Murdoch
UNIVERSITY

index.js --> server.js --> router.js (Advanced) --> requestHandler.js

Application Development in Node.js: Text Upload

Lecture 4 (A)

searchForm (4a) + submit form (4b)

Assignment One

- Assignment one is due on Friday in Week 7.
  - You are strongly advised to complete Lab 3 and Lab 4 before attempting a final prototype solution for the assignment
  - However, you definitely should read the question a few times after its release and start planning and developing initial and intermediate prototypes

Assignment One

- If you have not finished Lab 3 and Lab 4, do so as soon as possible, then you will be equipped to start serious working on the assignment
- All students should submit their assignment on LMS according to the instructions in the assignment question sheet AND have their working application residing under their home directory on ceto.murdoch.edu.au
- Late submission penalties will apply - refer to the unit information and learning guide and the assignment question sheet

Murdoch UNIVERSITY

Lecture Objectives

- Relevance to unit objectives:
  - Learning objective 1: Learning technical Client/Server details
  - Learning objective 2: Writing software
  - Learning objective 3: Requirements for Internet solutions
- Demonstrate the process of developing a Web Server application with Node.js

Recapitulation

- In last week's lectures, we developed the code for a very basic HTTP server (in the file named `server.js`), which can receive HTTP client requests
  - We demonstrated how to encapsulate the server functionality in a function and export that function, so that other scripts can import and use the server

Recapitulation

- We also covered some preliminaries for our application development (see lecture 3C)
  - We developed a router script (`router.js`) and exported a `route()` method
  - We re-factored our server script (`server.js`) to allow for the use of the `route()` method
  - We developed a start-up script (`index.js`) to start and control the application
    - Our two modules were imported into this script, which allowed access to the exported methods
- Let's briefly review the above 3 scripts

index.js Script

```
// import our exported modules
var server = require("./server");
var router = require("./router");

// call the startServer() function associated
// with the server object
// pass the route() function associated with
// the router object as its parameter
server.startServer(router.route);
```

server.js Script

```
var http = require("http");   // import http core modules
var url = require("url");     // import url core modules
function startServer(route){
  http.createServer( function (request, response) {
    var pathname = url.parse(request.url).pathname;
    route(pathname);
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }).listen(8888);
  console.log("Server has started.");
}
exports.startServer = startServer;
```

router.js Script


```
// create route function with pathname as parameter
function route(pathname) {
    console.log("Routing a request for " + pathname);
}
// export route function
exports.route = route;
```

Routing to Request Handlers

- Remember, 'routing' means to handle requests for different queries differently
  - Server and router functions are not the place to actually handle requests (it is not their role)
  - Also, such an approach would not scale well once an application becomes more complex
- Functions that have requests re-directed or routed to them are called **request handlers**

`requestHandlers.js` Script

- For our application, we want requests to a query named `/start` to be routed to (or handled by) a different function than requests to a query named `/upload`
- So, let us create a module called `requestHandlers.js`
  - In this module we will add placeholder functions (request handlers) for the two requests `/start` and `/upload`
  - We will then export these request handlers as functions of the module

requestHandlers.js Script


```
function reqStart() {
  console.log("Request handler 'start' was called.");
}
function reqUpload() {
  console.log("Request handler 'upload' was called.");
}
exports.reqStart = reqStart;
exports.reqUpload = reqUpload;
```

`requestHandlers.js` Script

- To re-direct the requests appropriately (and have our request handlers respond appropriately), we can pass a list of request handlers as an object from our main file (`index.js`) to the server (`server.js`), and then from the server on to the router (`router.js`)

```
index -> server -> router
```

`requestHandlers.js` Script

- The list of request handlers (in `index.js`) can be implemented using an appropriate data structure
  - We will use the associative array notation for objects, as this allows us to use the query as a key to its value; *in our case that value will be one of the request handler functions*
- This design approach demonstrates high cohesion and low coupling, making the design modular; it is thus more flexible and scalable

Re-Factor index.js

```
var server = require("./server");
var router = require("./router");
var requestHandlers = require("./requestHandlers");

// create 'handle' object literal
var handle = {};

// using the associative array notation, each array
// index is an object property which points to an
// appropriate request handler
handle["/"] = requestHandlers.reqStart;
handle["/start"] = requestHandlers.reqStart;
handle["/upload"] = requestHandlers.reqUpload;

// pass handle object (and route function) to server
server.startServer(router.route, handle);
```

Re-Factor `index.js`

- We have imported the `requestHandlers` module
- We then create an empty object `handle`
- Using the associative array notation, we create **property:value** pairs for each of our request handlers
  - These map different queries (as keys) to the appropriate request handler
- We then pass the `handle` object into `startServer()` as its second argument

Re-Factor server.js

```
var http = require("http");   // import http core modules
var url = require("url");     // import url core modules

function startServer(route, handle){
  http.createServer( function (request, response) {
    var pathname = url.parse(request.url).pathname;
    console.log("Request for "+pathname+" received.");
    route(pathname, handle);
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
  }).listen(8888);
  console.log("Server has started.");
}
exports.startServer = startServer;
```

Re-Factor router.js

```
function route(pathname, handle) {
  console.log("About to route a request for: " + pathname);
  // note access via associative array notation
  // if the path points to a function i.e., request handler
  if (typeof handle[pathname] === 'function') {
    handle[pathname](); // call the appropriate function
  } else {
    console.log("No handler found for: " + pathname);
  }
}
exports.route = route;
```

`router.js` Script

- Firstly, the `route` function accepts the second parameter `handle`
- Then we need to check if a request handler for the given query (i.e. pathname) exists
- This is done using the **`typeof`** and **`===`** operators:
  - Recall the **`typeof`** operator returns the data type of its operand

`router.js` Script

- The === operator performs identically to == except it does **not** perform type conversion
  - So **`typeof`** must match exactly the **`function`** data type
- Thus, `handle[pathname]` should point to one of our request handler functions
- If it does point to a function, the `if` statement will be true and the appropriate function is called
- If it does not point to a function, the else clause prints an error message to `console.log`

`router.js` Script

- Thus, we can access our request handler functions from our `handle` object, just as we would access a value in an associative array; via its **key (or property)**
- This is done via the succinct expression

  `handle[pathname]()`**`;`**
  - Note `handle[pathname]` will resolve to a request handler, and the parenthesis designates a function call

router.js Script

- So, depending on what pathname is in the request, the keys/properties could be:

```
handle['/']();              OR
handle['/start']();         OR
handle['/upload']();
```

- Of course, we would not see the above keys in the square brackets because they are the possible values for the **pathname**
- However, the pathname would point to the appropriate function (request handler)

Test Scripts: No Path

- To test, run `index.js` in an ssh terminal, and in another ssh terminal run curl on command line

```
node index.js
curl http://localhost:8888/
```

- Output should be like this:

```
Request for / received.
About to route a request for /
Request handler 'start' was called.
```

Test Scripts: `/start`


```
curl http://localhost:8888/start
```

- Output should be like this:

```
Request for /start received.
About to route a request for /start
Request handler 'start' was called.
```

- NOTE: in each of the previous two tests, the `reqStart` request handler was called

Test Scripts: `/upload`

```
curl http://localhost:8888/upload
```

- Output should be like this:

```
Request for /upload received.
About to route a request for /upload
Request handler 'upload' was called.
```

Test Scripts

- Note you can also run the previous tests in a browser by entering the three different URLs
  - The 'Hello World' browser response upon requesting the previous URLs comes from the anonymous function in our `server.js` file
  - The other outputs using `console.log` are displayed in the ssh terminal that started the server

Responding Request Handlers

- Remember, 'handling requests' means 'answering requests' as well as 'receiving requests'
- Thus, we need to enable our request handlers to speak with the client/browser
- At the moment, our server's anonymous callback function does this

Responding Request Handlers

- A straight-forward (and rather simplistic) approach is to have the request handlers ‘return the content’ they want to display to the client, and send this response data from the anonymous function back to the user: i.e.,

```
request handler -> router -> server
```

Responding Request Handlers

- Of course, the `route.js` and `server.js` code would need to be re-factored to deal with the returned content
- However, we could run into problems – the server's anonymous callback function could become too large and complex if it needs to handle many different types of requests.

Responding Request Handlers

- So instead of bringing the content to the server, a better approach is to bring the server to the content
  - That is, pass the **response** object (from our **server**'s anonymous callback function) through the **router** into the **request handlers**
- The **handlers** will then be able to use this object's functions to respond to requests themselves

Re-Factor server.js

```
var http = require("http");   // import http core modules
var url = require("url");     // import url core modules

function startServer(route, handle){
  http.createServer( function (request, response) {
    var pathname = url.parse(request.url).pathname;
    route(pathname, handle, response);
    // response functions removed from here!!
  }).listen(8888);
  console.log("Server has started on port 8888");
}

exports.startServer = startServer;
```

`server.js` Script

- The response object is passed as the third argument to the `route()` function
- The `response` function calls (`writeHead()`, `write()`, and `end()`), from within the server's anonymous function, have been removed because we now expect the `route()` function and the respective request handlers to take care of their own response output

Re-Factor `router.js`


```
function route(pathname, handle, response) {
  console.log("About to route a request for: " + pathname);
  if (typeof handle[pathname] === 'function') {
    handle[pathname](response); // pass response argument
  } else {
    console.log("No request handler found: " + pathname);
    response.writeHead(404, {"Content-Type": "text/plain"});
    response.write("Resource not found!");
    response.end();
  }
}
exports.route = route;
```

`router.js` Script

- The `response` object is passed as the third parameter to the `route()` function, and also as an argument to the re-directing handler object
- The response error function calls (originally from the server's anonymous function), have been added to the **else** clause of the **if** statement in the `route()` method
  - This takes care of the error output

Useful Request Handler Script

- We will now make use of a core module to demonstrate the functionality in our two request handlers
- We will use the `exec()` function (which belongs to the `child_process` core module) to execute a command such as `ls` (which lists the current working directory)
  - Note: `ls` is finished rather quickly (most of time), meaning it would not overly delay the handling of the next request *even if* the server's anonymous callback functions are run **synchronously**

requestHandler.js Script

```
var exec = require("child_process").exec;
function reqStart(response) {
  console.log("Request handler 'start' was called.");
  exec("ls -lah",
    function (error, stdout, stderr) {
      response.writeHead(200, {"Content-Type": "text/plain"});
      response.write(stdout);  //send the output to client
      response.end();
  });
}
```

- For details of exec method, see

  https://nodejs.org/api/child_process.html#child_processexeccommand-options-callback

requestHandler.js Script


```
function reqUpload(response) {
  console.log("Request handler 'upload' was called.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello Upload\n");
  response.end();
}
exports.reqStart = reqStart;
exports.reqUpload = reqUpload;
```

`requestHandler.js` Script

- Here, the handler function `reqStart` accept the `response` parameter, and makes use of any return value (from the `exec` call) in order to respond to the request directly
  - That is, the response outputs are not executed in the router or the server
- The `/start` handler responds from within the `exec()'s` anonymous callback, and the `/upload` handler still simply replies with "Hello Upload"

Testing Scripts: Fast Request

- This test will make HTTP requests to

  `http://localhost:8888/`
  OR
  `http://localhost:8888/start`

  which respond immediately (a fast request), and requests to

  `http://localhost:8888/upload`

  will be answered almost immediately as well

Testing Scripts: No Path

```
// start app in one ssh shell
node index.js
// use curl as a client in another ssh shell
curl http://localhost:8888/
// client output to console
total 36K
drwx------. 2 macca macca 4.0K Mar  4 10:21 .
drwx------. 7 macca macca 4.0K Feb 25 13:28 ..
-rw-r--r--. 1 macca macca  313 Feb 25 15:52 index.js
-rw-r--r--. 1 macca macca 2.5K Mar  4 10:14 requestHandlers.js
-rw-r--r--. 1 macca macca  401 Mar  4 10:21 router.js
-rw-r--r--. 1 macca macca  411 Mar  4 10:13 server.js
```

Testing Scripts: /start


```
// with server still running
// use curl as a client in another ssh shell
curl http://localhost:8888/start
// client output to console
total 36K
drwx------. 2 macca macca 4.0K Mar  4 10:21 .
drwx------. 7 macca macca 4.0K Feb 25 13:28 ..
-rw-r--r--. 1 macca macca  313 Feb 25 15:52 index.js
-rw-r--r--. 1 macca macca 2.5K Mar  4 10:14 requestHandlers.js
-rw-r--r--. 1 macca macca  401 Mar  4 10:21 router.js
-rw-r--r--. 1 macca macca  411 Mar  4 10:13 server.js
```

Testing Scripts: /upload

```
// with server still running
// use curl as a client in another ssh shell
curl http://localhost:8888/upload
// client output to console
Hello Upload
```

Responding Request Handlers: Slow Request

- We can also demonstrate that this design approach will work with a **slow request**
- This can be done with a call to `exec()` again, but calling a slow command called `"find"` which takes a long time to finish.
- We want to demonstrate that after this slow request, the next request can still be processed immediately without waiting for the slow request to finish.

Responding Request Handlers: Blocking Process

- Note:
  - The command `find` usually takes a long time to finish.
  - The server's anonymous callback function is called each time a new request arrives
  - In Node.js, these multiple calls to the callback function runs concurrently with each other (asynchronously)
  - Therefore, a slow command such as `find` would not delay the handling of the next request.

requestHandler.js Script


```
var exec = require("child_process").exec;
function reqStart(response) {
  console.log("Request handler 'start' was called.");
  exec("find /", { timeout: 10000, maxBuffer: 20000*1024 },
    function (error, stdout, stderr) {
      response.writeHead(200,{"Content-Type":"text/plain"});
      response.write(stdout);
      response.end();
  });
}
```

requestHandler.js Script


```
function reqUpload(response) {
  console.log("Request handler 'upload' was called.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello Upload\n");
  response.end();
}
exports.reqStart = reqStart;
exports.reqUpload = reqUpload;
```

Testing Scripts: Asynchronous Processing

- This test will make HTTP requests to

  `http://localhost:8888/`

  OR

  `http://localhost:8888/start`

  which may take 10 seconds or more to finish, but requests to

  `http://localhost:8888/upload`

  will be answered immediately, even if `/start` is still executing

Testing Scripts

- Run the `index.js`, and in another terminal, run on command line:

```
curl http://localhost:8888/start
```

- Immediately after running the previous command, in the third terminal, run on command line:

```
curl http://localhost:8888/upload
Hello Upload
```

Testing Scripts

- The output for `/upload` should display immediately, even though the `/start` process is still running
- When the `/start` process finishes, the output should be something like the format listed on the next slide
  - Obviously, the actual file names etc., will be those in your file system

Providing Content

- The server, router, and request handlers are in place, and tested using system functions for both quick and slow requests
- So now content can be added to the site which allows users to interact and choose a file, and view the returned file (from the server) in the browser
- As an example, let us look at how to handle incoming POST requests

Handling POST Requests

- Firstly, the server will send an HTML form to the client in response to the /start request.
- The form uses HTTP POST method and contains a `textarea` to get input from the user. When the submit button is clicked, the form data will be sent to the server as /upload request.
- As the HTML code is served by the `/start` request handler, `requestHandlers.js` needs re-factoring

Providing Content: Re-Factor `reqStart()`


```
function reqStart(response) {
  console.log("Request handler 'start' was called.");
  var body = '<html>'+
    '<head>'+
      '<meta http-equiv="Content-Type" content="text/html; '+
      'charset=UTF-8" />'+
    '</head>'+
    '<body>'+
      '<form action="/upload" method="post">'+
      '<textarea name="text" rows="20" cols="60"></textarea>'+
      '<input type="submit" value="Submit text" />'+
      '</form>'+
    '</body>'+
  '</html>';
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write(body);
  response.end();
}
```

Murdoch UNIVERSITY

Providing Content

- Things to note:
  1. The path in the url '`/start`' triggers the `reqStart` request handler
  2. The `textarea` form is sent back to the client (browser) where input from the user will be obtained
  3. Once the text is entered by the user and the submit button pressed, the `form action` calls '`/upload`' which triggers the `reqUpload` request handler (which at this point just returns 'Hello Upload')

Providing Content

- To test this very simple form, enter the request in a browser

  `http://localhost:8888/start`

- Enter some text in the `textarea`, and click the submit button
  - Note that the server responds with ‘Hello Upload’, but does nothing with the entered text
- That is fine for small sized resources which do not block

Providing Content

- However, POST requests can be potentially very large. The data in the POST request may be sent to the server via a sequence of chunks.
- Each time a chunk arrived in the server, the event `data` is triggered. These chunks need to be handled by a callback in response to each `data` event.
- Finally, event `end` is triggered indicating that all chunks in the POST request have been received by the server.

Providing Content

- This can be implemented by adding event listeners to the `request` object that is passed to the `onRequest()` function (or the anonymous callback function) whenever an HTTP request is received
  - Recall last week we used `response.on()`; this week we will use `request.addListner()`

```
request.addListener('data', function(chunk) {
   // called when a new chunk of data was received
});
request.addListener('end', function() {
   // called when final chunk of data received
});
```

Providing Content

- The POST data listeners can be handled by the server (for now), which can then pass the final data on to the router and the request handlers
  - A decision can then be made about what to do with the data received
  - i.e., it is feasible to suggest that an HTTP server's job is to give the application all the data (from a request) it needs to do its job

Providing Content

- The listeners for the `data` and `end` events can be placed in the server script file (for now)
- Here all POST `data` chunks can be accumulated in the `data` callback listener
- The call to the router can happen upon receiving the '`end`' event
- The accumulated data can then be directed to the router, which in turn passes it on to the request handlers

Re-Factor server.js

```
var http = require("http");
var url = require("url");

function startServer(route, handle) {
   function onRequest(request, response) {
      var pathname = url.parse(request.url).pathname;
      console.log("Request for " + pathname + " received.");

      request.setEncoding('utf8');
      // declare variable to accumulate incoming data
      var postData = "";
```

Re-Factor server.js

```
        request.addListener('data', function(dataChunk) {
                // accumulate data here
                postData += dataChunk;
                // only display for testing purposes
                console.log("Received POST chunk'"+dataChunk+"'.");
        });
        request.addListener('end', function() {
                route(pathname, handle, response, postData);
        });
    }
    http.createServer(onRequest).listen(8888);
    console.log("Server has started.");
}
exports.startServer = startServer;
```

Re-Factor `server.js`

- Things to note:
  - The received data is expected in UTF-8 encoding
  - An event listener for the `data` event accumulates into the `postData` variable whenever a new chunk of POST data arrives
  - The call to the router has been moved into the `end` event listener, ensuring that it is only called when **all** POST data has been received
  - `postData` is passed into the `route()` method, as it is needed in the request handlers

Re-Factor `router.js`


```
function route(pathname, handle, response, postData) {
  console.log("About to route a request for " + pathname);
  if (typeof handle[pathname] === 'function') {
     handle[pathname](response, postData);
  } else {
     console.log("No request handler found for " pathname);
     response.writeHead(404, {"Content-Type":"text/plain"});
     response.write("Resource not found!");
     response.end();
  }
}
exports.route = route;
```

Re-Factor `router.js`

- Things to note:
  - The `postData` is passed into the `route()` function as the fourth parameter
  - It is subsequently passed as a second parameter in the re-direction to the request handlers:

```
handle[pathname](response, postData);
```

Re-Factor requestHandlers.js


```
function reqStart(response, postData) {
    .. Same form code as previous versions ..
    .. postData passed in but not used ..
}
function reqUpload(response, postData) {
    console.log("Request handler 'upload' was called.");
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("You've sent: " + postData);
    response.end();
}
exports.reqStart = reqStart;
exports.reqUpload = reqUpload;
```

Re-Factor `RequestHandlers.js`

- Things to note:
  - The `postData` is passed in to both the `reqStart()` and `reqUpload()` functions as the second parameter
  - It is subsequently used in the write function in the `reqUpload()` function
  - It is not used in the `reqStart()` function, but is passed for consistency
    - It is possible it may be required in the future

A Better Solution

- As we have observed in Week 3, the server script is not really the best place in an application to handle data received from a form or other source
- A better solution would be to perform the tasks of accumulating the input form data in request handlers
- In our application, these tasks could be done in the `reqUpload()` function
  - Obviously, to do this we would need to re-factor `requestHandlers.js` and other scripts

A Better Solution

- In the tutorial for Week 4, you will be asked to re-factor the entire application to achieve this better solution
- Think carefully about how to pass the incoming data from the server to the router, then to the request handler
  - Existing parameters between scripts will need to be changed to enable this
  - In particular, the `request` object will be needed to access the `addListner` event handlers
  - Also, `postData` will no longer be necessary

Read the Scripts

- Study the scripts and analyze the operations line-by-line
- Please make sure you read and understand ALL of the code discussed in these lecture notes
  - You will need this understanding to complete the work for Lab 4 and Assignment 1
- Check with JavaScript and Node.js for any commands that you are unsure about

Acknowledgement

- Kissling, M., The Node Beginner Book: A comprehensive Node.js tutorial. 10/10/2015

Application Development in Node.js: Image Upload

Lecture 4 (B)

Lecture Objectives

- Relevance to unit objectives:
  - Learning objective 1: Learning technical Client/Server details
  - Learning objective 2: Writing software
  - Learning objective 3: Requirements for Internet solutions
- Demonstrate the process of developing a Web Server application to upload image files

Recapitulation

- In the previous lecture, we developed an application that:
  - Was designed with a modular approach
    - This made development easier, by re-factoring to cater for progressive changes
  - Provided a script to start the application
  - Consisted of a server, a router, and request handlers
  - Handled requests to post text data to a browser/client

Get Form Data

- The plan now is to show how to extract the user input data from an HTML form on the server.
- This includes getting the uploaded files from the client
- We will also show how to serve an image file from the server in a browser
- We will need to use a number of an external Node.js modules, including `formidable`, `fs`, `util` and `os`

Install `formidable` Module

- The module `formidable` is not a core module, therefore we need to install it on our computer. Type on command line:

```
npm install formidable
```

- To see the version of formidable module installed on your computer:

```
npm view formidable version
```

  - The current version should be 2.0.1 (March 2022)

- To import the module into an application, type the following into the script that uses it:

```
var formidable = require("formidable")
```

Murdoch UNIVERSITY

Usage: `formidable`

- The `request` object in the server contains the form data, such as text box values, selection of the radio button and selected file.
- Although these user input data are already transported to the server side, accessing them from the `request` object is not an easy task.
- Fortunately, `formidable` provides a `parse` method allowing us to easily gain access to these user input data, including the uploaded files.

Usage: `formidable`

- To get the user input from a POST form, we need to create a new `IncomingForm` object, which contains the `parse` method.

```
var formidable = require('formidable');
var form = new formidable.IncomingForm();
```

- The `IncomingForm` can then be used to parse the `request` object to obtain the `field` and `file` that were submitted through the form.

```
form.parse(request, function(error, field, file){
    // access input data from field
    // access file information from file
}
```

Usage: `formidable`

- Once the `request` is parsed, the callback is called with the input data in `field` and file information in `file`:
  - `field` contains the data from all input elements in the form (except file), in the form of a list of *`name`*`:`*`value`* pairs.
  - `file` contains the information about all uploaded files, including the `filepath` on the server and `originalFilename` for each uploaded file.

Example: Get Data From an HTML form

File `start.html`:

```
<!DOCTYPE html>
<html>
<head>
    <title> A HTML Form </title>
</head>
<body>
    <p>Please submit your assignment:</p>
    <form action="/upload" enctype="multipart/form-data" method="POST">
        Student Number  <input type="number" name="studentNumber" value="12345678">
    <br/>
        Student Name <input type="text" name="studentName" value="Jane Doe"> <br/>
        Unit <input type="radio" name="unit" value="ICT582">ICT582
             <input type="radio" name="unit" value="iCT286">ICT286
             <input type="radio" name="unit" value="iCT375" checked="checked" >ICT375
             <input type="radio" name="unit" value="iCT374">ICT376
  <br/>
        Assignment <input type="file" name="assignment" multiple="multiple"> <br/>
        <br/>
        <input type="submit" value="Upload your assignment" />
    </form>
</body>
</html>
```

Access Values of Input Elements

- The HTML form contains input elements for student number, student name, the unit, and the assignment file to submit.
- The data from most input elements (not including file) are available in the field object:

```
{ name : value, name : value, . . . }
```

- For example, if the student number entered by the user is `12345678`, this value can be accessed using the input element's name attribute `studentNumber`:

```
field.studentNumber
```

Access Information About an Uploaded File

- The information about a file is accessible from `file` object using the name attribute of the file element.
- For example, in the previous form, the name attribute for the file input element is `assignment`. Its information is available from `file.assignment` object.
  - `file.assignment.filepath`
    - `File.assignment.path` on Darwin
  - `File.assignment.originalFilename`
    - `File.assignment.name` on Darwin

Inspect Object Details

- You can view the details of an object using `inspect` method from `util` module.
- For example to see the details of the objects `field` and `file`:

```
form.parse(request, function(error, field, file){
    console.log(util.inspect({field: field, file: file});
    . . . .
}
```

Request Handler: `reqStart`

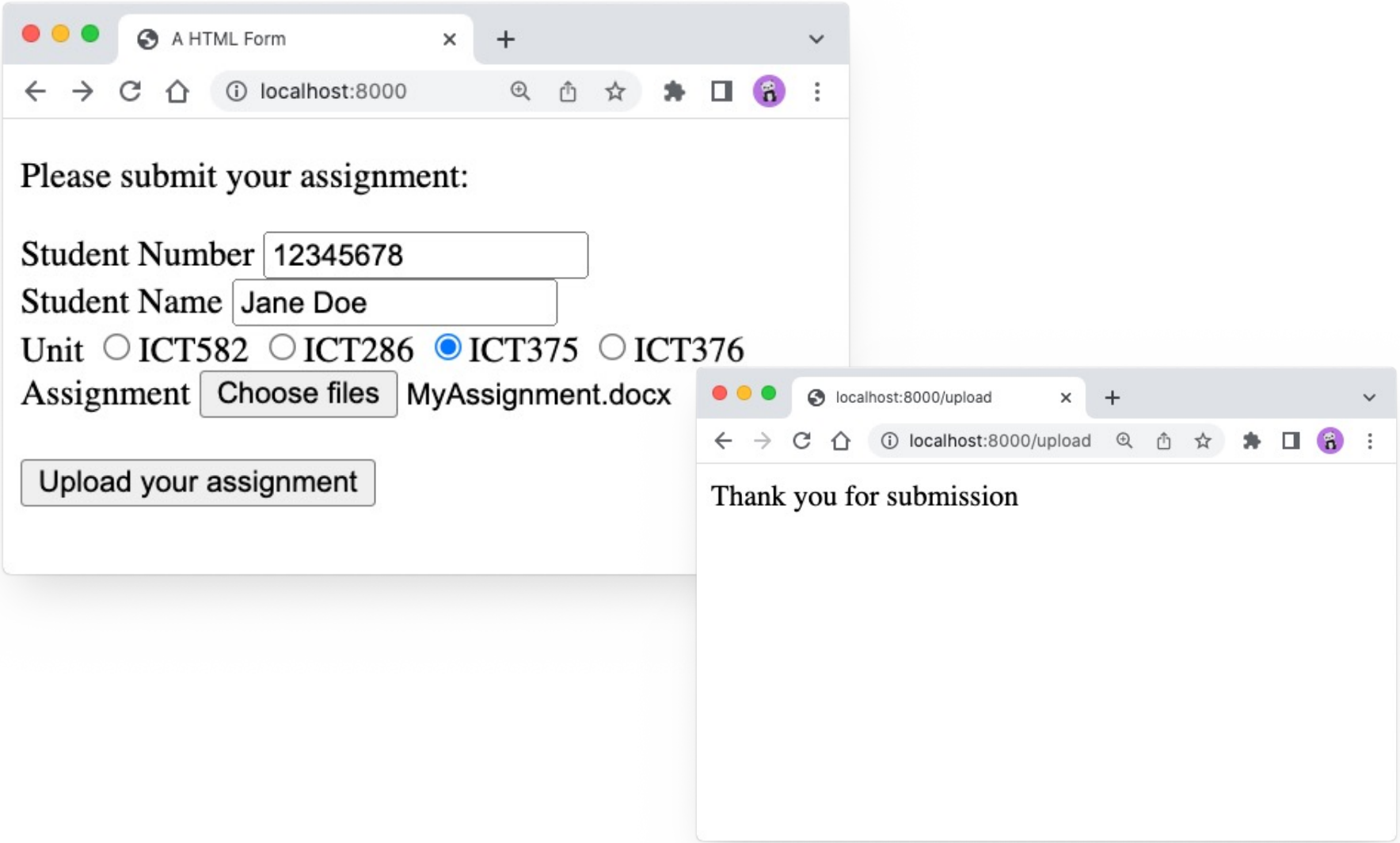

```
// file: handlers/start.js
var fs = require('fs');

function reqStart(request, response) {
    fs.readFile('./handlers/start.html', 'utf8', (err, data) => {
        if (err) {
           console.error(err);
           response.writeHead(404, {'Content-Type' : 'text/plain'});
           response.write('Error reading file "start.html"');
           response.end();
        } else {
           response.writeHead(200, {'Content-Type' : 'text/html'});
           response.write(data);
           response.end();
        }
    });
}

exports.reqStart = reqStart;
```

- Note both the script `start.js` and the html file `start.html` are stored under subdirectory `handlers`

Request Handler: `reqStart`



Request Handler: `reqUpload`

```
// file: handlers/upload.js
var formidable = require('formidable');
var util = require('util');
var fs = require('fs');
var os = require('os');

function reqUpload(request, response) {
    if (request.method == 'POST') {
        var form = new formidable.IncomingForm();
        form.parse(request, function(error, field, file) {
            //console.log(util.inspect({field : field, file: file}));
            var oldpath = os.type() == 'Darwin'
                ? file.assignment.path: file.assignment.filepath;
            var newFilename = os.type() == 'Darwin'
                ? file.assignment.name: file.assignment.originalFilepath;
            var newpath = "./assignments/" + field.studentNumber + "_" + newFilename;
            fs.rename(oldpath, newpath,
                (err) => { if (err) { console.log("error in fs.rename"); }
            });
            response.writeHead(200, {'Content-Type': 'text/html'});
            response.write("<p>Thank you for submission </p>");
            response.end();
        });
    } else {
        response.writeHead(200, {'Content-Type' : 'text/plain'});
        response.write('Hello, upload\n');
        response.end();
    }
}

exports.reqUpload = reqUpload;
```

Murdoch UNIVERSITY

Request Handler: `reqUpload`

- The program gets the information of an uploaded file using the value of name attribute in the file input element, which is `assignment` in the form.
- On the server, the uploaded file is temporarily stored in `file.assignment.filepath`. We constructs a new path by combining the directory path where the file will be moved to (`./assignments/`), the student number, and the original file name from the client:

```
newpath = "./assignments/"
      + field.studentNumber
      + "_"
      + file.assignment.originalFilename
```

Request Handler: `reqUpload`

- The program then move the file from the temporary location to the application specific location:

```
fs.rename(oldpath, newpath,(err) => {
    if (err) {
        console.log("error in fs.rename"); }
});
```

- Note there are some difference between the `formidable` on Windows and MacOS (Darwin).
  - `file.assignment.filepath (Windows)`
  - `file.assignment.path (Darwin)`
  - `File.assignment.originalFilename (Windows)`
  - `File.assignment.name (Darwin)`

Serving an Image File

- We obviously need to serve the content of a file, such as an image, to the client
- We can use Node.js' file system core module (**fs)** for this purpose
- So, let us add another request handler (to our application) for the URL query (`/show`), which will display the contents of an image file (say `test.png`) that resides in the directory `./images` of our server

Request Handler `reqShow`

```
var fs = require('fs');

function reqShow(request, response) {
    response.writeHead(200, {'Content-Type': 'image/png'});

    var readStream = fs.createReadStream("./images/test.png");
    readStream.on('open', function() {
        readStream.pipe(response);
    });
    readStream.on('erro', function() {
        response.writeHead(404, {'Content-Type' : 'text/plain'});
        response.write("File 'test.png' not found\n");
        response.end();
    });
}

exports.reqShow = reqShow;
```

Re-Factor `index.js`

```
var server = require('./server.js');
var router = require('./router.js');

// install request handlers
var handle = {};
handle['/'] = require('./handlers/start.js').reqStart;
handle['/start'] = require('./handlers/start.js').reqSart;
handle['/upload'] = require('./handlers/upload.js').reqUpload;
handle['/show'] = require('./handlers/show.js').reqShow;

server.startServer(router.route, handle);
```

- Note, in a real application, there may be many request handlers. Placing all handlers in one file can make the file unmanageable. It makes sense by placing each handler in a separate script file and place all handlers in the same sub-directory.
- In this application, all handlers, `start.js`, `upload.js` and `show.js` are stored under subdirectory `handlers`

Test `reqShow`

- By restarting the server and entering the following URL in the browser, the image file at `./test.png` should be displayed

  `http://localhost:8888/show`

- Obviously, you will need an image called `test.png` in the subdirectory `./images`

Read the Scripts

- Study the scripts and analyze the operations line-by-line
- Please make sure you read and understand ALL of the code discussed in these lecture notes
  - You will need this understanding to complete the work for Lab 4 and Assignment 1
- Check with JavaScript and Node.js for any commands that you are unsure about

Acknowledgement

- Kiessling, M., The Node Beginner Book: A comprehensive Node.js tutorial. 10/10/2015