ECS 189
WEB PROGRAMMING

5/8

Project idea

- Photo sorter
- User uploads photos
- Server uses Google Cloud Vision API to generate tags for images
- Server stores images and tags in a database
- Server gives user menu of tags, allows them to retrieve photos from database by tag

Simple server from last time

```
function handler (request, response) {
    var url = request.url;
    response.writeHead(200, {"Content-Type": "text/html"});
    response.write("<h1>Hello!</h1>");
    response.write("<p>You asked for <code>" + url +
                                        "<code></p>");
    response.end();
}
var server = http.createServer(handler);
server.listen(*your*port*number*);
```

Request and response objects

- Like a Netflix envelope you get in the mail
- The request object is the disk; it has the data in it
- The response object is the envelope itself; you put what you're sending back into it
- response.end() "drops it in the mailbox"

Handling different urls

- Our server will need to do different things given different URLs
- Recognize dynamic URLs (eg. add something to database), send them to dynamic handler to do something on server, make up an AJAX response, etc.
- Or, recognize static Web pages that match the URL
- Or, respond with "404 not found" in the header
- The idea of sending different urls to different sub-handlers is called routing.

Static URLs

- Include just a pathname, eg:

www.cs.ucdavis.edu/~amenta/s17/ecs189h.html

- There is an actual file on the server called ecs189h.html, which gets sent in the body of the response object (server code "puts it into the envelope")
- CSS and Javascript files typically come from the static server as well

Dynamic URLS

- Recall the complex URL we used to request data from the Yahoo weather API
- What was the Yahoo server doing with this?

https://query.yahooapis.com/v1/public/yql?q=
select * from weather.forecast where
woeid =2389646 & format=json &
callback=callbackFunction

Dynamic URLS

https://query.yahooapis.com/v1/public/yql?q=
select * from weather.forecast where
woeid =2389646 & format=json &
callback=callbackFunction

- There's no file named this. There is not even a file named yql.
- The server code deciphers the URL, figures out what the database call ought to be, gets the data, turns it into a callback-function call, and finally stuffs that into the response "envelope"

Dynamic URLS

https://query.yahooapis.com/v1/public/yql?q=
select * from weather.forecast where
woeid =2389646 & format=json &
callback=callbackFunction

- Typical format:
  - name of server operation or API function
  - ? separating API name and specific request
  - parts of request, separated by &
  - Not required – servers can accept whatever format they care to - but this is very common

Where is the URL?

- Where do we find the URL?

Where is the URL?

- Where do we find the URL?
- In the "request" object, specifically "request.url"
- Node.js has a url module we can use to parse the URL (break it up into its parts, using this "?" and "&" format)

Parsing the URL

```
var urlStr = request.url;          // a string
var urlList= urlStr.split("?");
var pathname = urlList[0];
var query = urlList[1];
```

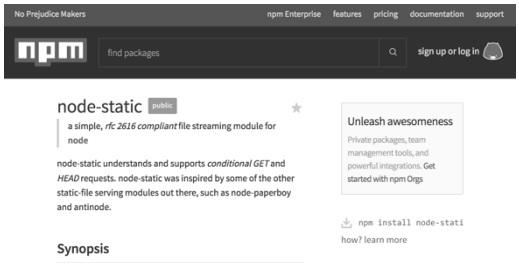
- For instance, if urlStr is "/photoSorter/sorter.css", then urlObj.pathname will also be "/photoSorter/sorter.css".
- query would be undefined

More interesting with dynamic URL

- So say
  urlStr contains “hello.html?dog=rover&cat=max”
- Then
  pathname contains “hello.html”
  query contains “dog=rover&cat=max”

Handler idea

- If there is a query string, we’ll need to collect data and/or create a Web page and pass it back in the response
- If no query string, assume it is a static request, pass back a file
- Fortunately, there are a bunch of static server modules out there that handle that part; we’ll use one
- But we need to understand some tricky stuff before we can use these modules



NPM

Node Package Manager (despite the joke in upper left)

- Repository for many, many node modules that other people wrote
- Varying quality
- The “require” (Javascript include) won’t work until we install the modules in our account
- Do this on the Unix command line, eg:
  npm install node-static

From the documentation

```
var static = require('node-static');
//
// Create a node-static server instance
// to serve the './public' folder
//
var file = new static.Server('./public');
```

- This creates an object that serves static files, from a subdirectory called /public.
- We need to embed this in a Web server.

Copy to blackboard!!!!!

```
require('http').createServer(function (request, response
  ) {
   request.addListener('end', function () {
       // Serve files!
       file.serve(request, response);
   }).resume();
}).listen(8080);
```

- Not much here we have not seen before but combined in a tricky way.

Using an object

require('http').createServer(….

- What's going on here? Usually we see "require" at the top of the file, like an include statement in C:

var http = require('http');

Using an object not in a variable

require('http').createServer(….

- What's going on here?
- require('http') returns an object containing the data and methods of the http module. Without putting it into a variable, we call its createServer method.
- We could call this an anonymous object.

The end

…createServer(….).listen(8080);

- Similar situation. The http method createServer() returns a server object, which has a listen method to listen to a port.
- Again, we use this object anonymously
- In this case the port number is 8080, but you'd use your own

Anonymous functions

- A Javascript language feature we have not used yet.
- Often found in situations where we want to use a function as a parameter, ie. from our simple Web server:

```
function handler (request, response) {
  …
  response.end();  }
var server = http.createServer(handler);
```

Anonymous function

…createServer(function (request, response) {
  …}).listen(8080);

- Here we're not bothering to give the request handler a name; we're just defining it inside the parentheses.
- Recall the alternative function def syntax:

var f = function(x,y) { return x+y; }

- Right-hand side is an expression that returns a function

Inside handler function

```
request.addListener('end', function () {
      // Serve files!
      file.serve(request, response);
  }).resume();
```

- Recall that file was our static file server object, and it seems to have a handy method serve, that takes the request and response and…does what?

Inside handler function

```
request.addListener('end', function () {
        // Serve files!
        file.serve(request, response);
    }).resume();
```

- Recall that file was our static file server object, and it seems to have a handy method serve, that takes the request and response and…does what?
- Puts the file from /public that was requested into the body of the response, hopefully, and then calls

```
response.end();
```

Inside handler function

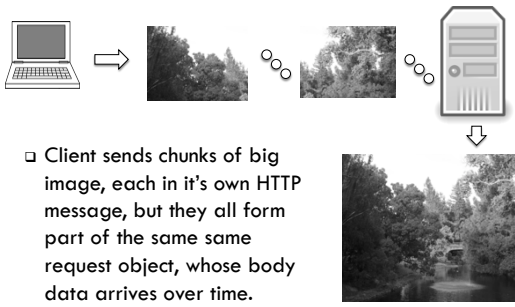
```
request.addListener('end', function () {
        // Serve files!
        file.serve(request, response);
    }).resume();
```

- But when is file.serve() getting called? A bit complex, but understanding this will be handy later.
- Turns out a request object is a data stream, meaning a source from which a whole bunch of data can be read, for instance like a file, or stdin in C.

Data stream

- Why would we want to read a lot of data from an HTTP request?
- Recall the request is any message coming from client to server. In our photo sorter, we'll be sending photos. These are big files.
- If asked to, HTTP chunks big files into a series of smaller messages that traverse the internet (aka packets).

Data stream



- Client sends chunks of big image, each in it's own HTTP message, but they all form part of the same same request object, whose body data arrives over time.

Inside handler function

```
request.addListener('end', function () {
        // Serve files!
        file.serve(request, response);
    }).resume();
```

- request.addListener returns the request object again; this is common in Node.
- Calling request.resume(), a the end, starts getting data from the data stream.

Event listeners

- Data streams in Node use callback functions, just like everything else. Here, we specify a callback function for when the data stream gets to its end and all the data is here:

```
addEventListener('end', function () {
    // Serve files!
    file.serve(request, response);  })
```

- What is the callback function named?

Event listeners

- Data streams in Node use callback functions, just like everything else. Here, we specify a callback function for when the data stream gets to its end and all the data is here:

```
addEventListener('end', function () {
    // Serve files!
    file.serve(request, response);  })
```

- What is the callback function named?
- Trick question! It is an anonymous function again.

Inside handler function

```
request.addListener('end', function () {
    // Serve files!
    file.serve(request, response);  }).resume();
```

- So file.serve(), which actually serves the static file, gets called when the request data stream receives its end event, by the anonymous end callback.
- This is well after request.addListener, and the anonymous handler function, have exited.
- But the values of request and response are still correct. Why?

Inside handler function

```
request.addListener('end', function () {
    // Serve files!
    file.serve(request, response);  }).resume();
```

- The closure of file.serve() is the anonymous function inside request.addListener
- And the closure of that function is the anonymous handler function inside createServer
- So file.serve has permanent access to the values of their local variables when file.serve was created.

Homework

- Write a server that combines the easy query server we wrote with the static query handler defined by node-static.
- Recall "undefined" means there is no such property in the urlObj – eg. if query is undefined, try static.
- Add a "404 not found" message; see the documentation for node-static to see how to do that.