SPL 121
Assignment 4

Due: 19/1/2012, 23:59 PM

TAs in charge: Achiya Elyasaf and Majeed Kassis

5/1/2012

1 Assignment's Objectives

The purpose of this assignment is to gain experience with communication protocols, socket programming in Java and C++. For this assignment, there is no focus on concurrency issues – you can implement the server parts using the simple thread per connection model. Careful design before the actual implementation of the task is essential to make the programming process less painful and more efficient. We expect you to follow the Test-Driven Development method, which means you should be ready to demonstrate to the grader your unit tests for main classes you develop for this assignment.

2 General Description

In season 2 episode 8 of Fox' famous series—*That 70s Show*—Hyde (Danny Masterson) decides to look for a job. He finds Leo's (Tommy Chong) advertisement for a job in the *Photo Hut* and applies for it. As it turns out, Leo is a lovely hippie, which means that the employees at his Hut are not efficient in any way (Figure 1).

In this assignment, we simulate a decent Photo Hut with one *work manager*, several *employees*, and many *clients* with different demands. As soon as a client enters the store, he sends the work manager a picture. The work manager returns to the client the job id for that picture. The client then sends a single job request (an image processing task) for that job. The work manager tries to find an available employee to do that job (using OpenCV). Once the employee is done, he notifies the manager. At any time, the client can ask the manager for the job status. The manager responds with the appropriate message. Once the job is done, the client can get the new photo from the manager.

A typical interaction between the actors of the simulation will be:

- The client uploads an image file to the store - he obtains in response an ID to refer to his picture.

Figure 1: That '70s Show - Leo's Photo Hut (Season 2 Episode 8 - Sleepover)

- The client sends a request to process the image - for example, produce a black and white version of his picture.
- The manager submits the job to produce the requested version of the image to a worker.
- The client asks the manager for the status of his requested job and obtains an answer such as "pending", "in progress" or "completed"
- Once the status is "completed", the client requests the new picture (the black and white version) and downloads it.

All messages in this assignments will be written in XML and exchanged over the HTTP protocol. Our work manager will be implemented as a Web server: it will listen to port 80 (by default), get HTTP requests and will respond to them. In Section 3, we explain the communication protocol to be developed on top of HTTP and in Section 4 we explain how to use XML.

3 Communication Protocol

3.1 HTTP Requests and Responses

HTTP is a client-server oriented protocol. Like most similar protocols, it is based on a simple paradigm: the client opens a connection to the server and makes a request; the server answers with a response, which usually contains a description of the resource which the client asked. The format of both the request and the response is similar:

- An initial line (specifying the type of request/response),
- Zero or more header lines,
- A blank line,
- an optional body (*e.g.*, the content of the file requested).

All the protocol is encoded in the UTF-8 character encoding.

Initial Request Line: The initial request line is composed of three parts, separated by spaces: a method name, an identifier (recognized by the server) of the requested resource and the version of HTTP used. Your server should only support 3 methods: GET, PUT and POST. The typical initial request lines will look like this:

```
GET /path/to/resource HTTP/1.1
PUT /path/to/resource HTTP/1.1
POST /path/to/resource HTTP/1.1
```

Initial Response Line: The initial response line (also called the *status line*) is also composed of three parts, separated by spaces: the HTTP version used, a status code (denoting if the processing of the request was successful or not) and a phrase in plain English describing the status code. For example, a successful response initial line might look like this:

```
HTTP/1.1 200 OK
```

And an unsuccessful one (for example, the requested resource was not found):

```
HTTP/1.1 404 File not found
```

Your server should support the following status codes (you can find here a complete list of all status codes):

- **200 OK** - Standard response for successful HTTP requests.
- **201 Created** - The request has been fulfilled and resulted in a new resource being created.

- **202 Accepted** - The request has been accepted for processing, but the processing has not been completed.
- **204 No Content** - The server successfully processed the request, but is not returning any content.
- **206 Partial Content** - The server is delivering only part of the resource.
- **400 Bad Request** - The request cannot be fulfilled due to bad syntax.
- **404 Not Found** - The requested resource could not be found but may be available again in the future.
- **500 Internal Server Error** - A generic error message, given when no more specific message is suitable.

When answering with a 500 internal server error, the error string should be as verbose as possible.

Header Lines: The structure of header lines is the same for requests and responses. The purpose of the header lines is to provide additional information about the request or the response. Header lines are also structured as simple text lines, one line per header of the form "Header-name: Value". Header lines comply with the following rules (see RFC-822):

1. Each line should end with a line-feed ASCII character (new line).
2. The header name is not case sensitive (but the value is). Please note that since most of our testing is automatic, you must use the headers exactly as they appear in this document.
3. Any number of spaces and lines can be between the : and the value.
4. Header lines beginning with a space or a tab are part of the previous header line.
5. Multiple values in a single header line are separated by a comma. The following header lines are equivalent:

```
HEADER-1:  value-a, value-b
HEADER-1:  value-a,
           value-b
```

For all requests, you are asked to supply the *Host* header which specifies the server IP or DNS name. For all responses, you are asked to supply the *Server* header which also specifies the server IP or DNS name. For any other request/response that includes a body you are asked to supply the following headers:

- Content-Type: specifies the MIME-type of the data in the body of the response (more on that later).
- Content-Length: specifies the length of the data in the body of the response in number of bytes.

In addition, you can add as many custom headers you like.

3.2 REST Terminology

The *REST* (Representational State Transfer) architectural style was developed in parallel with HTTP/1.1, based on the existing design of HTTP/1.0. REST has a terminology that we will use through this document (you can read more here):

- *Resource:* The concept of resource is primitive in the Web architecture, and is used in the definition of its fundamental elements. A resource is simply the entity, item, or thing you want to expose. You are already used to resources as a web browsing consumer. An example for a resource - http://en.wikipedia.org/wiki/Representational_state_transfer
- *Representation:* A representation of a resource is typically a document that captures the current or intended state of a resource. An example for a representation of the previous resource example - http://en.wikipedia.org/w/index.php?title=Representational_state _transfer&useformat=mobile. The same resource could be rendered in different representations, depending on the needs of the client and/or the capacity of the server.
- *URI:* A URI (uniform resource identifier) is a string of characters used to identify a name or a resource on the Internet. Such identification enables interaction with representations of the resource over a network (typically the World Wide Web). An example for a URI can be the 'local path' part of the initial request line of a HTTP request.
- *URL:* A URL (universal resource location) is a URI that, in addition to identifying a network-homed resource, specifies the means of acting upon or obtaining the representation: either through description of the primary access mechanism, or through network "location". For example, the URL http://www.cs.bgu.ac.il/ identifies a resource—CS's home page, and implies that a representation of that resource (such as the home page's current HTML code, as encoded characters) is obtainable via HTTP from a network host named www.cs.bgu.ac.il.

  URLs are used by a Web client to ask for specific resources and representations from the Web server.

We will use resources to describe a photo that was uploaded by the client, and all the jobs and photos that are related to that original photo (that is, the "photo" resource is an abstract resource, which corresponds to several representations, as different files and different metadata descriptions). We will use representations to describe a certain image which can be either the original image or any of the derived images (the original image with applied effects).

3.3 Handling URLs

Your work manager will support only one type of resources: dynamic pages. The URL requested is provided in the initial line of the client's request (after the get method). All special characters (e.g., spaces) are converted to their hexadecimal representation. For example, if you give the following URL to a Web browser: `http://www.cs.bgu.ac.il/plopy plapa` the following request will be generated: `GET /plopy%20plapa HTTP/1.1`. Note how the space between plopy and plapa is changed to the encoding %20 (which is the hexadecimal value of the encoding of the space character). To convert from a URL to a regular string, you can use the java.net.URLDecoder class.

3.3.1 Dynamic Pages

A *dynamic page* is a kind of Web page that has been prepared with fresh information (content and/or layout), for each individual viewing. It is not static because it changes according to time (e.g. news content), the user (e.g. preferences in a login session), the user interaction (e.g. web page game), the context (e.g. parametric customization), or any combination thereof.

3.3.2 MIME-types

MIME is an agreed upon protocol to encode files and data inside other messages (for example, in emails). MIME-types are strings which identify the content of the encoded data. For example, the MIME-type of a regular html file is text/html. An HTTP server must associate the MIME-type in the responses it sends to clients, so that the client can determine what to do with the responses content. For example, a Web browser determines whether to pass the content to an html renderer or to a picture renderer depending on the MIME-type it receives. As the "Content-Type:' header of the server response is mandatory in this assignment, you will need to identify the MIME-type of the file being sent to the client and place the appropriate identifier in the header. To this end, you can employ the supplied MimeType class (which is thread safe). The constructor receives a file name containing the MIME database (which is also supplied in the assignment site). The type recognition is based on simple file extension matching, which suffices for your simple web server.

Using the MimeType class to extract a MIME-type is simple:

```
MimeType mt = new MimeType("MIME.types");
String MIME_type = mt.getType("filename.extension");
```

4 XML

XML (eXtensible Markup Language) defines a set of rules for encoding documents in machine readable form. It is a markup language for documents containing structured information. Structured information contains both content (words, pictures, etc.) and some indication of what role that content plays (for example, content in a section heading has a different meaning from content in a footnote, which means something different than content in a figure caption or content in a database table, etc.). Almost all documents have some structure. XML is important to know, and easy to learn at first although the details of the formalism can become tricky (further material here and here). For this assignment, the short intro below provides sufficient information.

4.1 XML Syntax

XML documents form a tree structure. XML documents must contain a root element, which is considered as the "parent" of all other elements in the document. Each element may have one or more sub-elements, resulting in a branching tree with one root.

- Open tag: <elementName>
- Close tag: </elementName>
- Element: <elementName> elementValue </elementName>
- Child element:

  ```
  <parentElement>
          <childElement> childElementValue </childElement>
  </parentElement>
  ```

- Attribute: <element attributeName="attributeValue"> elementValue </element>
- Comment: <!– this is a comment –>

Important:

- All XML elements must have a closing tag.
  Incorrect: <e1> <e2> </e1>
  Correct: <e1> </e2> </e2> </e1>
- XML tags are case sensitive <e1> != <E1>
- XML elements must be properly nested
  Incorrect: <e1> <e2> </e1> </e2>
  Correct: <e1> <e2> </e2> </e1>

- XML documents must have a root element
  Incorrect: <e1> </e1> <e2 </e2>
  Correct: One root allowed only.

- XML attribute values must be quoted
  Incorrect: <e1 att=gf>
  Correct: <e1 att="gf">

- White space is preserved in XML
  <e1> I   like   this </e1> != <e1> I like this </e1>

4.2 An XML Example

A bookstore description:

```
<bookstore>
  <book category="COOKING">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="CHILDREN">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="WEB">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

Figure 2 illustrate a tree encoding for a bookstore example.

4.3 XML and Java

Parsing XML in Java you can use JAXP—Java API for XML Processing. The JAXP API contains multiple XML specific parsers. We suggest using the *DOM* (Document Object Model) parser which is called *DocumentBuilder* in JAXP.

For instance, using the DOM interface, you can parse an entire XML document and construct a complete in-memory representation of the document. Creating a Java DOM XML parser is done using the, javax.xml.parsers.DocumentBuilderFactory class. The

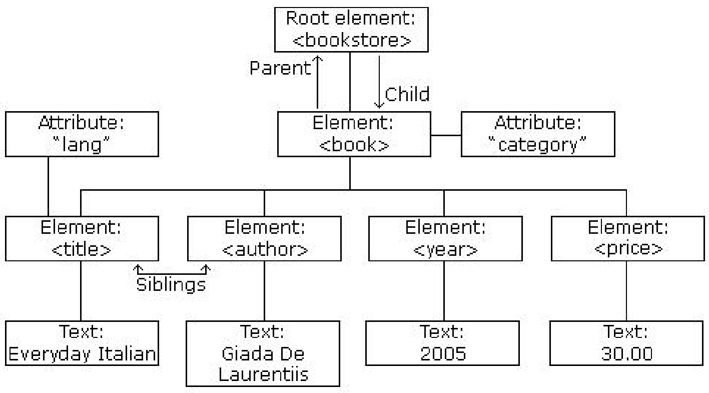


Figure 2: A tree illustration for a bookstore example

DocumentBuilder instance is the DOM parser. Using this parser, you can obtain a Document object which represents the XML file. Follow these links for more information: DocumentBuilderFactory, DocumentBuilder and Document.

In the assignment files you can find an example of using these classes (xml.java).

4.4 XML and C++

In the assignment files you can find an example for parsing XML in C++ (xml.cpp).

5 Clients

Our client can upload new images to server, ask for a new effect or view the results of previous requests.

In order to upload a new image the client should use a PUT request like this:

```
PUT /upload HTTP/1.1
Host: [server name]
Content-Type: image/jpg
Content-Length: [body size]

[image content]
```

The work manager should create a new resource with a new image as representation number 0. The server manages a list of several representations for each resource. These representations correspond to different variants of the image in different formats. The server refers to each representation by a unique number, starting with 0 for the first uploaded image, and going up as new formats are associated to the same resource when the user asks the server to perform transformations on the image.

According to HTTP definitions, the 201 return code is the most appropriate when a new resource is created:

Example 1: HTTP 201 response

```
HTTP/1.1 201 Created
Server: [server name]
Content-Type: text/html; charset=utf-8
Content-Length: [body size]

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>[server name]/photos/[new resource id]</title>
  </head>
  <body>
    <b>HTTP/1.1 201 Created:</b>
    <a href="photos/[new resource id]">Resource [new resource id]</a>
    has been created successfully.<br/>
    Original image can be found <a href="photos/[new resource id]?rep=0>
    here</a>."
  </body>
</html>
```

More on XHTML here.

For any representation of any resource, one can ask for any of the following 3 effects: RGB to grayscale, Gaussian Blur or Resize. Each effect is encoded as an XML expression which is sent to the manager using an HTTP POST message. The client supplies the parameters expected by each effect. The following list specifies the effects and for each one, it indicates which OpenCV function can be used to implement the effect:

1. RGB to grayscale
   - Function: void cvtColor(const Mat& src, Mat& dst, int code, int dstCn=0)
   - XML code:

     ```
     <cvtColor>
       <code> CodeValue </code>
     </cvtColor>
     ```

   - CodeValue: CV_RGB2GRAY
2. GaussianBlur
   - Function: void GaussianBlur(const Mat& src, Mat& dst, Size ksize, double sigmaX, double sigmaY=0, int borderType=BORDER_DEFAULT)
   - XML code:

     ```
     <GaussianBlur>
       <kSize> kernelSizeValue </kSize>
       <sigmaX> sigmaXValue </sigmaX>
     ```

```
  <sigmaY> sigmaYValue </sigmaY>
  <borderType> borderTypeValue </borderType>
</GaussianBlur>
```

- borderType parameter values:
  - BORDER_CONSTANT
  - BORDER_DEFAULT
  - BORDER_ISOLATED
  - BORDER_REFLECT
  - BORDER_REFLECT_101
  - BORDER_REPLICATE
  - BORDER_TRANSPARENT
  - BORDER_WRAP
- kSize:
  - Size of the kernel matrix.
  - Matrix of nxn size where n=3,5,7,9,11

3. Resize
   - Function: void resize(const Mat& src, Mat& dst, Size dsize, double fx=0, double fy=0, int interpolation=INTER_LINEAR)
   - XML code:

```
<resize>
  <scaleFactorX> fxValue </scaleFactorX>
  <scaleFactorY> fyValue </scaleFactorY>
  <interpolation> interpolationMethod </interpolation>
</resize>
```

   - interpolationMethod values:
     - INTER_NEAREST
     - INTER_LINEAR
     - INTER_AREA
     - INTER_CUBIC
     - INTER_LANCZOS4

The complete XML code for a job looks as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<JobRequest>
  <InputRepresentation="[representation id]"/>
  <effectsList>
    <effect_1>
    :
    </effect_1>
  :
  :
    <effect_n>
    :
    </effect_n>
  </effectsList>
</JobRequest>
```

5.1 A Client Request Example

Example: We wish to convert representation 3 of resource 5 to grayscale, resize it to half size, and then apply a Gaussian blur filter over the image. The XML code will be:

Example 2: A client request example

```
<?xml version="1.0" encoding="UTF-8"?>
<JobRequest>
  <InputRepresentation="3"/>
  <effectsList>
    <cvtColor>
      <code> CV_RGB2GRAY </code>
    </cvtColor>
    <resize>
      <scaleFactorX> 0.5 </scaleFactorX>
      <scaleFactorY> 0.5 </scaleFactorY>
      <interpolation> INTER_LINEAR </interpolation>
    </resize>
    <GaussianBlur>
      <kSize> 3 </kSize>
      <sigmaX> 2 </sigmaX>
      <sigmaY> 2 </sigmaY>
      <borderType> BORDER_REPLICATE </borderType>
    </GaussianBlur>
  </effectsList>
</JobRequest>
```

The curl command (more on curl in Section 5.2) will be: `curl -d "@code.xml" -H "Content-type: text/xml" "[server name]/photos/5?rep=3"`, where code.xml contains the xml code for this job . The complete HTTP request will be:

```
POST /photos/5?rep=3 HTTP/1.1
User-Agent: curl/7.21.3 (x86_64-pc-linux-gnu) libcurl/7.21.3 Ope
nSSL/0.9.8o zlib/1.2.3.4 libidn/1.18
Host: [server name]
Accept: */*
Content-type: text/xml
Content-Length: 474

[xml code]
```

5.2 Client Implementation

A client can be any program that can generate HTTP messages. Internet Explorer and Firefox are such programs. Each time you ask for a page—you generate a HTTP GET request (as explained in Section 3.3). When you submit data in a forum—you generate a POST request. However, it will be hard to generate custom requests using the browsers. (Some browsers have plugins that allow you to enter complex requests like POST with XML bodies such as the Poster plugin for Firefox.)

Luckily, we already have in Linux (and Windows) a small program called curl. Using this program you can automatically generate HTTP requests. Here are some examples to learn how to use curl (to read the complete manual, write *man curl* in the shell or press here):

- To GET the main CS Web page (and generate a GET request): *curl www.cs.bgu.ac.il*
- To GET the main CS Web page and see the generated GET request and response at the beginning, add the –trace-ascii flag: *curl –trace-ascii "dumpfile.txt" "www.cs.bgu.ac.il"*. Use `cat dumpfile.txt` to view the trace.
- To add extra headers use the -H flag for each header, like this: *curl -H "header1: value1, value2" www.cs.bgu.ac.il*
- To upload (PUT) the file 1.jpg to the main CS Web page as a binary file (of course it will fail): *curl -T "1.jpg" "www.cs.bgu.ac.il"*.
- To update (POST) the main CS Web page and send an attached file 1.jpg (of course it will fail): *curl –data-binary "@1.jpg" -H "Content-type: image/jpg" "www.cs.bgu.ac.il"*.
- To send (POST) the main CS Web page an XML file as text/XML (of course it will fail): *curl –data-ascii "@1.xml" -H "Content-type: text/xml" "www.cs.bgu.ac.il"*.

You can create automatic scripts (however it is not obligatory) using your shell such as bash (bash manual) to automate a sequence of curl calls.

If you wish, you can also write your client in Java or any other language. However **your program must work with curl** as well. This constraint helps you verify that the Web server you implement is a real implementation of the HTTP protocol and can be used by standard HTTP clients.

5.3 Viewing Jobs and Representations

At any time a client can check the status for a job or a representation, as well as watch the picture of a representation. The most convenient way to do that will be using a standard Web browser. The following list contains all the GET requests of what the client can ask:

1. *See a representation photo:* [server name]/photos/[resource id]?rep=[representation]
2. *See a list of all representations and jobs of a resource:* [server name]/photos/[resource id]
3. *See a list of all the resources* [server name]/photos
4. *See a list of all the jobs* [server name]/jobs

More on this list in Section 6.

6 Photo Hut Work Manager

As explained before, the work manager you are asked to implement is a thin version of a Web server. The server should support a subset of the HTTP protocol. The interaction between a client and the manager is simple: the client opens a TCP connection to the server and sends a request to the server. The server parses the request, and answers the client with a response.

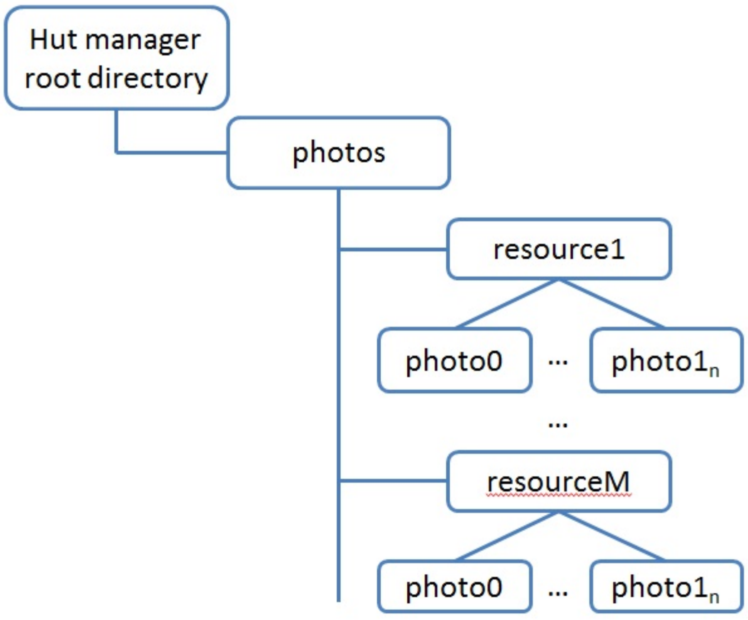
Since the communication to the manager is done only using HTTP, the manager only responds to HTTP requests and never initiates ones. All responses should be dynamic pages (meaning dynamically created), except for representations requests that should result in the picture of the representation.

Each resource and each job has a unique ID. Both IDs are numbers starting form 0, so that resources URIs are photos/0, photos/1, . . ., and jobs URIs are jobs/0, jobs/1, ... Representation IDs are unique only within a resource and they start from 0, where the original uploaded image get that id. In order to manage the jobs, the work manager should hold 3 lists and update them constantly: non-submitted-jobs; submitted-jobs; and finished-jobs. When a new job is submitted it enters the non-submitted-jobs list. This is a FIFO list, and once an employee asks for a new job, he will get the first job in that list. The manager will then move the job to the submitted-jobs list. Again, once the employee finishes the job and uploads the new representation, the manager moves that job to the finished-jobs list.

6.1 Local files and URLs

The only files you are allowed to save on your local file system are the pictures. You can save them wherever you like, though we suggest the directories structure presented in Figure 3, or you can put them all in the same directory with unique names such as `[resource id]-[representation id].jpg`.

Figure 3: Suggested directories structure for the Work Manager



6.2 Work Manager Responses

In this section, we will survey the different server responses for each request. All responses are created dynamically and in all of them the headers section should include the proper content-type and the correct content-Length.

[job status] refers to the list to which the job belongs (non-submitted-jobs / submitted-jobs / finished-jobs).

6.2.1 GET /photos/[resource id]

This request should result in a list of all the representations of a resource. Under each representation you should give a list of all the jobs that use that representation as an input. The result body should be:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <title>[server name]/photos/[resource id]</title>
  </head>
  <body>
    <p>JOB STATUS CATEGORY</p>
    <ol>
      <li><a href="photos/[resource id]?rep=0">Representation 0</a>
       <ul>
         <li><a href="jobs/25">JOB 25</a>. Status - [job status]</li>
         <li><a href="jobs/42">JOB 42</a>. Status - [job status]</li>
                  :
       </ul>
      </li>


                  :
                  :

      <li><a href="photos/[resource id]?rep=N">Representation N</a>
       <ul>
         <li><a href="jobs/48">JOB 48</a>. Status - [job status]</li>
         <li><a href="jobs/49">JOB 49</a>. Status - [job status]</li>
                  :
       </ul>
      </li>
    </ol>
  </body>
</html>
```

6.2.2 GET /photos/[resource id]?rep=[representation id]

This request should result into a 200 OK response. The body should be the correct image. If there is no such representation the response should be 404. If there is such representation, but it is not ready, the response should be 206.

6.2.3 GET /jobs/[job id]

This request should result into a 200 OK response. The XML of the job should appear in the body of the message. Please notice that this XML is the same XML as in Example 2 only that it includes the ¡OutputRepresentation=””/¿ as well:

```
<?xml version="1.0" encoding="UTF-8"?>
<JobRequest>
  <InputRepresentation="3"/>
  <OutputRepresentation="8"/>
  <effectsList>
  ...
  </effectList>
</JobRequest>
```

6.2.4 GET /jobs

This request should result with a `200 OK` response. In the body you should put the following XHTML page:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <title>[server name]/jobs</title>
  </head>
  <body>
    <p>Jobs Status</p>
    <ol>
      <li>Non Submitted Jobs
        <ul>
          <li><a href="jobs/[job id]">JOB [job id]</a>:
          <a href="photos/[resource id]">Resource [resource id]</a>.</li>
          ...
        </ul>
      </li>
      <li>Submitted Jobs
        <ul>
          <li><a href="jobs/[job id]">JOB [job id]</a>:
          <a href="photos/[resource id]">Resource [resource id]</a>.</li>
          ...
        </ul>
      </li>
      <li>Finshed Jobs
        <ul>
          <li><a href="jobs/[job id]">JOB [job id]</a>:
          <a href="photos/[resource id]">Resource [resource id]</a>.</li>
          ...
        </ul>
      </li>
    </ol>
  </body>
</html>
```

6.2.5 GET /photos

This request should result with a 200 OK response. In the body you should put the following XHTML page:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <title>photos</title>
  </head>
  <body>
    <p>Photos</p>
    <ol>
      <li><a href="photos/0">Photo 0</a>
        <ol>
          <li><a href="photos/0?rep=0">Representation 0</a></li>
          <li><a href="photos/0?rep=1">Representation 1</a></li>
          :

        </ol>
      </li>
      <li><a href="photos/2">Photo 2</a>
        <ol>
          <li><a href="photos/2?rep=0">Representation 0</a></li>
          <li><a href="photos/2?rep=1">Representation 1</a></li>
          :

        </ol>
      </li>

      :

    </ol>
  </body>
</html>
```

6.2.6 PUT /upload

This request is used by clients to upload a new image (or resource). The manager should save the image (that is inside the body of the message), create a new resource, set the image as representation 0, and send a response. The response should be a 201 message with a link to the new resource page in the body. Please refer to Example 1 for further explanations.

6.2.7 POST /photos/[resource id]

This request is also used by the client in order to submit a new job. The response should be a 202 message with a link to the output representation in the body (as in Example 1).

6.2.8 PUT /photos/[resource id]?rep=[representation id]

This request is to be used by employees only, in order to upload the output image of their work (in other words—representation). The manager should save the image (that is inside the body of the message), create a new resource, and send a response. The response should be a 201 message with a link to the new resource page in the body. Please refer to Example 1 for further explanations.

6.2.9 POST /jobs/get-new-job

This request is used by employees to ask for a new job. If there is an available job, the server should move the job to the submitted-jobs list and respond with a 200 message. The response should contain a custom header: `resource: [resource id]`, as well as the job XML as described in Section 6.2.3. If there is no available job, the server should response with an empty 204 message.

Important: Please notice that the manager should not send a job when its <InputRepresentation=""/> is not ready yet.

6.2.10 POST /shutdown

If a client calls this page, the server should shutdown gracefully. If an employee tries to reach the server when it is down - he should exit gracefully as well.

6.3 Work Manager Implementation

You will implement the work manager using Java.

Please notice that besides maintaining the 3 lists, you will have to know anytime what resources you have, what representations you have for each resource, what representations are ready, etc. This means you will have to maintain a few more data structures.

6.3.1 Work Manager Communication Architecture

We will use thread per client architecture for our work manager. In your implementation, **do not break the abstraction**, meaning—use a server protocol and a connection handler so that the whole server logic will be detached from the server architecture. This will allow you later to replace the inefficient "thread per client" architecture.

Since both clients and employees communicate the manager using HTTP, the connection handler should be the same for both of them.

The default port number should be 80 though it can get different port from the program parameters. In the assignment code you can find the `UrlGet.java` file which demonstrate an HTTP GET request and response.

7 Photo Hut Employees

Your employees have a simple life cycle:

1. Get a new job by calling `[server name]/POST /jobs/get-new-job`.
2. If there was no available job (message 204) sleep for 30 seconds and try again.

   Otherwise, process the job and submit the result back to the appropriate URI by using `PUT /photos/[resource id]?rep=[representation id]`.
3. Start from the beginning.

If at any time the server is not available (e.g. shutdown), the employee has to close itself gracefully as well.

Please notice that you do not need more than one thread for this implementation.

7.1 C++ Implementation

You will implement the employee using C++ and the OpenCV library. Your program will receive 2 parameters (in this order): [Work manager name/ip], [Work manager port]. In the assignment page you can find an example for a C++ Echo client. You will have to create a makefile for the employee as well. Submitted assignments without a working makefile will not be checked!

Even if you did not implement your work manager yet, you can partially test your employee. Follow the helpdesk home page manual in order to create your own homepage. Inside the `.html` folder that you have just created place 2 files: `some-picture.jpg` and `job.xml`. Write a job inside the xml file (as described in Section 6.2.3) and check manually that your client sends the GET command correctly (and receives the response) and do the OpenCV task(s).

8 Organizing Your Work and Submission

We suggest you the following milestones:

1. Implement the Employees. You can integrate testings as described in Section 7.1.
2. Implement the Work Manager. At the beginning use the thread per client architecture (`Connection Handler` and `Server Protocol`) and answer only simple GET requests (don't handle the directory scheme or other type of requests).
3. Finish the Work Manager implementation.
4. Handle errors.
5. Test the whole system together.

You can use the following as a check list, which must be completed before submitting your work:

1. Correct implementation of receiving HTTP requests, and responding to them.
2. Handling MIME-type correctly.
3. Correct implementation of employees
4. Correct error reporting to clients and employees.
5. A makefile, which compiles the employees into an executable and the work manager into a single jar named Manager.jar.
6. Make sure the following command executes your manager: `java -jar WebServer.jar [Work manager port]` You can find help in this link, follow the The Basics section, and look at the table.
7. Finally you should submit one zip that contains 2 directories: manager and employee. These directories must include the source files. Place the makefile in the root directory of the zip file together with the manager and employee directories.

9 Questions?

Please send any further questions/clarifications to the course Forum.

Good Luck
and Have Fun !