root directory. The server checks the validity of the user credentials that a client sends in an authentication header (see Sec.3.2.1) by comparing them with the credentials stored for that user. Based on the results of this comparison, the server responds with either an *HTTP 200 "OK"* or an *HTTP 401 "Unauthorized"* response. Note that all client requests for any resource must include an authentication header; otherwise, the server must generate a 401 response. Finally, the server must never serve the contents of the `.passwd` file even when it receives a properly authenticated request. Instead, it returns *HTTP 403 "Forbidden"* if a request is made to the file. If your library of choice (suggested in Sec. 3.4) does not provide this implementation, it should be easy to build. Basically, the authentication mechanism uses the MD5 algorithm to generate a secret message to be exchanged. A random number generator can be used to generate a nonce for the MD5 algorithm. Both MD5 and random number generators are provided in most of programming languages. For more details about how the Digest algorithm works, please refer to [13].

3.1.2 Directory Listing

When the server receives an HTTP GET request for a directory, it first checks whether the directory exists under the server's root directory. If it does, the server builds an XML document that includes all the entries present in the requested directory. The format of the XML document uses the following elements:

- `ResourceList`: this is the root element enclosing all resources.
- `Resource`: an element representing each entry in the directory. It has attribute `category`, indicating whether a resource is a "file" or "directory".
- `ResourceName`: the name of a resource (including file extension if present).
- `ResourceSize | ResourceNumItems`: for a file, use `ResourceSize` to represent the size of the file in bytes. On the other hand, for a directory, use `ResourceNumItems` to represent the number of entries contained in the directory.
- `ResourceURL`: a fully-qualified URL for a resource.
- `ResourceDate`: the date of the last file or directory modification defined as: ⟨year⟩, ⟨month⟩,⟨day⟩, ⟨hour⟩,⟨min⟩, ⟨sec⟩. For a file, this is the last modification date, while for a directory this reflects the date of the last modified file in the directory.
- `ResourceType`: MIME type of a resource defined in [14].

For example, using this XML format, a PDF file with the name `test.pdf` would be represented as:

```
<ResourceList>
 <Resource category="file">
  <ResourceName> test.pdf</ResourceName>
  <ResourceSize> 128 </ResourceSize>
  <ResourceURL> http://lim.cs.jhu.edu/lim/test.pdf </ResourceURL>
  <ResourceDate>
   <year>2010</year>
   <month>10</month>
   <day>13</day>
   <hour>06</hour>
   <min>24</min>
   <sec>10</sec>
  </ResourceDate>
  <ResourceType> application/pdf </ResourceType>
 </Resource>
</ResourceList>
```

If the directory contains multiple resources, then the `ResourceList` element contains multiple `Resource` elements. If the directory name does not exist, the server returns a *HTTP 404 "Not Found"* response.

Furthermore, all directory listings should include a resource entry corresponding to the parent directory for connectedness. The `ResourceName` for the parent directory is to be "**..**", following Unix convention. The only exception to this rule is when listing the user's home directory. In this case, the parent is *WEBROOT*, and any access to it must elicit a *HTTP 403 "Forbidden"* response (for *WEBROOT* see Figure 2).

3.1.3 Downloading a File

When a user selects the name of a file resource, the client application issues an HTTP GET request to download the corresponding resource (see Sec.3.2.3). Once it receives the request, the server constructs an XML document that contains the requested file's metadata as well as its contents. The format of the server's XML response is as follows:

- `ResourceDownload:` this is the top-level element, enclosing the following elements.
- `Resource:` an element representing the file that is downloaded.
- `ResourceName:` the name of the requested file.
- `ResourceSize:` the size of the file in bytes.
- `ResourceType:` MIME-type of the requested file defined in [14].
- `ResourceEncoding:` encoding scheme used for the ResourceContent. Should be "Base64" [12] for binary files, and a particular text encoding scheme for text files.
- `ResourceContent:` the file's contents.

Following the previous example with a PDF file, named **test.pdf**, will be delivered as in the following XML sample:

```
<ResourceDownload>
 <Resource category="file">
  <ResourceName> test.pdf </ResourceName>
  <ResourceSize> 128 </ResourceSize>
  <ResourceType> application/pdf </ResourceType>
  <ResourceEncoding>Base64</ResourceEncoding>
  <ResourceContent> ... [content of pdf file encoded using the Base64 encoding]
   ...
  </ResourceContent>
 </Resource>
</ResourceDownload>
```

If the server receives a malformed request or a request for non-existing file, it responds with a *HTTP 404 "Not Found"* response.

3.1.4 Uploading Resources

Once the server receives the HTTP PUT request, defined in Sec.3.2.5, it parses the XML document and creates a new file or a new directory. Upon successful creation, it responds to the client with a *HTTP 201 "Created"* response and a message embedded in the body of the response. For example, after a file has been successfully uploaded, the server's HTTP response contains an 'Uploading [filename] successful' message. If the client wants to create a directory with a name that already exists, the server generates a *HTTP 400 "Bad Request"* response. If the client wants to create a file that already exists, the new file replaces the existing one.

3.1.5 Deleting Resources

When a client sends an HTTP DELETE request to remove the file corresponding to the URL in the HTTP request, the server checks whether the file exists. If so, it deletes the file and responds with an *HTTP 200 "OK"*; otherwise, it replies with a *HTTP 404 "Not Found"* response and possibly a more descriptive message in the response's body. Deleting a directory follows the same procedure but deletes all the sub-directories recursively.

3.2 Web Client

The client is an application running on an Android device that initiates operations towards a Dbox server. The client must include the following functionality: 1) a GUI for interacting with the user, 2) a protocol engine for sending HTTP requests to the server and receiving HTTP responses from the server, 3) an XML engine for generating valid XML requests and parsing the server's responses.

The following subsections describe the client's operations in more detail.

3.2.1 Login and Authentication

The first thing that users need to do after starting the client is to provide their username and password through a login screen. The client uses this username to generate an HTTP GET request for the user's home directory. In addition, the user also provides the server name and its port number. For example, if the username is `lim` and server name and port number is lim.cs.jhu.edu and 8080, respectively, then the client requests the URL `http://lim.cs.jhu.edu:8080/lim/`.

If the server verifies the user's credentials, it will generate the response described in Sec.3.1.2. If the response is *HTTP 200 "OK"*, the client parses the XML document and displays the list on the screen; otherwise, the server will generate an *HTTP 401 "Unauthorized"* response. The client should display an error message in the later case and request the user to re-enter her login information.

For actual authentication, the client uses the HTTP 'Digest' authentication described in [13]. It is designed to replace HTTP's basic authentication and is stronger than many popular schemes proposed for other Internet services, such as CRAM-MD5 (used for LDAP, POP, and IMAP). In short, the scheme never sends the password across the network in cleartext format. Instead, it sends a fingerprint or digest of the password, which is a one-way hash of the password. Most HTTP client libraries mentioned in Sec.3.4 provide this functionality. Even after logging in, the client should include this authentication header to all HTTP requests.

3.2.2 Directory Listing

By clicking a directory entry on the screen, the client initiates an HTTP GET request for the URL corresponding to the directory. If the server responds positively (Sec.3.1.2), the client shows the entries contained in the requested directory. Other than displaying the user's home directory, the listing should include an entry linking the directory to its parent directory. In other words, even though the directory is empty, there should be at least one entry shown on the screen, and the name of the entry should be "**..**" following the Unix convention (see the last "**..**" entry in Figure 1).

In addition, the UI should include a *Home* button that allows the user to return to her home directory. The location of this button is left to the implementer (e.g., it can be one of the options available when the user clicks the Menu key [9]).

3.2.3 Downloading Resources

The client initiates the process of downloading the contents of a file, after the user selects the file, by generating an HTTP GET request for the URL corresponding to the file. The client parses the server's response (see Sec.3.1.3) and stores the file to the pre-defined directory on the device's SD card (See Sec.3.5). If the file already exists in the device's SD card, simply replace it.

3.2.4 Refreshing the Screen

The user can initiate a refresh of the current directory at any point in time by clicking on the 'Menu' key and selecting the refresh operation. Choosing to refresh generates the same HTTP request as the one described in Sec.3.2.1. The client then compares the server's response with the current contents of the directory and updates its contents accordingly (i.e., it adds new entries, removes non-existing entries, and updates the metadata of existing entries).

3.2.5 Uploading Resources

The client can also upload new user files to the server. Specifically, consider that the user has currently navigated into the 'music' directory. Pressing the device's 'Menu' key ([9]), and selecting the 'Upload' option brings up a menu that allows the user to select a single file from a *pre-defined* directory (see Sec.3.5) on the device's SD card. Once the user selects the file from the directory, the client attempts to send the selected file to the corresponding 'music' server directory (i.e., to the 'music' directory's URL) using the HTTP PUT method.

For the server to be able to store the file correctly, the client includes relevant information into the XML document in addition to the file's actual contents:

- `ResourceUpload`: this is the top-level element enclosing all other elements.
- `Resource`: an element representing the file that is being uploaded. Its `category` attribute be set to "file".
- `ResourceName`: the file's name.
- `ResourceLocation`: a directory string used to identify the location to store the uploaded file. The string starts with the user's ID, followed by a path to the target directory.
- `ResourceSize`: the size of the uploaded file in bytes.
- `ResourceType`: MIME-type of the uploaded resource defined in [14].
- `ResourceEncoding`: encoding scheme used for the ResourceContent. Should be "Base64" [12] for binary files, and particular text encoding schemes for text files.
- `ResourceContent`: actual contents of the uploaded resource.

For example, if the user wants to upload a PDF file named **test2.pdf**:

```
<ResourceUpload>
 <Resource category="file">
```

```
  <ResourceName> test2.pdf</ResourceName>
  <ResourceLocation> lim/music </ResourceLocation>
  <ResourceSize> 128 </ResourceSize>
  <ResourceDate>
   <year>2010</year>
   <month>10</month>
   <day>13</day>
   <hour>06</hour>
   <min>24</min>
   <sec>10</sec>
  </ResourceDate>
  <ResourceType> application/pdf</ResourceType>
  <ResourceEncoding>Base64</ResourceEncoding>
  <ResourceContent> ... [contents of pdf file]... </ResourceContent>
 </Resource>
</ResourceUpload>
```

Creating a directory is similar. Specifically, the user initiates the creation of a new directory by clicking the device's 'Menu' key and selecting the 'Upload Directory' option. The resulting XML document differs in two ways. First, the `category` attribute has value "directory". Second, the XML document includes only the `ResourceName` and `ResourceLocation` entries.

Thus, creating a new directory called 'mymusic' under the 'music' directory would look like this:

```
<ResourceUpload>
 <Resource category="directory">
  <ResourceName>mymusic </ResourceName>
  <ResourceLocation> lim/music</ResourceLocation>
 </Resource>
</ResourceUpload>
```

Depending on the server's response (see Sec.3.1.4), the client displays the message from the server indicating the success or failure of the operation (e.g., using the 'Toast' notification provided by Android [2]). If the file was successfully uploaded, it should be visible the next time the user refreshes the screen.

3.2.6 Deleting Resources

The user can delete files and directories by selecting the resource from the current directory and selecting the 'Delete' option from the 'Menu' options. The client then sends an HTTP DELETE request to delete the corresponding resource. After successfully deleting the resource, the server generates an *HTTP 200 "OK"* response. Refreshing the screen removes the deleted resource.

3.2.7 Logging Out

The user can log out by selecting 'Logout' from the 'Menu' key [9]. Doing so, returns the user to the login screen from which the user can re-login by providing her username and password.

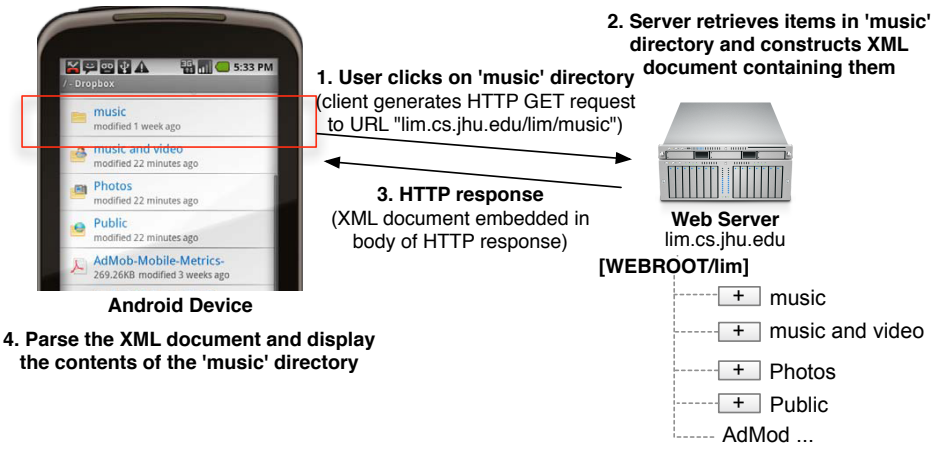

Figure 2: An example of the sequence of steps for retrieving a directory list. The client needs to handle (1) user interaction on the device, (2) generate HTTP request, and (3) parse the XML document and display its contents to the user. The server needs to be able to receive requests from multiple clients and generate appropriate responses. The same basic steps are used in all other client-server interactions. *WEBROOT* is the directory in which the server is running.

3.3 Server and Client Interaction Example

Next, we clarify the interactions between the client and server described in the previous sections through an example in which the client retrieves the contents of a directory from the server. We assume in this example that the client was just served with a list of directories and files after the user signed in (using username `lim`).

1. The user in Figure 2 clicks on the 'music' entry in the current directory to view the contents of that sub-directory. The client's GUI event handler responds to the click and sends an HTTP GET request for URL `http://lim.cs.jhu.edu/lim/music/`

2. The server parses the URL in the HTTP GET request to determine the corresponding resource (in this case, to the 'music' directory). Based on the request type (i.e., GET), the server reads the content of the `WEBROOT/lim/music` directory, and constructs an XML document containing the directory's entries (including its parent directory).

3. The server sends an HTTP response with the constructed XML document in its body.

4. The client receives the response, parses the XML document, and displays the list of entries of the 'music' directory.

While this example illustrates the sequence of actions when retrieving the contents of a directory, all other user-level tasks follow the same four steps: (1) User initiates an operation through the device's UI. (2) Client sends an HTTP request to the appropriate URL. (3) Server receives the request, parses the URL, constructs a XML document and sends an HTTP response. (4) Client parses the XML document and refreshes the UI accordingly.

3.4 Programming Languages, HTTP libraries, XML Parser

We mention some widely-used HTTP and XML related libraries that can be used to implement the DBox client and server. The list is not exclusive and students can use your libraries of choice. In any case, the project's documentation should explicitly mention any third-party code that was used.

Programming Language: Since the server is a Linux application, one can use the programming language of their choice. On the other hand, since the Dbox client needs to run on Android devices (Motorola Droid phone in this case), it should be developed in Java.

HTTP server libraries: The HTTP server library handles the clients' HTTP requests and generates HTTP responses. Python provides the `BaseHTTPServer` or the more advanced `Tornado` [11] module. In Java, although not a standard library, Apache has developed the `HttpComponents` library [3]. Likewise, C++ has libraries such as `pion-network-library` and `GNU libmicrohttpd`.

HTTP client libraries: Java has two well-known client libraries: `HttpClient` and `HttpURLConnection`, provided by Apache and the Java standard library, respectively. Both provide the HTTP methods and features that the Dbox client needs. The `Restlet` library is also widely used, but it does not provide Digest authentication.

XML Parser: Java provides the `javax.xml, Xerces,` and `XMLPull` (in the `javax.xml.stream package`) libraries. Python2.5 has seven different XML interfaces. Among them, the most well-known are `ElementTree, xml.sax`. [7] and `pyXML` provide well-written tutorials for these modules. C# provides `System.Xml.XmlReader`, Ruby has `REXML`, C has `Expat`, and C++ the `Xerces-C++` parser.

3.5 Download and Upload Directories

Both download and upload operations interact with the device's SD card. Specifically, let's assume that a directory named 'Downloads' exists on the device's SD card (default on the Motorola Droid). In this case, all resources downloaded from the server will be stored in the 'Downloads' directory. When uploading files, the client retrieves files from the 'Uploads' directory and transfer them to the server. To simplify its implementation, the client only uploads and downloads resources in these two *pre-defined* directories.

3.6 Conventions

The names used for directory and URL are all case-sensitive. When constructing a URL, the server's domain name is always followed by a username. For instance, the default server name `lim.cs.jhu.edu` (or 128.220.71.91) is appended by user name (`lim`) resulting in `http://lim.cs.jhu.edu/lim/` for accessing the home directory of the `lim` user.

3.7 Developing Android Application

Students need to learn on their own to develop Android applications by following the official tutorials and APIs described in [1]. This should be easy since developing Android applications is very similar to developing other Java applications. Moreover, multiple tutorials and examples are available on the web, including [8].

4 Deliverables

Student teams must submit all server source code, including a `Makefile` if necessary. The submission must also include a `README.txt` file that includes: **(1)** the names of the team's members (including whether team is registered for the 344 or the 444 version of the class), **(2)** instructions on how to run the submitted code (and to install if a particular installation process is needed) and, **(3)** if the work is not complete, a description of the missing parts. In addition, each team must also need to submit **(4)** the whole Android project for the DBox client so that TA can simply import it via Eclipse. Do not include any binaries or files that were used to test the implementation (e.g., downloaded files). Each team should create a single compressed archive containing all files. The name of the file should be: `groupNumber_CS344Spring11.tar.gz (.zip)` or `groupNumber_CS444Spring11.tar.gz (.zip)`. Each team must submit this file through JHU blackboard (NO EMAILS!).

5 Grading Policy

We will *strictly* follow the 'Late Policy on Homework and Projects' described on the course's website. Grading is mostly based on whether the team has correctly implemented the components described in Sec.3. Teams should make sure that they follow best software engineering practices (e.g., thoroughly documenting source code).

References

[1] Android. Android Developers. Available from: `http://developer.android.com/index.html`.

[2] Android. Creating Toast Notifications. Available from: `http://developer.android.com/guide/topics/ui/notifiers/toasts.html`.

[3] Apache. HttpComponents Core. Available from: `http://hc.apache.org`.

[4] Duncan Cragg. The REST Dialogues. Available from: `http://duncan-cragg.org/blog/post/getting-data-rest-dialogues/`.

[5] Joe Gregorio. REST. Available from: `http://www.xml.com/pub/au/225`.

[6] Joe Gregorio. REST and WS. Available from: `http://bitworking.org/news/125/REST-and-WS`.

[7] Doug Hellmann. Parsing XML Documents with ElementTree. Available from: `http://blog.doughellmann.com/2010/03/pymotw-parsing-xml-documents-with.html`.

[8] Ying Huang. Begin Android Journey in Hours. Available from: `www.cs.uiuc.edu/class/fa09/cs425/mps/tutorial.pdf`.

[9] Motorola Corporation. Droid user guide. Avalaible from: `http://www.motorola.com/staticfiles/Support/US-EN/Mobile%20Phones/DROID-by-Motorola/US-EN/Documents/Static-Files/DROID_UG_Verizon_68000202474a.pdf`.

[10] Leonard Richardson and Sam Ruby. *RESTful Web Services: Web services for the real world*. O'Reilly, May 2007.

[11] Tornado. Tornado Web Server. Available from: `http://www.tornadoweb.org/`.

[12] Wikipedia. Base64. Available from: `http://en.wikipedia.org/wiki/Base_64`.

[13] Wikipedia. Digest Access Authentication. Available from: `http://en.wikipedia.org/wiki/Digest_access_authentication`.

[14] Wikipedia. Internet Media Type. Available from: `http://en.wikipedia.org/wiki/Internet_media_type`.

[15] Wikipedia. Representational state transfer. Available from `http://en.wikipedia.org/wiki/Representational_State_Transfer`.