Università
della
Svizzera
italiana

Facoltà
di scienze
informatiche

Software Atelier III – Web 2.0 Technologies
Prof. Cesare Pautasso

Saeed Aghaee, Daniele Bonetta, Vasileios Triglianos

Node.JS Project – 22.10.2013 / due 29.10.2013

The goal of this project is to begin to work with Server-Side JavaScript based on Node.JS.

Exercise 1. Static File Server (40pt) - REQUIRED

Starting from the skeleton of the hands-on lecture, create a node.js server to let browsers download files present in your hard disk. The file server should respond to the "/file/*" URI, and should map it to a local folder called "/NodeStaticFiles".

For instance, suppose to have the following three files on your local disk:

/NodeStaticFiles/file01.txt
/NodeStaticFiles/file02.json
/NodeStaticFiles/file03.html
/NodeStaticFiles/dir/file04.pdf

Your node.js server should reply to the following HTTP requests serving the content of the corresponding file in the responses:

GET /file/file01.txt → Should start the download of "file01.txt"
GET /file/file02.json → Should start the download of "file02.json"
GET /file/file03.html → Should start the download of "file03.html"
GET /file/dir/file04.pdf → Should start the download of "dir/file04.pdf"

If the URI provided by the browser does not match an existing file pathname, the server should respond with 404 Not Found. For example:

GET /file/file05.html → Should answer with the error status code: 404 Not found

The server should respond with 405 Method Not allowed to requests which use a method other than GET.

It should be possible to download multiple files at the same time.

Exercise 2. MIME Type Headers (20pt) - REQUIRED

Your static file server should indicate in the response header the correct "Content-Type" field. The server should automatically generate the header according to the file extension. For instance, the following request:

HTTP GET /file/file03.html

Should generate an answer with the following HTTP header:

Content-Type: text/html

Types to be supported are: text/html, text/javascript, text/css, text/plain, application/json, video/mp4, video/ogg, image/gif, image/jpeg, image/png, audio/mpeg, application/zip, application/pdf

Exercise 3. Directory Explorer (40pt) - REQUIRED

The server should be extended to serve a listing of the content of directories.

To do so, add an handler called explore(request, response) which serves requests to the "/explore/*" URI. The handler should allow the client to explore your hard disk starting from the "/NodeStaticFiles" folder.

The handler should answer to any GET request with an HTML page containing links to all the folders and the files directly included in the given path.

For instance, given the following structure:

```
/NodeStaticFiles/file01.js
/NodeStaticFiles/subdir01/file02.txt
/NodeStaticFiles/subdir02/file03.jpg
```

The file explorer should answer to the query "GET /explore" with an HTML page listing the content of the "/NodeStaticFiles" folder and including links to navigate in the subfolders as well as to download the content of the files using the "/file/*" handler.

.	(link to the current directory)
file01.js	(link to download the file)
subdir01	(link to explore the directory)
subdir02	(link to explore the directory)

By clicking on "subdir01" the browser should generate a GET request to "/explore/subdir01/", and the server should respond with the following information:

.	(link to the current directory)
. .	(link to the parent directory)
file02.txt	(link to download the file)

Exercise 4. File Upload and Folder Creation (25pt) - OPTIONAL

Add an handler called upload(request, response) serving requests to the "/upload" URI.

The handler should answer GET requests with an HTML form that contains the input fields required to upload a file. The form should allow the user to upload with a POST request any arbitrary file to the "/NodeStaticFiles" folder (or one of its subfolders).

Also, the form should allow to create new subfolders.

Hint: the upload() handler will have to manage both ''GET'' and ''POST'' methods.

Exercise 5. File Statistics (25pt) - OPTIONAL

Add a new handler called stats(request, response) that answers to the URI "/stats?file=".
The handler should answer with an HTML page containing some information about the number of words appearing in the text file specified by the user. More in detail, the handler should do the following :
1) Check that the file is a text file.
2) Open the file specified by the user (through the "?file=" query parameter)
3) For each word appearing in the file, count its frequency (i.e., the number of times each word is present)
4) Generate an HTML page showing the statistics about the frequencies.

For example, the following request:

```
HTTP GET /stats?file=subdir01/file02.txt
```

Should produce an HTML page with a content similar to:

The file "file02.txt" has the following word frequencies: the word "usi" appears 10 times, the word "Lugano" appears 22 times, and the word "Switzerland" appears 3 times.

You can also display the frequencies in an HTML table.

Statistics should be generated only for files with extension "txt" and "html". For the other cases the server should answer with an error (400 Bad Request)

Hint: you can reuse and extend your frequency counting algorithm so that it runs on the server.

Exercise 6. Tag Cloud (25pt) - OPTIONAL

Add an handler called cloud(request, response) that answers to the URI "/cloud?file=".

The handler should use the functionalities defined in the "stats()" handler (developed in the previous exercise) to obtain the information about the word frequencies. Then, it should answer with an html page showing a "tag cloud" relative to the file.
The tag cloud should be generated using plain HTML and CSS, and should correspond to the word frequencies (i.e., words with an high frequency should be rendered with a font size bigger than words with lower frequencies).

We do not expect you to produce complex visualizations of the data. However, the nicer the tag cloud will look like, the more extra points you will get.

Exercise 7. JSON Tag Cloud (25pt) - OPTIONAL

Extend your cloud(request, response) handler to manage the "Content-Type" HTTP field of the client request.

The handler should answer with the HTML tag cloud (i.e., the page generated in the previous exercise) by default. When the client indicates the "application/json" mime type in the "Accept" header, the handler should send back the word frequencies table in JSON.

For example, the following client request:

```
GET /cloud?file=file02.txt
Accept: application/json
```

Should generate an answer with the following header and body:

```
200 OK
Content-Type : application/json
…
{ 'usi' :10, 'Lugano' : 22, 'Switzerland' : 3 }
```

Requirements

You have to complete all of the required features (100 points) and choose any optional feature (25 points each).

You can work on this project in teams of up to two people, as long as you tell us in advance the team members using the Moodle notes.

The project should be uploaded on Moodle as a .zip file of your source files (excluding the files you have used to test the tag cloud). *You should also put a small readme.txt file in the .zip archive, containing the following information:*

- *Names of the team members*
- *Enumerate the optional features you have implemented*

You should test your project with at least 64 files in different subfolders.

References
http://nodejs.org/api/fs.html
http://www.nodebeginner.org/
http://www.jetbrains.com/webstorm/webhelp/running-and-debugging-node-js.html