Murdoch
UNIVERSITY

index.js, server.js, and router.js (BASIC)

Implementing a Basic Web Server

Lecture 3 (A)

Lecture Objectives

- Relevance to unit objectives:
  - Learning objectives 1, 2, and 3
- To introduce the basic requirements for implementing a web server
- To understand the basic HTTP response handling mechanisms of a web server
- To use Node.js to write a simple web server
- To prepare for Lab 3 and Lab 4

Lecture Outline

- Basic understanding of what sockets are and how they are utilized in the technology that we are learning
- Brief review of HTTP client / server methods
- Introduction to Node.js modules to implement a Simple Web Server

Sockets

- A socket is a programming language concept which allows a TCP or UDP connection to be formed between two network programs
  - They serve as “end-points” of the TCP or UDP connection
    - That is, they are used to set up a “point-to-point” connection
  - Internet client and server code use socket connections to send data from one side to the other

Sockets

- Programmers establish socket connections by setting parameters like port numbers, transport protocol, etc.
- The socket API handles the details for TCP, UDP, IP, etc.
  - This is the concept of ***layered software***

Sockets in Node.js

- In Node.js, the `Socket()` method is available via the core modules **'net'** and **'dgram'**
- However, in many cases, you do not have to use socket directly. For example, in the **http** module, the `request()` method would use the `socket` method to create a TCP connection with the server. As a user of the **http** module, you do not need to use the `socket` method for this purpose.

More About Sockets

- You can learn more details about sockets and how to use their APIs in the unit: **ICT374 Operating Systems and Systems Programming**
- In this unit, we are more interested in the HTTP request-response handling mechanism built on top of the sockets
  - Thus, we do not need to work at the socket level

HTTP Revisited

- An HTTP session is a sequence of network request-response transactions
- A client initiates a request by establishing a Transmission Control Protocol (TCP) connection to a particular port on a host (ie, a server computer)
- An HTTP server listening on that **host : port** sends back a status and response message
  - That is, the body of the requested resource or an error message

HTTP Request Methods

- HTTP defines 8 methods indicating desired action on the identified resource
- The resource could be pre-determined (static) or dynamically generated
  - In most cases, the resource will be a file or the output generated by executing a program / application (stored on the server file system)

HTTP Request Methods

- GET – requests a specific resource
- HEAD – similar to GET but without the body
  - This could be used to retrieve meta-information in the response headers
- POST – submits data to be processed to the specified resource (modifies the resource)
- PUT – uploads a representation of the specified resource (overwrites the resource)

HTTP Request Methods

- OPTIONS – returns the HTTP methods that the server supports for specified URL
  - Can be used to check functionality of a web server
- TRACE – echoes back the received request so that a client can see what the intermediate servers have added or deleted
- DELETE – deletes the specified resource

HTTP Request Methods

- CONNECT – converts the request connection to a transparent TCP/IP tunnel, usually to facilitate SSL-encrypted connection through an unencrypted HTTP proxy
- **NOTE:** HTTP servers are required to implement at least the GET and HEAD methods

Example Client Request

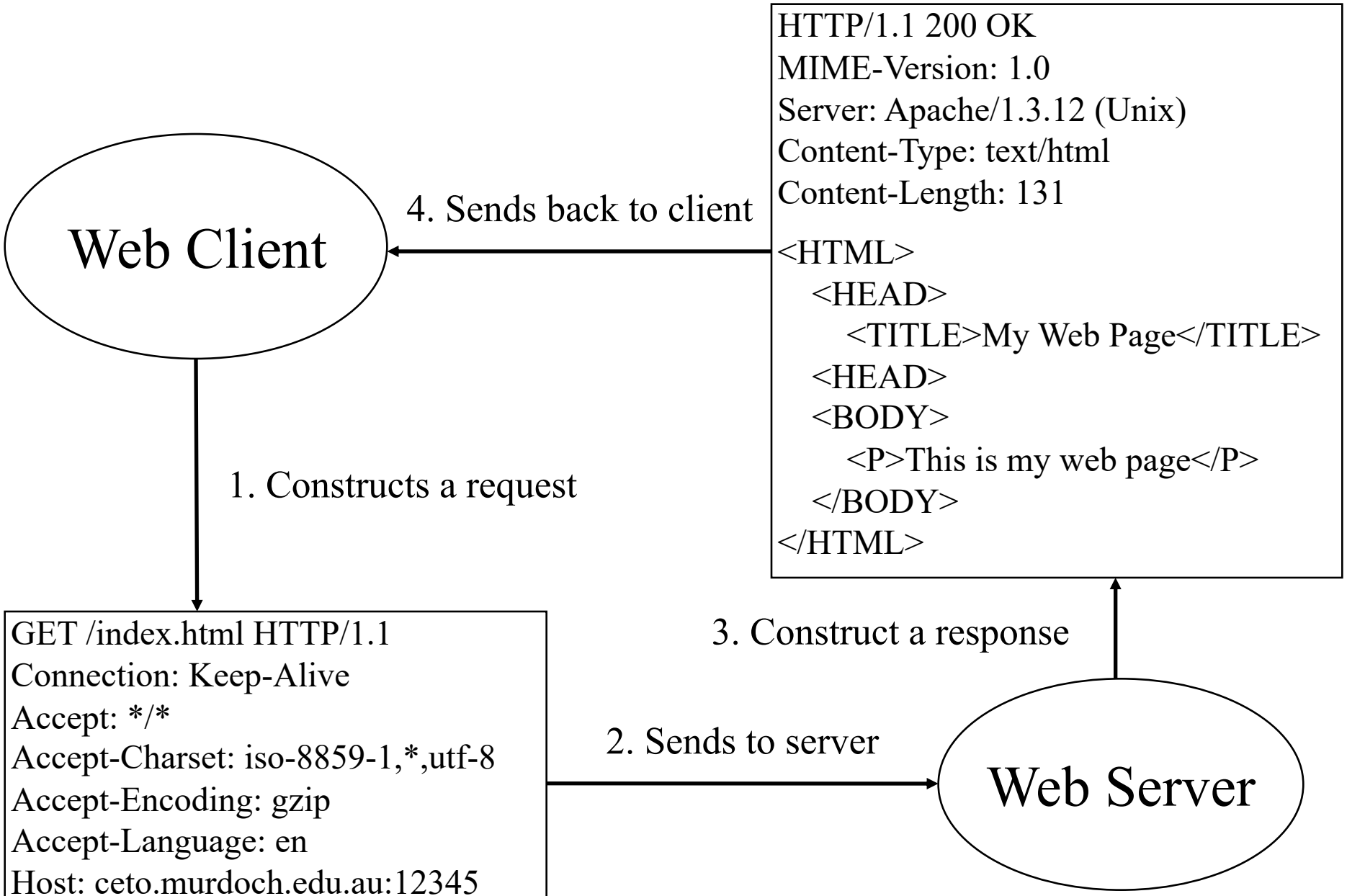
- Client Request

```
GET /index.html HTTP/1.1
Connection: Keep-Alive
Accept: */*
Accept-Charset: iso-8859-1,*,utf-8
Accept-Encoding: gzip
Accept-Language: en
Host: ceto.murdoch.edu.au:12345
User-Agent: Mozilla/4.0
```

Example Server Response


```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Etag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Content-Length: 131
Connection: close
Content-Type: text/html; charset=UTF-8

<HTML>
    <HEAD>
        <TITLE>My Web Page</TITLE>
    <HEAD>
    <BODY>
        <P>This is my web page</P>
    </BODY>
</HTML>
```


HTTP/1.1 200 OK
MIME-Version: 1.0
Server: Apache/1.3.12 (Unix)
Content-Type: text/html
Content-Length: 131
<HTML>
<HEAD>
<TITLE>My Web Page</TITLE>
<HEAD>
<BODY>
<P>This is my web page</P>
</BODY>
</HTML>
4. Sends back to client
Web Client
1. Constructs a request
GET /index.html HTTP/1.1
Connection: Keep-Alive
Accept: */*
Accept-Charset: iso-8859-1,*,utf-8
Accept-Encoding: gzip
Accept-Language: en
Host: ceto.murdoch.edu.au:12345
User-Agent: Mozilla/4.0
2. Sends to server
3. Construct a response
Web Server

Task: An HTTP Web Server

- We need to implement a basic HTTP web server in Node.js, so that we can test various web clients
  - We will develop the server capabilities to be more advanced next week
- It must be able to communicate with HTTP clients using GET and HEAD methods and possibly other methods such as POST, OPTION, TRACE, etc.

Node.js HTTP Server Method

- The syntax of the method used to create a HTTP server in Node.js is:

  ```
  http.createServer([requestHandler])
  ```

  where `requestHandler` is a callback function
- The `createServer` method returns a server object, which can be assigned to a variable
- The optional callback function passed to the server method is called *once every time* an HTTP request is received by the server

Simple Web Server

- Our server script should do something very simple:
  - Respond to a request from a web client
    - Listen on the designated port and IP numbers
    - When a request arrives:
      - Identify the requested resource from the URL
      - Process the request as required
      - Formulate the appropriate response
  - It will use HTTP to do the transport
    - This means it will only be able to communicate with a HTTP client

Simple Web Server Algorithm

- The basic structure of the script (from our requirements on the pervious page) is:

1. LOAD MODULES – http module in this case
2. DEFINE A CALLBACK FUNCTION TO HANDLE INCOMING REQUESTS, PASSING THE REQUEST AND RESPONSE OBJECTS AS PARAMETERS
    a) WRITE A RESPONSE HEADER
    b) WRITE THE RESPONSE MESSAGE
    c) CLOSE THE RESPONSE MESSAGE
3. CREATE THE SERVER OBJECT PASSING THE CALLBACK FUNCTION AS A PARAMETER
4. SET SERVER EVENT LISTENER TO 'LISTEN' ON PORT AND IP NUMBERS

Simple Web Server Script: Callback Function


```
// server.js – layout used for clarity
// import http module
var http = require('http');

// define callback function with parameters
function onRequest(request, response) {
  // write response header and message
  response.writeHead(200,
        {'Content-Type':'text/plain'});
  response.write('hello client!');
  response.end();
}
```

Script Explained

- Firstly, import the appropriate module (**http**)
  - Assign returned object to an instance variable
- Define a function to handle requests
  - We have called this function **`onRequest()`**
  - The caller of the callback function will pass *request* and *response* objects the parameters
  - The function creates a *response* header and writes the *response* message
  - Note the `response.end()` to end the *response* message

Simple Web Server Script: Server Creation


```
// create server object with the callback function
// passed as a parameter
// assign created server to instance variable
var server = http.createServer(onRequest);

// set server to listen on port:ip numbers
server.listen(8888,'127.0.0.1');

// output messages to screen
console.log('Server running at
  http://127.0.0.1:8888/');
console.log('Process ID:', process.pid);
```

Note: when you try the above program on ceto, please replace port 8888 with the one assigned to you!

Murdoch UNIVERSITY

Script Explained

- Use `http.createServer()` method to create and return the server object
  - This method takes **`onRequest`** as a parameter
  - The object is assigned to an instance variable
- Use the instance variable to set the event listener, specifying the port and IP numbers in that order

```
server.listen(8888,'127.0.0.1');
```

- Output a message to screen
- Note: the `process` object is a Node.js global, just like the `console` object.

Running The Web Server

- To execute this server, save the script as file **`server.js`**
- On command line type:

```
-> node server.js
// output to screen
-> Server running at http://127.0.0.1:8888/
-> Process ID: 1234
```

- The reason for printing the process id (pid) is so that the server can be stopped or ‘killed’ once you are finished with it

Important Points

- This raises some very important points:
  - Servers are written to run continually as background processes (i.e., daemons)
    - Numerous clients may attempt to send requests to an http server at the same time
    - An http server listens on a designated port number so that it can handle incoming requests associated with that port number, no matter how many requests there may be
  - Each server you start will continue to run and consume system resources if left running
    - Continually running servers could also compromise security if the server is not configured securely

Important Points

- This raises some important points (cont.):
  - As part of normal operation of a Web server, you would want it to run continually
    - As long as it is configured to run efficiently and securely, there should be no problems with the server machine
  - However, for our usage, it is **very important** to stop or kill any server when you have finished using it
    - Ceto is used by many students doing many units in different countries
    - If everyone leaves their server programs running continually in the background, the machine will become very inefficient and even crash!

Stopping The Web Server

- For Node.js, a server like we have just seen can be stopped or 'killed' using the `kill` command or control keys

```
-> kill -9 1234 (for pid 1234)
(OR you can use Ctl-c on command line)
```

- For servers such as `apache`, you should use the following *apache control* commands to start, stop and restart:
  - `apachectl start`
  - `apachectl stop`
  - `apachectl restart`

IMPORTANT MESSAGE

- When you have finished working on ceto.murdoch.edu.au, **YOU must stop any server process that you have started**
  - The sysadmin and you are the only two people who can kill processes started by you
  - Remember there are many people using this machine, so many people running servers unnecessarily will affect the efficiency of ceto
  - If you do not stop them yourself the sysadmin could become very annoyed

Simple HTTP Web Server: Alternative Layout


```
// an alternative layout commonly used
// NOTE the use of an anonymous callback function as
// the parameter in the call to http.createServer
// import http module
var http = require('http');

// create server with anonymous callback function
http.createServer( function (request, response) {
  response.writeHead(200,{'Content-Type': 'text/plain'});
  response.write('hello client!');
  response.end();
}).listen(8888,'127.0.0.1');

console.log('Server running http://127.0.0.1:8888/');
console.log('Process ID:', process.pid);
```

Murdoch UNIVERSITY

Web Server Script Explained

- Things to note:
  - We do not assign the returned server object to an instance variable; we just call the method from the returned server object
  - We define a callback function (our request handler) and pass it anonymously into `http.createServer()` as a parameter
    - We discussed this approach last week
  - Because we did not assign the returned server object to an instance variable, the `listen()` method is invoked using the dot notation

Exporting the Server

- As discussed last week, in order for other scripts to be able to utilize our web server, we need to export it
  - We only need to export that specific functionality associated with starting the server
  - We can do this by encapsulating the required functionality in a function

Exporting Our Simple Web Server


```
// using alternative layout
var http = require('http');   // import http module
function startServer() {
   http.createServer( function (req, res) {
      res.writeHead(200,{'Content-Type': 'text/plain'});
      res.write('hello client!');
      res.end();
   }).listen(8888);
   console.log('Server running on port: 8888');
}
// export the function
exports.startServer = startServer;
```

Running the Server on Ceto

- When executing this server code on ceto.murdoch.edu.au, you **must** use the port number assigned to you; **not 80 or 8888**
  - For the port assigned to you, see the Unit Information page
- You may also wish to specify the IP number or hostname for ceto in the `listen()` method of the server code
  - If you do not specify the host name or IP address, the server will listen on all network interfaces on the server machine
- **And do not forget to kill the server when you have finished working**

Read the Scripts

- Study the scripts and analyze their operations line-by-line
- Check with JavaScript and Node.js documentation for any code that you are unsure about
- In Lab 3, you will test your server with the three client approaches that we will be discussing in the next set of lecture slides

Acknowledgement

- The code snippets were sourced from:

  Basarat, A.S., Beginning Node.js

- The conceptual theory was derived from:

  Node.js website: https://nodejs.org/en/

  Basarat, A.S., Beginning Node.js

Implementing Web Clients

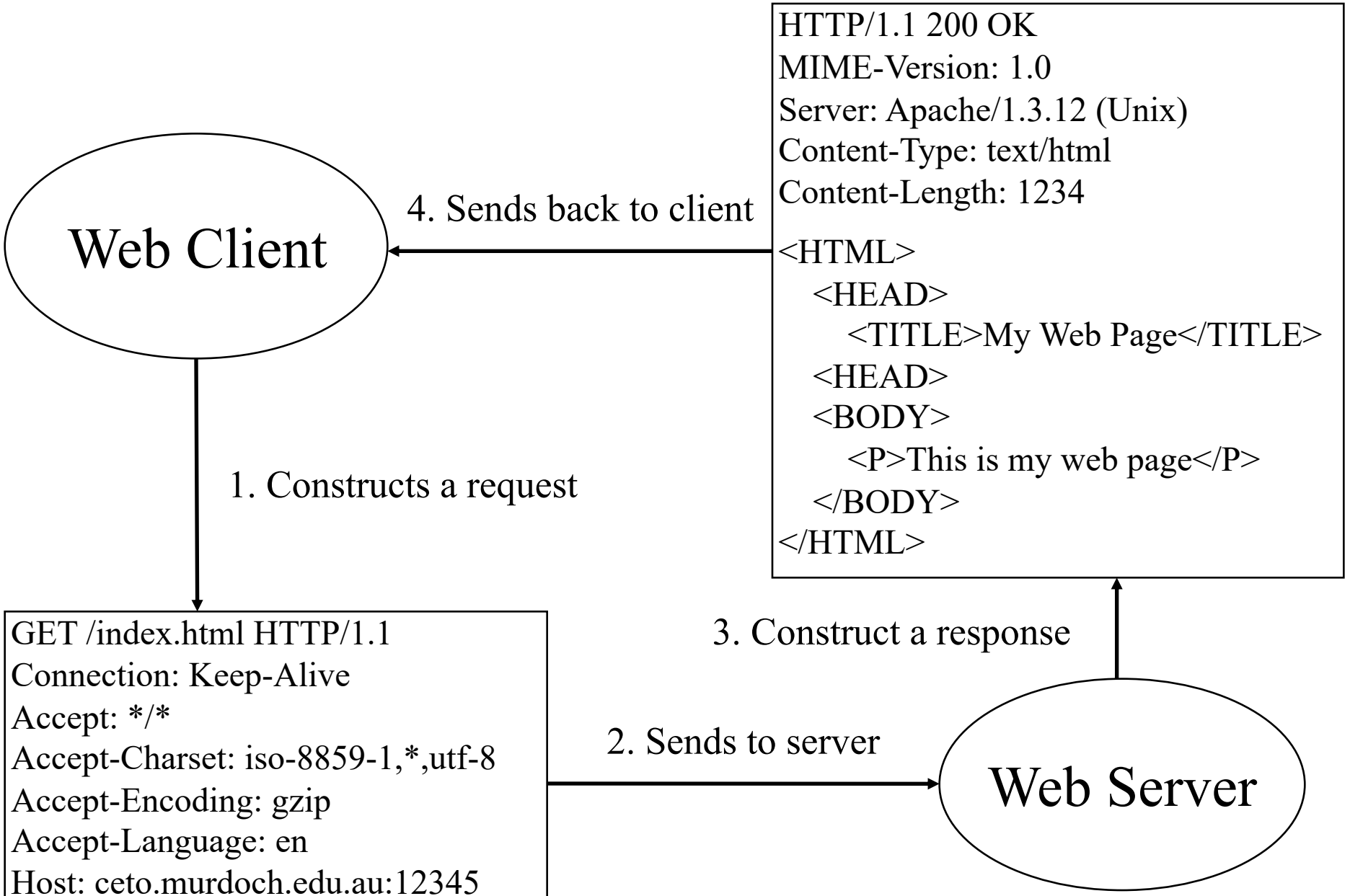
Lecture 3 (B)

Lecture Objectives

- Relevance to unit objectives:
  - Learning objectives 1, 2, and 3
- To introduce the basic requirements in implementing a web client
- To use Node.js to write a simple web client
- To prepare for Lab 3 and Lab 4

Lecture Outline

- Briefly review our basic Web Server from the previous lecture slides
- Use a Web browser and the Linux utility `curl` to test our basic Web Server
- Introduction to Node.js modules to implement a simple Web Client


HTTP/1.1 200 OK
MIME-Version: 1.0
Server: Apache/1.3.12 (Unix)
Content-Type: text/html
Content-Length: 1234
<HTML>
<HEAD>
<TITLE>My Web Page</TITLE>
<HEAD>
<BODY>
<P>This is my web page</P>
</BODY>
</HTML>
4. Sends back to client
Web Client
1. Constructs a request
GET /index.html HTTP/1.1
Connection: Keep-Alive
Accept: */*
Accept-Charset: iso-8859-1,*,utf-8
Accept-Encoding: gzip
Accept-Language: en
Host: ceto.murdoch.edu.au:12345
User-Agent: Mozilla/4.0
2. Sends to server
3. Construct a response
Web Server

Simple HTTP Web Server


```
// server.js – layout used for clarity
var http = require('http');   // import http module
function onRequest(req, res) {
  res.writeHead(200,{'Content-Type': 'text/plain'});
  res.write('hello client!');
  res.end();
}
var server = http.createServer(onRequest);
server.listen(8888,'127.0.0.1');
// output message to screen
console.log('Server running http://127.0.0.1:8888/');
console.log('Process ID:', process.pid);
```

Simple HTTP Web Server: Alternative Layout


```
// an alternative layout commonly used

var http = require('http');   // import http module
http.createServer( function (req, res) {
  res.writeHead(200,{'Content-Type': 'text/plain'});
  res.write('hello client!');
  res.end();
}).listen(8888,'127.0.0.1');
console.log('Server running http://127.0.0.1:8888/');
console.log('Process ID:', process.pid);
```

Starting The Server

- To run the server on command line:

```
-> node server.js
-> Server running at http://127.0.0.1:8888/
-> Process ID: 1234
```

Simplest HTTP Web Clients

- The two simplest clients to demonstrate the working of an HTTP server are:
  - The Linux utility `curl`
  - A web browser
- The next slide show how to use these two clients to test our http server
- **Firstly, ensure that the server is running**

Simplest HTTP Web Clients

- Using `curl` as the client on command line (in a different terminal):

```
-> curl http://127.0.0.1:8888
-> hello client!
```

- To display in a web browser, type the following in the browser url:

```
-> 127.0.0.1:8888
OR
-> localhost:8888
```

Task: A HTTP Web Client

- We want to implement a simple HTTP web client script in Node.js
- It must be able to communicate with HTTP servers using GET and HEAD methods and possibly other methods such as POST, OPTION, TRACE, etc.

Node.js HTTP Client Method

- The syntax of the method to create a HTTP client is:

```
http.request(options[,callback])
```

where `options` is an object to specify request header information, and `callback` is an optional callback function

Simple Web Client

- Our client script should do something very simple, which all web browsers do:
  - Fetch the resource specified by a URL
    - Send a HTTP request message
    - Wait for a response
    - Process the response when it arrives
  - It will use HTTP to do the transport
    - This means it will only be able to communicate with a HTTP server

Simple Web Client Algorithm

- The basic structure of the script (from our requirements on the pervious page) is:

1. LOAD MODULES – http module in this case

2. CREATE 'OPTIONS' OBJECT TO SET URL, HTTP METHOD, ETC.

3. DEFINE A CALLBACK FUNCTION TO HANDLE INCOMING RESPONSES, PASSING THE RESPONSE OBJECT AS A PARAMETER

    a) SET UP EVENT LISTENER WITH DATA + ANONYMOUS FUNCTION PARAMETERS
    b) CALL THE FUNCTION TO RECEIVE EACH DATA PACKET

1. CREATE REQUEST OBJECT, PASSING OPTIONS AND OUR CALLBACK FUNCTION AS PARAMETERS

Simple Web Client Script: `options` Object


```
// import http core module using require method
// assign return object to var http
var http = require('http');
// set options for client request with object literal
// here setting url, port, path, and method
var options = {
      host: 'ceto.murdoch.edu.au',
      port: 8888,
      path: '/',     // application root
      method: 'GET' // no comma
};
```

Script Explained

- Firstly, import the appropriate module (http)
  - Assign returned object to an instance variable
- Define the `options` object to set:
  - The hostname
  - The port number
  - The required resource (in this case `'/'`, which is the root of the application)
  - The HTTP method (in this case `GET`)

Simple Web Client Script: Callback Function


```
function onResponse(response){
     response.setEncoding('utf8');
     // event listener with 2 parameters
     response.on('data', // data event
       function(data){    // incoming data
          console.log(data);
       }  // end anonymous function
     );    // end event listener
} // end anonymous function
```

Script Explained

- Define the function to handle responses
  - We called this function **`onResponse()`**
  - The caller of the callback function would pass a `response` object as the parameter
  - It includes an event listener `response.on()`
  - The `response.on()` method takes two parameters:
    - A `'data'` event to accept incoming data packets
    - An anonymous callback function which allows for multiple packets of **data** to be received
    - Note that in our example each packet received is simply printed to screen

Script Explained

- Note: UTF stands for Unicode Transformation Format. The '8' means it uses 8-bit blocks to represent a character.
- UTF-8 is a compromise character encoding that can be as compact as ASCII (if the file is just plain English text) but can also contain any unicode characters (with some increase in file size).
- In the example, we use `response.setEncoding` to set the character encoding so that the incoming response body (data) will be returned as a string of the specified character encoding rather than as a buffer object.

Simple Web Client Script: Create Request


```
var client = http.request(options,onResponse);
client.end();  // end request method
```

Simple Web Client Script: Script Explained

- Next the `http.request()` method is called to create the *request* object
- The `http.request()` method returns an instance of `http.createClient` class, which is assigned to an instance variable
- The method takes two parameters:
  - The **`options`** object
  - The **`onResponse()`** function
- Finally, the request object **must** be closed using **`client.end();`**

Why Client Script?

- You would typically use a web browser to display developed web applications, so why use scripts like the previous one?
- Scripts are used for testing your server code whilst it is under development
  - In particular, you can get access to header information from both client and server
  - This can help with debugging, if the communication between them is not functioning as you expect

Accessing Response Headers

- The web client can access the header information returned by the server

```
var http = require('http');
var options = {
      method: 'HEAD',
      host: 'localhost',
      port: '8888'
};
http.request(options, function(response){
      console.log(response.headers);
}).end();
```

Response Header

- The `options` object literal includes the **HEAD** method
- The `request` method has two parameters:
  - The `options` object
  - An anonymous function with a `response` object as its parameter
  - The argument passed to `console.log` is the returned header information obtained by `response.headers`
  - The `end()` function **must** terminate the request

Response Header

- If the web client needs to process the header information, **querystring** module is needed

```
var http = require('http');
var qs = require('querystring');
var options = {
    method: 'HEAD',
    host: 'localhost',
    port: '8888'
};
http.request(options, function(response){
    console.log(qs.stringify(response.headers));
}).end();
```

Response Header

- The **querystring** module is imported and assigned to an instance variable **`qs`**
- The `options` object literal remains the same
- The `request` method again has two parameters:
  - The `options` object
  - An anonymous function with a `response` object as its parameter

Response Header

- The **`qs`** object is used to call the `stringify()` method, which converts the value of its argument
- The argument passed to `stringify()` is the returned header information obtained by `response.headers`

- **Note:** *the output from this request will be one long string that needs to be parsed to extract the relevant information*

Accessing Request Headers

- The web server can access the header information sent by the client

```
var http = require('http');   // import http module
http.createServer( function(request, response) {
  console.log(request.headers);
  response.writeHead(200,{'Content-Type': 'text/plain'});
  response.write('hello client!');
  response.end();
}).listen(8888,'127.0.0.1');
console.log('Server running at ');
console.log('http://127.0.0.1:8888/');
```

Request Header

- The only required change to the server code is the addition of the following line to the `createServer()` method

```
console.log(request.headers);
```

- The argument passed to the `console.log` is the returned header information obtained by `request.headers`
- **Note:** *the output will be sent to the ssh session window that started the server*

Request Header

- If the web server needs to process the header information, **querystring** is needed

```
var http = require('http');   // import http module
var qs = require('querystring');
http.createServer( function (request, response) {
  console.log(qs.stringify(request.headers));
  response.writeHead(200,{'Content-Type': 'text/plain'});
  response.write('hello client!');
  response.end();
}).listen(8888,'127.0.0.1');
console.log('Server running at http://127.0.0.1:8888/');
```

Request Header

- The **querystring** module is imported and assigned to an instance variable **`qs`**
  - The **`qs`** object is used to call `stringify()` method, which converts the value of its argument
  - The argument passed to `stringify()` is the returned header information obtained by `request.headers`
- **Note:** *the output from this response will be one long string that needs to be parsed to extract the relevant information*

Running the Server on Ceto

- When executing this server code on ceto.murdoch.edu.au, you **must** use the port number assigned to you; **not 80 or 8888**
- You may also wish to specify the IP number or hostname for ceto in the `listen()` method of the server code
- **And do not forget to kill the server when you have finished working**
  - The repetition of this message MUST be getting annoying for you; just think how the sysadmin feels when you don't kill your servers

Read the Scripts

- Study the scripts and analyze their operations line-by-line
- Check with JavaScript and Node.js documentation for any commands that you are unsure about
- In lab 3, you will test your server with the three client approaches discussed in these lecture notes

Acknowledgement

- The code snippets were sourced from:

  Basarat, A.S., Beginning Node.js

- The conceptual theory was derived from:

  Node.js website: https://nodejs.org/en/

  Basarat, A.S., Beginning Node.js

Application Development In Node.js: Preliminaries

Lecture 3 (C)

Recapitulation

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

Why Export?

- By exporting functionality from various parts of an application we can make it modular:
  - This makes the development process easier to control and we end up with a better design
  - It is also better for future development and maintenance
- So the point can be made here: it is expected that for your work in this unit, you will make your own applications modular
  - In assignment 1 **this is** an assessment factor

Where to Place the Server?

- So how can we use our server to develop an application AND where in an application do we place our server module?
- It is common practice to have a main file called `index.js` which is used to start an application by making use of the other modules of the application
- Thus the server module can be called from `index.js`

How to Organize The Application?

- However, we will come back to this later …
- Firstly, in order to have our application accept requests from various client sources (and react accordingly), our server needs to redirect program flow to different parts of our application code to satisfy different HTTP client requests
- This is called '**routing**'

Routing Requests

- We need to be able to route requests by deciding upon the most appropriate code to execute according to the request
  - The code to execute the different requests will be a collection of **request handlers** that do the actual work when a request is received
- So to determine where to route to, we need to look at the URL and extract information from the HTTP requests

Routing Requests

- All the information we need to process the request is available through the `request` object, which is passed as the first parameter to our **`onRequest()`** callback function (or the anonymous function)
- We can utilize some additional Node.js modules to process the incoming requests
  - Namely, '**`url`**' and '**`querystring`**'

URL and Querystring Modules

- The **`url`** module provides methods which allow us to extract the different parts of a URL; this includes the **`path`** and the **`query`**
  - `Path` is where the required resource is located within the file system that resides on the server
  - `Query` may be the actual required resource
- The **`querystring`** module can be used to parse the `query` part of the URL request parameters

Examples

Example url assuming last lecture's server is running:

```
http://localhost:8888/startServer?foo=bar&hello=world
```

```
var path=url.parse(string).pathname;  -> /startServer
var qstr=url.parse(string).query;     -> foo=bar&hello=world
querystring.parse(qstr)["foo"];       -> bar
querystring.parse(qstr)["hello"];     -> world
```

where the **`string`** to be parsed is the `request.url`

- Look up the online documentation for these core modules to learn how to obtain the various parts of the **`url`** and other usages for **`querystring`**

Routing Requests

- The application will need be able to distinguish between requests based on the URL path requested
- This will allow the mapping of requests to request handlers based on the URL path
- So, let us now add to our `onRequest()` (or anonymous function), the logic needed to find the URL path the client has requested

server.js Script

```
var http = require("http");   // import http core modules
var url = require("url");     // import url core modules
http.createServer( function (request, response) {
  // use url module to get pathname of requested resource
  var pathname = url.parse(request.url).pathname;
  console.log("Request for " + pathname + " received.");
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}).listen(8888);
console.log("Server has started.");
```

Exporting The Server

- Now let us export the server so that other scripts are able to utilize our web server
  - Remember, we only need to export that specific functionality associated with starting the server
  - We do this by encapsulating the required functionality in a function

Exporting The Server

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

Create `router.js` Script

- To keep with our modular design approach, let us now create a new file called `router.js`, with the following content:

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

`router.js` Script

- Note that the **`route()`** function takes the **`pathname`** as its parameter
  - At this stage we just print the **`pathname`**
- Also note that we have exported the **`route()`** function

Re-Factor `server.js`

- To use the **`route`** module, we need to re-factor `server.js`
  - Firstly, we pass the `route()` function as a parameter to `startServer()`
  - We then enter a new line of code in the server script, to call the `route()` function with its argument - the `pathname`
  - The `pathname` is obtained in the server code

Re-Factor server.js

```
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;
```

`index.js` Script

- We mentioned earlier that it is typical to use a script (`index.js`) to start and control the modules of an application
- To use the functionality provided by both of our modules, we import both modules into a new script called `index.js`
  - As `startServer()` and `route()` functions were exported, we can import the appropriate modules and include them in the same manner that we do for the core modules

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);
```

`index.js` Script

- Note that the `require` directive uses the filename (minus the file extension) to import the modules
  - The preceding characters to the filenames (`'./'`) indicate that both scripts are located in the current working directory (i.e., the same directory as the `index.js` script)

`index.js` Script

- The instance variables assigned the imported objects are then able to access the functions provided by our modules
- So we call `server.startServer()` and pass the parameter `router.route`
  - Note that we passed the function `route()` not the object `router` (neither the result of function invocation)
  - i.e., the router object is used to access the function, and it is the function that is passed, not the object

Testing `index.js`

- To test in a terminal, run on command line:

```
node index.js
```

- In another terminal, run on command line:

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

- Output should be like this:

```
Server has started.
Request for / received.
Routing a request for /
```

  - Note: / refers to the root of the server application, not the root of the filesystem of the machine the server resides on

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 Labs 3 and 4 and Assignment 1
- Check with JavaScript and Node.js for any code that you are unsure about

Acknowledgement

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