1 Project 3

Due: Mar 29 by 11:59p

This document first provides the aims of this project. It then lists the requirements as explicitly as possible. Subsequently, it provides a log of the project in operation to help understand the requirements. It then describes the files you have been provided with. Finally, it provides some hints as to how those requirements can be met.

NOTE: Intra-document links (in red) do not work if you are reading the PDF version of this document.

1.1 Aims

The aims of this project are as follows:

- To implement some simple web services.
- To expose you to using the express.js server framework.
- To give you some more experience with asynchronous code.

1.2 Requirements

This project requires you to implement web services which allow adding and retrieving images from a database as well as using the stored images to perform steganography.

An image is identified by two strings:

group A non-empty string which does not contain any `NUL` or `/` characters.

name A non-empty string which does not contain any `NUL` or `/` characters

Note that above restrictions on *group* and *name* allow them to be path components within a URL or a Unix filesystem path.

An example identification for an image might have *group* `inputs` and *name* `rose`. Since *group* and *name* cannot contain `/` characters, we can unambiguously use the single string *group*/*name* to identify an image, for example `inputs/¬ rose`.

Note that an image id like `inputs/rose` does not say anything about its representation. The image representations which must be handled by your project are PNG and PPM.

Set up your gitlab project so that it contains a `prj3/steg-ws` subdirectory within the top level `submit` directory. It should be possible to install your project by cloning your gitlab project, setting the current directory to its `submit/prj3/steg-ws` subdirectory and then running `npm install`.

After the install, the `submit/prj3/img-store` should contain a `index.js` executable file which when run will start a web server on the port specified by its first command-line argument. This web server should support an underlying database and should accept the following HTTP requests:

Image store service: `POST /api/images/`***group*** The body of this request should be of type `multipart/form-data` containing a field `img` which specifies a file upload of a PNG or PPM file. A new image should be created in the underlying database with id *group*/*name* where *group* is the *group* suffix of the request URL and *name* is a new name.

If the request is successful, it should return a response with status `201 CREATED` and return a `Location` header where the uploaded image can be accessed using the next web service.

Image retrieval service: `GET /api/images/`***group*****/*****name*****.*****type*** Return the *type* (either `png` or `ppm`) representation for the image with id *group*/*name*.

Image meta-information service: `GET /api/images/`***group*****/*****name***`/meta` If successful, the response body should consist of a JSON object containing meta-information for the image with id *group*/*name*. The return'd JSON must have the following fields:

`"width"` The width of the image.

`"height"` The height of the image.

`"maxNColors"` The maximum number of colors in the image.

`"nHeaderBytes"` The number of bytes in the header of the PPM representation of the image.

`"creationTime"` The time at which this image was stored in the underlying database. The time is represented as the number of milliseconds since the epoch `1970-01-01T00:00:00Z`.

Image list service: `GET /api/images/`***group*** If successful, the response body should consist of a JSON list containing the names of all the images stored under *group*.

Message hide service: `POST /api/steg/`***group*****/*****name*** This request is used to hide a message using the image identified by *group*/*name* which should already be in the database. The new image containing the hidden message will be stored in the database.

The body of this request should consists of a JSON object containing two fields:

`outGroup` A string which specifies a group for the newly created image containing the hidden message.

`msg` A string specifying the message to be hidden.

A new image containing the hidden message should be created in the underlying database in the group specified by `outGroup` with a new name.

If the request is successful, it should return a response with status `201 CREATED` and return a `Location` header which can be used to retrieve the hidden message using the next web service.

Message unhide service: `GET /api/steg/`***`group`***`/`***`name`*** The response body should contain a JSON object with a single field `msg` which contains the message which is hidden in the image identified by *group*/*name*.

Any errors in the above requests should be reported using suitable HTTP status codes. The body of the error response should be a JSON object containing a `code` field with a suitable value which may be different from the HTTP status code and a `message` field giving details about the error. The server can also log a message to standard error.

All data should be stored in the database called `images` at the mongo url `mongodb://localhost:27017`.

1.3 Example Log

The behavior of the program is illustrated by the following annotated log. It uses curl to exercise the web services.

```
#Note that since the output of many of the commands are not terminated
#by a newline, the LOG contains an extra newline after those
#outputs so that the shell $-prompt always starts on a newline.

#start web server in background (the shell prints [jobId] processId).
$ ./index.js 1234 &
[1] 414
$ listening on port 1234

#create convenience shell variables
$ AUX=$HOME/cs580w/projects/prj3/aux
$ URL=http://localhost:1234

# store rose.png in group inputs.   Options used for curl:
#    --silent:        no output except for what is explicitly requested
#    --form:            specify form parameter using key=value; here we
#                         are setting key as img and value a file upload
#                         (indicated with the @).   Automatically sets
#                         content-type of req body to multipart/form-data
#    --dump-header: dump response headers to - (which means stdout).
# Note that short options -s, -F, -D can be used instead.
```

```
$ curl --silent --form img=@$AUX/inputs/rose.png \
        --dump-header - $URL/api/images/inputs
HTTP/1.1 100 Continue

HTTP/1.1 201 Created
X-Powered-By: Express
Location: http://localhost:1234/api/images/inputs/0.png
...

Created

#retrieve image from URL specified by above Location header but as ppm.
#redirect output into t.ppm.
$ curl --silent $URL/api/images/inputs/0.ppm >t.ppm

#verify retrieved image identical to input image
$ cmp $AUX/inputs/rose.ppm t.ppm

#store logo.png in group inputs.
$ curl --silent --form img=@$AUX/inputs/logo.png \
       --dump-header - $URL/api/images/inputs
HTTP/1.1 100 Continue

HTTP/1.1 201 Created
X-Powered-By: Express
Location: http://localhost:1234/api/images/inputs/1.png
Content-Type: text/plain; charset=utf-8
...

Created

#retrieve image from URL specified by above Location header but as ppm
$ curl --silent $URL/api/images/inputs/1.ppm >t.ppm

#verify retrieved image identical to input image
$ cmp $AUX/inputs/logo.ppm t.ppm

#list all image names stored under group inputs
$ curl --silent $URL/api/images/inputs
["0","1"]

#hide message "hello" in image inputs/0.   New curl options:
#   --header:        Set content type of request to application/json.
#   --request:       Specify HTTP method of POST.
#   --data:          Body to be sent with POST request.
# Note that short options -s, -H -X, -F, -d can be used instead.
```

```
$ curl --silent --header 'Content-Type: application/json' \
       --request POST \
       --data '{ "outGroup": "outputs", "msg": "hello" }' \
       --dump-header - $URL/api/steg/inputs/0
HTTP/1.1 201 Created
X-Powered-By: Express
Location: http://localhost:1234/api/steg/outputs/2
...

Created

#unhide message by doing a GET on above Location
$ curl --silent $URL/api/steg/outputs/2
{"msg":"hello"}

#put web server back in foreground and terminate using ^C
$ fg
./index.js 1234
^C
$

#combine above steps together using a shell script do-test.sh:
$ $AUX/do-test.sh
usage: ...aux/do-test.sh [-v] URL PPM_IMG_PATH MSG_PATH

#start server in background
$ ./index.js 1234 &
[1] 26934
$ listening on port 1234

#repeat above using shell-script; using -v verbose to see commands executed
$ $AUX/do-test.sh -v  $URL \
     $AUX/inputs/rose.ppm $AUX/messages/hel*
curl --silent \
    --form img='@/home/umrigar/cs580w/projects/prj3/aux/inputs/rose.ppm'
\
     --dump-header '/tmp/headers.txt' \
     --write-out '%{http_code}' \
     --output /dev/null
  'http://localhost:1234/api/images/inputs'
curl --silent \
     --write-out '%{http_code}' \
     --output '/tmp/img.ppm' \
  'http://localhost:1234/api/images/inputs/0.ppm'
cmp '/home/umrigar/cs580w/projects/prj3/aux/inputs/rose.ppm' '/tmp/img.ppm'
curl --silent \
      --header 'Content-Type: application/json' \
      --request POST \
      --data '{ "outGroup": "outputs", "msg": "hello world" }' \
      --write-out '%{http_code}' \
      --output /dev/null \
      --dump-header '/tmp/headers.txt' \
  'http://localhost:1234/api/steg/inputs/0'
curl --silent \
      --write-out '%{http_code}' \
      --output '/tmp/msg.txt' \
  'http://localhost:1234/api/steg/outputs/1'
cmp '.../aux/messages/hello-world.txt' '/tmp/msg.txt'
hello world

#run same test but without verbose -v option
$ $AUX/do-test.sh $URL \
    $AUX/inputs/rose.ppm $AUX/messages/hel*
hello world

#run test using max size msg which can be hidden in rose
$ $AUX/do-test.sh $URL \
    $AUX/inputs/rose.ppm $AUX/messages/*max*
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx...
truncated; total length: 1203

#run test using msg which is too big to be hidden in rose
#note StegError printed on this terminal by server
$ $AUX/do-test.sh  $URL $AUX/inputs/rose.ppm \
    $AUX/messages/*big*
StegError {
  code: 'STEG_TOO_BIG',
  message: 'group/name: message too big to be hidden in image',
  isDomain: true }
hide failed

#put server in foreground and terminate using ^C
$ fg %1
./index.js 1234
^C
$
```

1.4 Provided Files

The lib directory contains `npm` packages which encapsulate solutions to the previous projects.

ppm This is essentially the same as the ppm.js file you were provided with for your previous project.

steg This is a slightly modified version of the `steg_module.js` file from the solution for *Project 1*. The changes include the following:

- Instead of errors being returned as `error` strings within wrapper return objects, errors are now thrown as `StegError` objects containing `code` and `message` properties.
- Since errors are now being thrown, the wrapper objects around return values were no longer necessary and have been eliminated.
- Modifications to accomodate the changes made to `ppm.js` between projects 1 and 2.

img-store This is a slightly modified version of the `img-store.js` file from the solution for *Project 2*. The changes include the following:

- The addition of new methods `clear()` and `putBytes()`. The previous `put()` method which is used to add the contents of an image file to the database becomes a wrapper around the new `putBytes()` method which is used to add the bytes of the image to the database.
- The name for an image being added to the database is optional. When it is not specified, it is auto-generated by this module using consecutive integers starting from `0`. Note that this sequence of names is always reinitialized when this module is loaded.

These packages will need to be installed into your project just like external packages. They are available as `.tar.gz` bundles on the course web site. The details for installing these packages are given at the start of the Hints section.

The prj3 directory contains a start for your project. It contains the following files:

.gitignore A `.gitignore` file set up to tell `git` to ignore the `node_modules` directory.

steg-ws.js A skeleton file. You must complete the `TODO` sections in this file.

README A README file which must be submitted along with the project. It contains a initial header which you must complete; replace the dummy entries with your name, B-number and email address at which you would like to receive project-related email. After completing the header, you may append any additional content which you would like read during the grading of your project.

index.js A file which is used to start up the web server using express.js. In addition to the required single `PORT` command-line argument, it also allows additional arguments specifying paths to image files which are preloaded into the database. These optional arguments may permit easier development.

Note that this file will clear the underlying database each time it is started.

The `aux` directory contains the following subdirectories:

inputs A directory which contains some image files in both PNG and PPM formats.

outputs A directory which contains image files containing hidden messages in both PNG and PPM formats.

messages A directory with files containing messages to be hidden. These include a vanilla `hello world` message, a message which is the largest message which can be stored in the `rose` image, and a message which just exceeds the length of the largest message which can be stored in the `rose` image.

Additionally, the aux directory also contains a do-test.sh shell script which facilitates testing your project using the command-line. It runs a basic test cycle over the web services. It is invoked using the following 3 command-line arguments:

`URL` The base URL at which the web services are running.

`PPM_IMG_PATH` A path to a file containing a PPM image.

`MSG_PATH` A path to a file containing a message to be hidden.

The script performs the following sequence of steps:

1. Uses the image store web service to store the image specified by `PPM_IMG_PATH`.
2. Uses the image retrieval web service to retrieve the image previously stored.
3. Verifies that the retrieved image is identical to the original image specified by `PPM_IMG_PATH`.
4. Uses the message hide web service to hide the message specified by `MSG_PATH` using the image stored earlier.
5. Uses the message unhide service to retrieve the hidden message.
6. Displays the JSON retrieved for the unhidden message; this is truncated for long messages.

1.5 Testing Web Services

There are two main methods for testing web services.

Using command-line tools This is exemplified by the use of curl in this document. The advantages include easy automation; the disadvantages are that they are unintuitive and clumsy to use and often require the use of obscure options to obtain the required behavior.

Using GUI tools These are very intuitive to use with the use of a decent UI. Restlet is recommended for testing REST web services; there is a client available for Chrome. It too allows storing and automation of tests. In particular it does allow using `multipart/form-data` requests. This slightly dated *blog entry* provides details, including screen-shots.

1.6 Working Project

A working version of this project is available at:

`http://zdu.binghamton.edu:1234`

Please note that this URL is only reachable from within the campus network. It may not be particularly stable; if you find it is unreachable, email me and I will restart it.

Note that the above service was started without any images preloaded, but various images may have been added by those using it. Note also that all uploaded images will disappear whenever I restart the server.

1.7 Hints

The following points are not prescriptive in that you may choose to ignore them as long as you meet all the project requirements.

The actual number of lines of code you will need to write will probably be only around 100. What you will need to do is look at online documentation to figure out the details. The hints below start out by giving you detailed information and then get more vague so that you can discover stuff for yourself.

The provided index.js takes care of all command-line processing while the lib packages provide all the necessary PPM, image-store and steganography functionality. All you need to do is implement the `TODO` placeholders in the steg-ws.js file.

You may proceed as follows:

1. Study the *user web services* example which was covered in class.
2. Familiarize yourself with express.js.

3. Familiarize yourself with the provided steg-ws.js skeleton file. In particular, look at the fully implemented image meta-information service as well as the provided utility functions.

4. Start your project by copying the prj3 directory to your `work` directory. Go into your newly created `steg-ws` directory and initialize it using `npm init -y`. Then install the necessary packages:

   (a) Install local packages from `.tar.gz` bundles on the course web site:

   ```
   $ npm install --save \
     http://zdu.binghamton.edu/cs580w/lib/img-store.tar.gz
   $ npm install --save \
     http://zdu.binghamton.edu/cs580w/lib/ppm.tar.gz
   $ npm install --save \
     http://zdu.binghamton.edu/cs580w/lib/steg.tar.gz
   ```

   (b) Install external packages:

   ```
   $ npm install --save express body-parser multer
   ```

   express is a web server framework; body-parser is express.js middleware for parsing HTTP request bodies; since body-parser does not support parsing of `multipart/form-data` bodies, that functionality is provided by the multer package.

   At this point, you should be able to run the image meta-information retrieval service:

   ```
   $ AUX=$HOME/cs580w/projects/prj3/aux
   $ ./index.js 1234 $AUX/inputs/rose.png &
   [1] 9454
   $ rose.png: inputs/rose
   listening on port 1234

  $ curl --silent http://localhost:1234/api/images/inputs/rose/meta
     {"creationTime":152130... nHeaderBytes":13}
   $
   ```

5. Start by implementing the *image retrieval service*.

   (a) Add code to set up a suitable route `setupRoutes()`. Note that express.js patterns can even be used to match **within** path components.

   (b) Add code to the `getImage()` handler. Set up error handling exactly as done for `getMeta()`. Your action code will simply need to retrieve the image bytes using a suitable img-store method and return those bytes as binary. You will need to set the `Content-Type` of the response depending on the image type you are returning; express.js's res.type() makes this trivial.

You can test your changes by using the extra arguments facility of index.js to preload image files into the database.

6. Proceeding as in the previous step, add code for the *image list service*.
7. Now attempt the *message unhide service*. You will need to retrieve the image bytes from the image store and create a new `Steg` object on those bytes and then use its unhide method. You can use express.js's res.json() method to take care of sending the response; this will automatically take care of setting up the `Content-Type` header.

   You can test this service by preloading your server with the steg'd image contained in the outputs directory.
8. Now work on the *message hide service*. Since the request body uses JSON, you will need to set up a JSON parser as middeware for this service.
9. Finally implement the *image store service*. You will need to use the multer middleware to parse the `multipart/form-data` request body. You can choose to store the uploaded image in either memory or a temporary file. Note that the `multer` API allows you to access the `originalname` of the file; you can use this to determine the image type.
10. Iterate until you meet all requirements. Verify that you detect errors and return appropriate error messages.

1.8 Late Submissions

If you are submitting the project late, please adhere to the new late submission policy announced a few weeks ago. Specifically, please email me a message before the due date, followed by another one once you have completed the project.