IN4MATX 133: User Interface Software

Lecture 11:
Server-side development

Professor Daniel A. Epstein
TA Jamshir Goorabian
TA Simion Padurean

Announcements

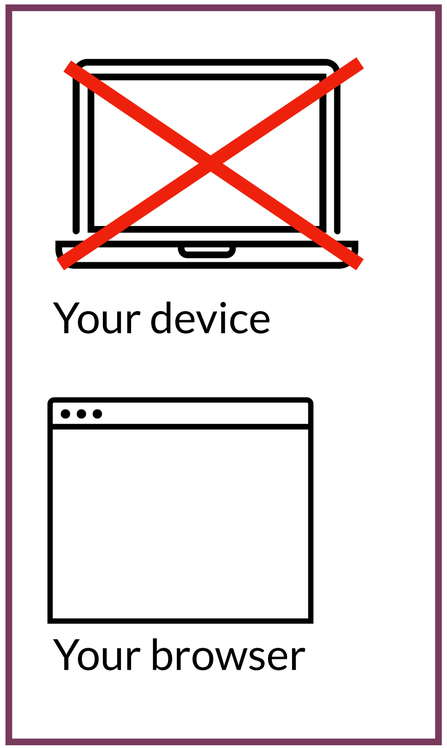
- A2 due ~~Thursday 4am~~ -> Saturday 4am
- A3 will be posted “soon”, probably before new A2 deadline
- Quiz 2 tomorrow
  - Bring a (dark color) pencil/pen!
- On Slack (and email), message the channel or all of the course staff
  - I’m getting more direct messages than I can handle

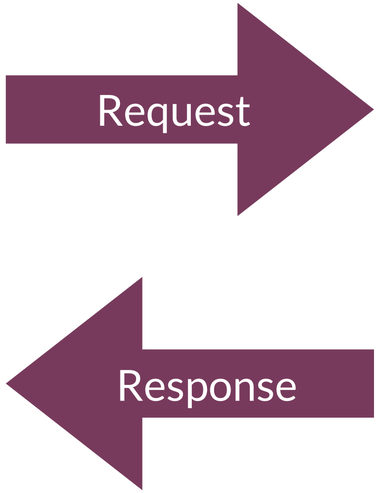
Today's goals

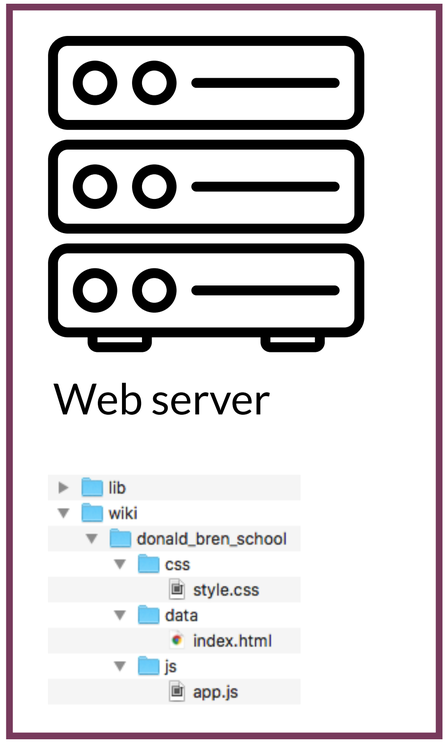
By the end of today, you should be able to…

- Explain the advantages and disadvantages of different tools for server-side development
- Develop a server with Express which can route to multiple endpoints, parse data sent, and respond appropriately
- Leverage templating engines to format simple pages

Client-side and server-side JavaScript







Edit what's being rendered
Trigger or react to events

Navigate file system programmatically
Dynamically generate pages or views
Transport, store, or interact with data

Client-side

- Runs in the browser
- Changes happen in real-time in the browser
- Cannot make HTTP requests to many APIs
- Examples: AJAX, Angular, React, Vue.js

Server-side

- Runs in the command line, etc. (but maybe can still be accessed from the browser)
- Changes happen in response to HTTP requests
- Can make HTTP requests to most APIs
- Examples: Node, ASP.NET

Server-side in this course: Node.js

- Event-driven, non-blocking I/O model makes it efficient
- Best for highly-interactive pages
  - When a lot of computation is required, other frameworks are better
  - Event-driven loops are inefficient
- Lower threshold for us: we're already learning JavaScript!

Other server-side environments

- Ruby, via Ruby on Rails
- Python, via Django or web2py
- These days, you can create a dynamic website in almost any language

Node package manager (npm)

- Included in the download of Node
- Originally libraries specifically for Node
- Now includes many JavaScript packages

Node.js hello world

```
var http = require('http');
```
← Require the http library

Node.js hello world

```
var http = require('http');
var server = http.createServer(function(req, res) {


});
```

Require the http library

Anonymous function with request and response parameters

Node.js hello world

```
var http = require('http');
var server = http.createServer(function(req, res) {
 res.writeHead(200);
 res.end('Hello World');
});
```

Require the http library

Anonymous function with request and response parameters

"Ok" status in the header, write hello world text

Node.js hello world

```
var http = require('http');
var server = http.createServer(function(req, res) {
  res.writeHead(200);
  res.end('Hello World');
});
server.listen(8080);
```

Require the http library

Anonymous function with request and response parameters

"Ok" status in the header, write hello world text

Listen on port 8080

Running Node.js

- `node file.js`
- Or if you’ve defined a start script in your `package.json` file, `npm start`

```
…”scripts”: {
    "start": "node file.js”
  }
…
```

Remember,
Node.js is server-side JavaScript

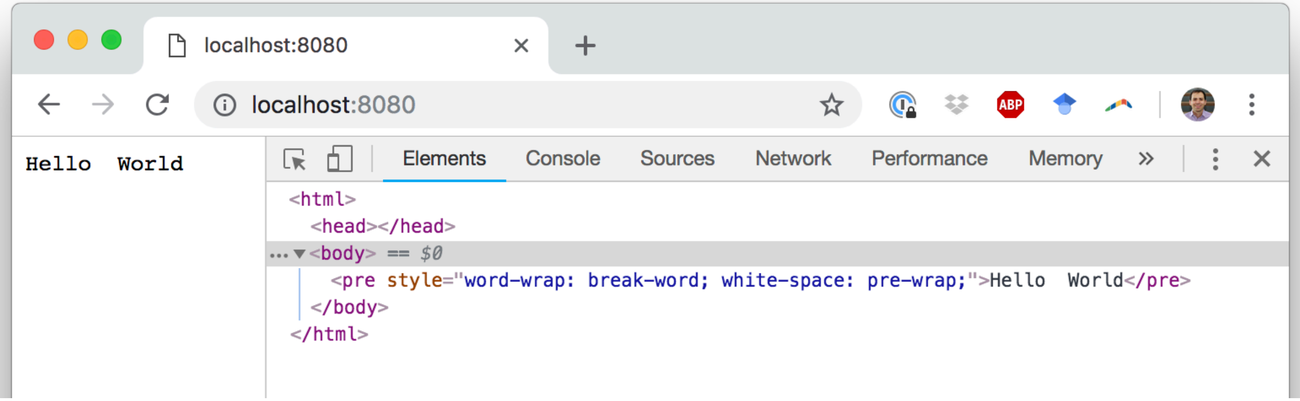
Where is the JavaScript running?

Server-side

```
node hello.js
```

hello.js:

```
var http = require('http');
var server = http.createServer(function(req, res) {
  res.writeHead(200);
  res.end('Hello World');
});
server.listen(8080);
console.log('Hello, console');
```



Node is listening on port 8080.
But the JavaScript is **not** running in the browser.
It's running in the console.

Where is the JavaScript running?

Client-side

```
live-server
```

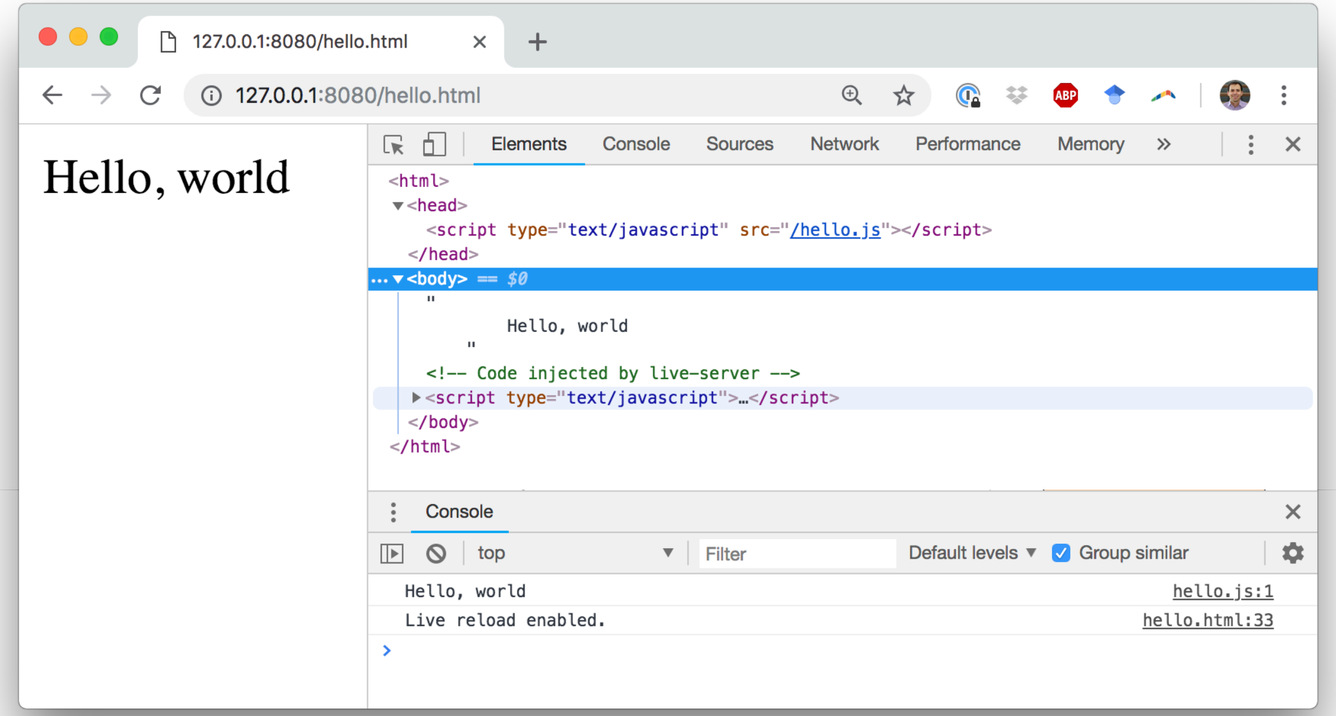
hello.html:

```
<html>
  <head>
    <script src="./hello.js"></script>
  </head>
  <body>
    Hello, world
  </body>
</html>
```

Live-server is listening on port 8080. The JavaScript is running in the browser.

hello.js:

```
console.log('Hello, world');
```



Question

Which can make an HTTP request to the Spotify API?

(Assume the browser uses default settings)

(1) A browser open to spotify.com

(2) A browser with client-side JavaScript at localhost:8888

(3) A browser with server-side JavaScript at localhost:8888

(4) A server running in the Spotify domain

A 4

B 1, 4

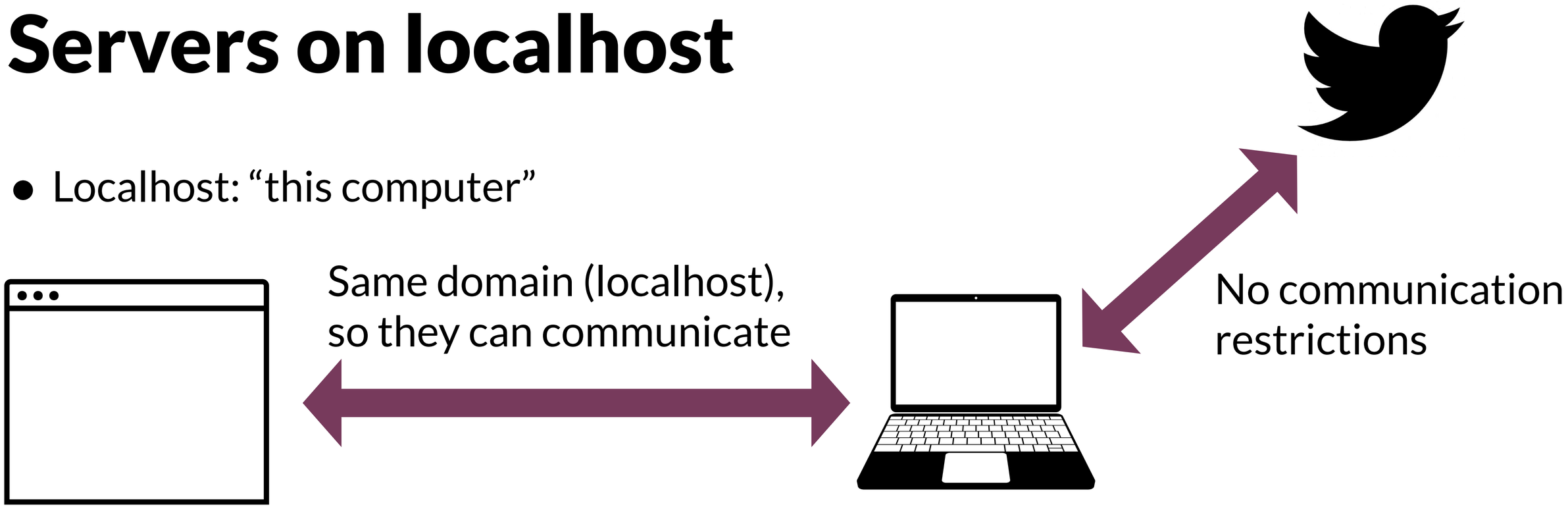
C 1, 2, 4

D 1, 3, 4

E 1, 2, 3, 4

Servers on localhost

- Localhost: “this computer”



Live server: localhost:8080

Browser implements same-origin policy to protect the other data you have open in the browser

Twitter proxy: localhost:7890

No same-origin policy restrictions, can communicate with Twitter

Node file system

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

Require the file system library

```
fs.readFile("/path/to/file", function(err, data) {
 console.log(data);
});
```

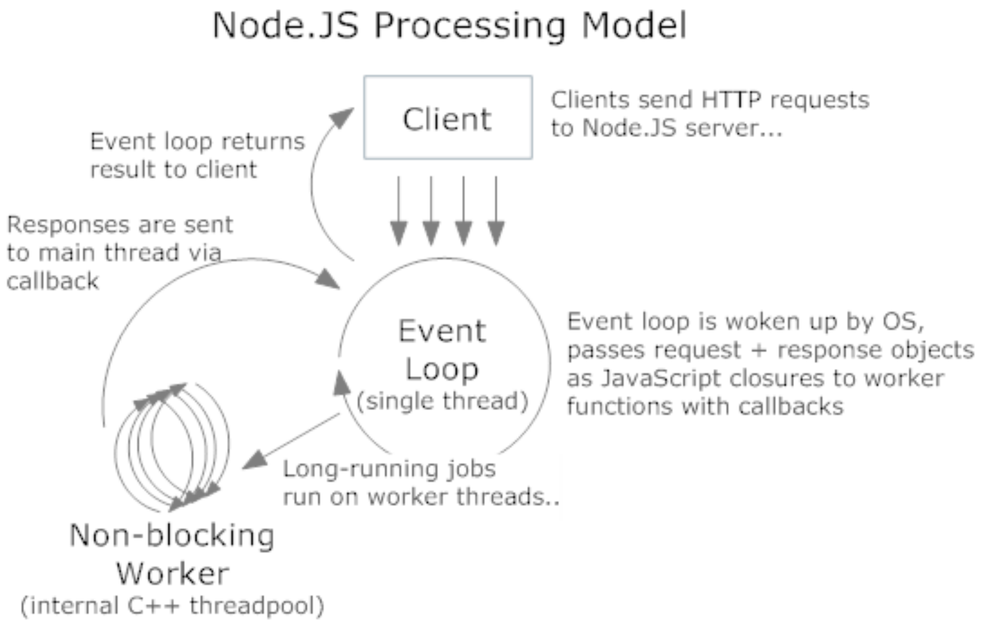
Read file, wait for asynchronous response

Node file system

```
var http = require('http');
var fs = require('fs');
var server = http.createServer(function(req, res) {
fs.readFile(__dirname + req.url, function (err,data) {
    if (err) {
      res.writeHead(404);
      res.end(JSON.stringify(err));
      return;
    }
    res.writeHead(200);
    res.end(data);
  });
});
server.listen(8080);
```

Node processing model

- Requests are handled in a single-threaded event loop
  - Every time someone loads a page node manages, it's added to this loop
- Requests are then processed asynchronously
  - When the work a request asks for is done, responses are returned to the client



Node modules

- Some modules are built-in

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

- Others can be installed via npm
- Added to your `node_modules` folder

What does Node.js add?

- OS-level functionality like reading and writing files
- Tools for importing and managing packages
- The ability to listen on a port as a web server
- But it’s just JavaScript, and it’s pretty basic as a web framework

What does a “good” server-side web framework need?

- To speak in HTTP
  - Accept connections, handle requests, send replies
- Routing
  - Map URLs to the webserver function for that URL
- Middleware support
  - Add data processing layers
  - Make it easy to add support for user sessions, security, compression, etc.
- Node.js has these, but they’re somewhat difficult to use

Express.js

- A fairly minimal web framework that improves Node.js functionality
  - Can route HTTP requests, render HTML, and configure middleware

```
var expressApp = express();

expressApp.get('/', function (httpRequest, httpResponse)
{
 httpResponse.send('hello world');
});
expressApp.listen(3000);
```

Express installation

- `npm install express`
  - Will save it to your node_modules folder

Express routing

- By HTTP method

```
expressApp.get(urlPath, requestProcessFunction);
expressApp.post(urlPath, requestProcessFunction);
expressApp.put(urlPath, requestProcessFunction);
expressApp.delete(urlPath, requestProcessFunction);
expressApp.all(urlPath, requestProcessFunction);
```

- urlPath may contain parameters (e.g., `'/user/:user_id'`)

httpRequest object

```
expressApp.get('/user/:user_id', function (httpRequest, httpResponse) …
```

- Has a lot of properties
  - Middleware can add properties
  - `request.params`: object containing url route params (e.g., user_id)
  - `request.query`: object containing query params (e.g., &foo=9 => {foo: '9'})
  - `request.body`: object containing the parsed body (e.g., if a JSON object was sent)

httpResponse object

```
expressApp.get('/user/:user_id', function (httpRequest, httpResponse) …
```

- Has a lot of methods for setting HTTP response fields
  - `response.write(content)`: build up the response body with content
  - `response.status(code)`: set the HTTP status code for the reply
  - `response.end()`: end the request by responding to it (the only actual response!)
  - `response.send(content)`: write content and then end
- Methods should be chained

```
response.status(code).write(content1).write(content2).end();
```

Middleware

- Give other software the ability to manipulate requests

```
expressApp.all(urlPath, function (request, response,
next) {
 // Do whatever processing on request (or setting
response)
 next(); // pass control to the next handler
 });
```

Middleware

- Middleware examples:
  - Check to see if a user is logged in, otherwise send error response and don't call `next()`
  - Parse the request body as JSON and attach the object to request.body and call `next()`
  - Session and cookie management, compression, encryption, etc.

Example Express server

```
var express = require('express');
var app = express(); // Creating an Express "App"
app.use(express.static(__dirname)); // Adding middleware
app.get('/', function (request, response) { // A simple request
handler
 response.send('Simple web server of files from ' + __dirname);
});
app.listen(3000, function () { // Start Express on the requests
 console.log('Listening at http://localhost:3000 exporting the
directory ' +
 __dirname);
});
```

Example Express user list

```
app.get('/students/list', function (request, response) {
 response.status(200).send(in4matx133.enrolledStudents());
 return;
});
app.get('/students/:id', function (request, response) {
 var id = request.params.id;
 var user = in4matx133.isEnrolled(id);
 if (user === null) {
 console.log('Student with _id:' + id + ' not found.');
 response.status(400).send('Not found');
 return;
 }
response.status(200).send(user);
 return;
});
```

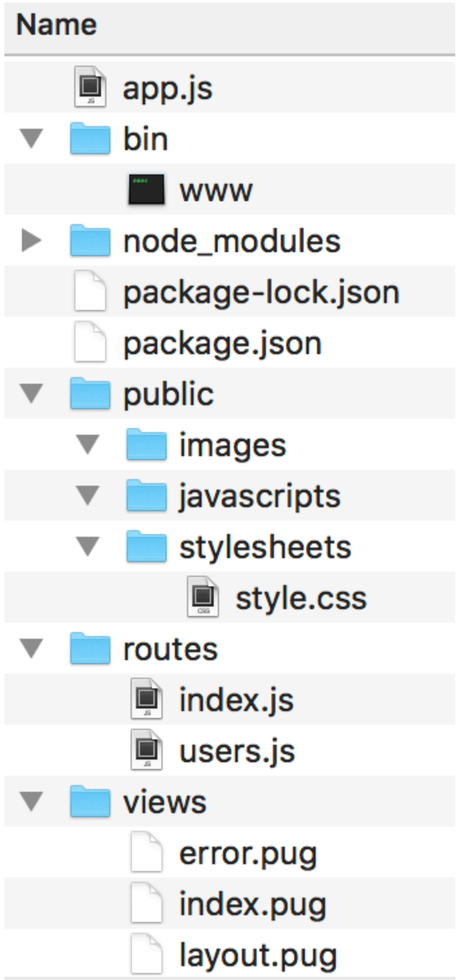
Express generator

- Express provides a tool that can create and initialize an application skeleton
  - Sets up a directory structure for isolating different components
  - Your app doesn't have to be built this way, but it's a useful starting point

https://expressjs.com/en/starter/generator.html

Express generator

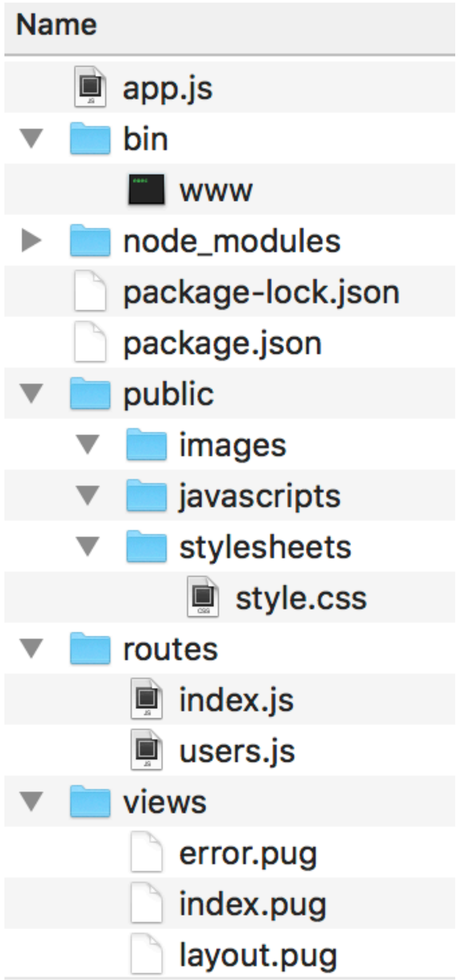
- `npm install express-generator -g`
- Can be invoked on command line with `express`
- Adds some boilerplate code
  and commonly used dependencies
- Install dependencies with `npm install`
  - `cd` into project directory first
- Run with `npm start`



https://expressjs.com/en/starter/generator.html

Express generator

- `package.json`, `package-lock.json`, and `node_modules` folder: library management and installed libraries
- `public` folder: all public-facing images, stylesheets, and JavaScript files



Express generator

- Routes folder: files which handle your URL mappings

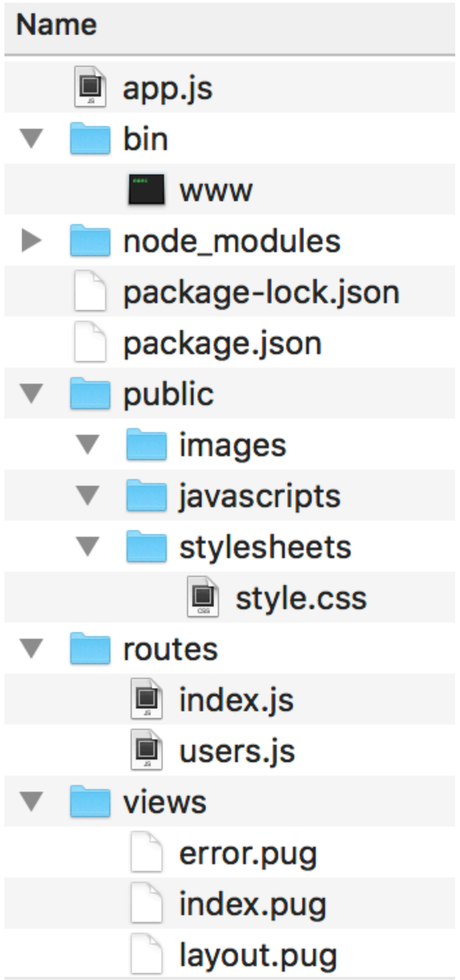
```
var express = require('express');
var router = express.Router();

/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});

module.exports = router;
```

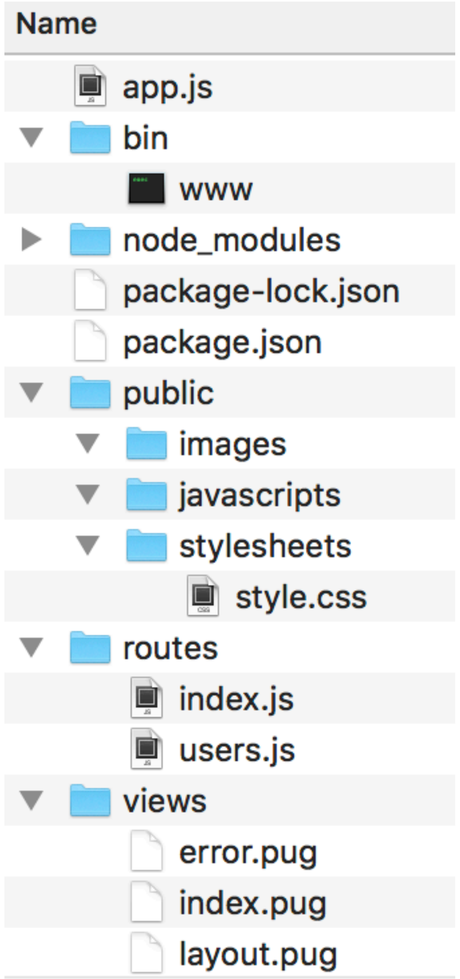
Variable passed to renderer

So another page can import your router



Express generator

- Views folder: any webpages which need to be rendered
- Uses a *view engine*, Pug, which generates HTML



Pug view engine

- layout.pug

```
doctype html
html
  head
    title= title
    link(rel='stylesheet', href='/
stylesheets/style.css')
  body
    block content
```

- index.pug

```
extends layout
```

← Imports other file

```
block content
  h1= title
  p Welcome to #{title}
```

← Parses variable passed

```
<!DOCTYPE html>
<html>
  <head>
    <title>Express</title>
    <link rel="stylesheet" href="/
stylesheets/style.css">
  </head>
  <body>
    <h1>Express</h1>
    <p>Welcome to Express</p>
  </body>
</html>
```

https://pugjs.org/api/getting-started.html

Express generator

- app.js: sets up middleware, routers, etc.

```
var indexRouter = require('./routes/index');
var usersRouter = require('./routes/users');

var app = express();

app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', indexRouter);
app.use('/users', usersRouter);
```

Middleware

Import route files

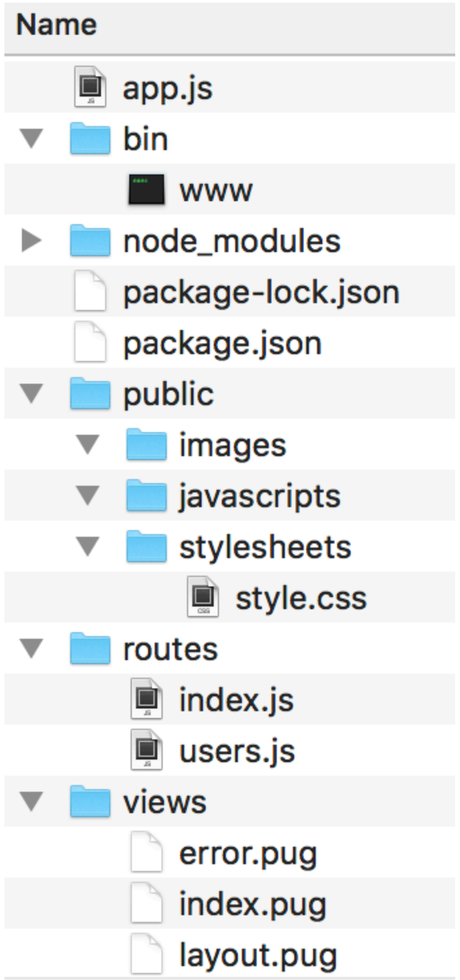
To parse content as json

To encode URLs

To handle cookies (user state)

To treat the public folder as static content

Use route files



Express generator

- bin/www: set up what port to listen on
- File that is run with `npm start`

```
var app = require('../app');
var http = require('http');

var port = normalizePort(process.env.PORT || '3000');
app.set('port', port);
var server = http.createServer(app);

server.listen(port);
server.on('error', onError);
server.on('listening', onListening);
```

Today's goals

By the end of today, you should be able to…

- Explain how programs access web resources
  and common ways they respond
- Implement a fetch request to get a resource from a web API
- Use promises to make an asynchronous request

IN4MATX 133: User Interface Software

Lecture 11:
Server-side development

Professor Daniel A. Epstein
TA Jamshir Goorabian
TA Simion Padurean