ECS 162
Web Programming

4/29

Assignment 4

- Set up and test out server.
- Server has to:
  - Serve static files (http, css, js)
  - Respond to AJAX queries, providing JSON
- Today, we make a server that does all these things
- By Thursday, you will try it out, give it some files to serve, and make it answer a specific query.

miniServer2.js from last time

```
const express = require('express');
const port = // put your port number here
function handler(req, res) {
    let url = req.url;
    res.send('You requested '+url);
}
const app = express();
app.get('/*', handler );
app.listen(port, function () {console.log('Listening...');} );
```

Request and response objects

- Like the Netflix envelopes we used to get in the mail
- The request object is the DVD; it has the data in it
- The response object is the envelope itself; you put what you're sending back into it
- res.send() "drops it in the mailbox"

- What is the actual internet traffic that this metaphor corresponds to?

HTTP request

Head

Body

(body often empty)

HTTP response

Head

Body

Static URLs

- Include just a pathname, eg: on the UCD CS server:

www.cs.ucdavis.edu/~amenta/s19/ecs162.html

- There is an actual file on the server (here ecs162.html), which gets sent in the body of the HTTP response (server code “puts it into the envelope”).

Dynamic URLS

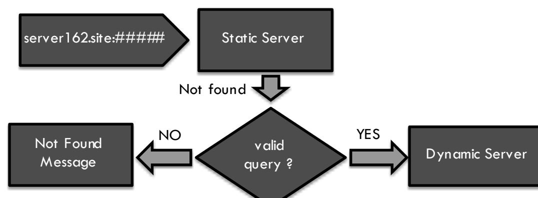
- The URL we used with the OpenWeatherMap API requested the server to get something out of a database, format it, and make JSON; this is dynamic

http://api.openweathermap.org/data/2.5/forecast/hourly?q=Davis,US&units=imperial&APPID=xxx

- In this case, server162.site handles both static and dynamic HTTP requests.

Handling different urls

- The idea of sending different urls to different sub-handlers is called routing.



- Exactly one of the rectangular boxes returns the response.

Static server

- Since we are using express, we will use its static server module.
- We get node modules through npm, the Node Package Manager (despite joke name upper left)

NPM

- Repository for many, many node modules
- Varying quality, probably many viruses, etc. Look for well-known, open source modules
- The require statement that "includes" a module gives an error message until we install the module
- Do this on the Unix command line, eg:

    npm install node-static
- Creates files in subdir node_modules

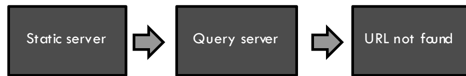
The server

- Main function is just these five lines of express.
- Top line makes object; last hangs waiting for HTTP requests.

```
const app = express();
app.use(express.static('public'));
app.get('/query', queryHandler );
app.use( fileNotFound );
app.listen(port, function (){console.log('Listening...');} );
```

Middleware

- Server control flow is a pipeline of "middleware functions"
- Ours will be pretty simple



- A middleware function either calls res.send() or it calls a special function called "next", which moves on to the next pipeline stage.

Routing defines a virtual directory structure

Handler as middleware

function queryHandler(req, res, next) {
… }

- Takes request object, response object, and next function as input.
- Trys to figure out response to request. If it can, fills in the response object and calls res.send(). The end.
- If it can't, calls next.
- HAS TO HAVE this structure, otherwise messes up pipeline.

Building the pipeline

- We build an express pipeline by adding middleware functions using pipeline constructor methods such as:
  app.use(), app.all(), app.get(), app.post()
- Each of these takes an optional path as its first argument, which controls which HTTP requests the middleware gets applied to.
- The second (and maybe more) arguments are middleware functions, which go into the pipeline in order.

The pipeline

- Order of functions in pipeline is order in which they were inserted.

Constructor functions

- app.get(), app.post() - The middleware it adds only gets applied to HTTP GET or POST requests, respectively. The url is required and has to exactly match (but regular expressions allows * (all) or ? (either), etc).
- app.all() – any kind of HTTP request, but url rules as above.
- app.use() - applies it to anything ***beginning with*** the path, and to everything if the path is not specified. Usually at least the "not found" handler applies to everything.

Confusing

app.enabled()
app.engine()
app.get()
app.get()
app.listen()
app.METHOD()
app.param()
app.path()
app.post()

- app has two “get” methods, one for getting its properties and the other for adding middleware that only applies to get requests.

Queries

- The ? in a query signals the end of the path and the beginning of the query
- Queries are key-value pairs, separated by &

  q=Davis,US&units=imperial&APPID=xxx

  animal=bear

  word=malapropism
- rec.query contains the query as an object.

Returning JSON

- Most of our queries will return JSON.
- The response object has a method that takes an object, stringifies it, puts it in the body of the HTTP response, and then sends the response:

```
res.json( {"beast" : qObj.animal} );
```

- You don’t need res.send() when you call res.json().

Homework

- Change the query so that it takes a word as input and returns the palindrome.
  - Input: word=malapropism in query string
  - Output: ‘{ “palindrome” : “malapropismmsiporpalam” }’
- Then make a little app that exercises this AJAX request-response. I gave you the html (you can make it better, and add css, if you want). You need to add the Javascript.
- Javascript should include an onclick function that sends the HTTP request, and the callback function that gets run when the response gets back.

HTTP request from Assn 3 (CORS)

```
let url = "http://api.openweathermap.org/data…"
let xhr = new XMLHttpRequest();
xhr.open(method, url, true);
xhr.onload = function() {…};
xhr.onerror = function() {…};
xhr.send();
```

HTTP request for Assn 4 (AJAX)

```
let url = “query?animal=bat"
let xhr = new XMLHttpRequest();
xhr.open(method, url, true);
xhr.onload = function() {…};
xhr.onerror = function() {…};
xhr.send();
```

- URL does not contain name of server (domain name and port). By default, it goes back to the server the Web page came from.