Web Applications

CSE183

Fall 2020

Node.js & Express

Baskin Engineering
UC SANTA CRUZ

Notices

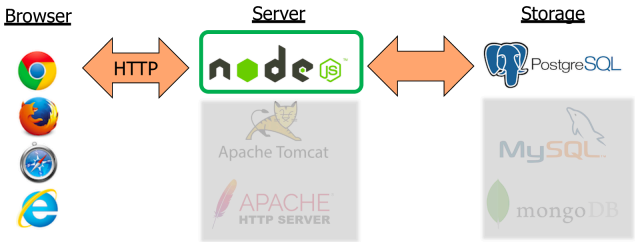
- **In-Class Quiz 3** at 09:25 **Monday November 16**
- **Assignment 7** due 23:59 **Thursday November 19**
- **Assignment 8** available 08:00 **Friday November 20**
  - **Due 23:59 Saturday November 28**
    - **48 Hour extension to accommodate Thanksgiving**
- **Assignment 9** available 08:00 **Monday November 23**
- **Assignment 9 is now an *INDIVIDUAL* project**

Today's Lecture

- JavaScript Promises
- Event Driven Programming
- Node.js
- Express for RESTful APIS
- OpenAPI
- The Books Example

- Assignment 7 - Introduction
- Questions

Our Full Stack Web Application



JavaScript
Node.js
Express
OpenAPI

JavaScript Callback Issues #1

- Assume:

```
function myFunc(msg, doneCallback) {
  // do something with msg
  callback(`${msg} and goodbye`);
}
```

- Then if we write:

```
myFunc('hello', function(msg) {
  console.log(msg);
});
console.log('continuing...');
```

- In what order will the console log messages appear?

JavaScript Callback Issues #2

- Assume:

```
function func1(msg, doneCallback) { ... }
function func2(msg, doneCallback) { ... }
function func3(msg, doneCallback) { ... }
```

- Then if we write:

```
func1('hello', function(msg) {
  func2(msg, function(msg) {
    func3(msg, function(msg) {
      console.log(`finally done everything! ${msg}`);
    });
  });
});
```

- We have "The Pyramid of Doom" ☹

JavaScript Promises

- Rather than specifying a done callback

```
doSomething(args, doneCallback);
```

- Return a **Promise** that will be fulfilled when done

```
let donePromise = doSomething(args);
```

(`donePromise` will be fulfilled when `doSomething` completes)

- We don't need to immediately wait until promises are fulfilled
- Get the value of a promise (waiting if need be) with `then()`

```
donePromise.then(function (value) {
  // value is the promised result when successful
}, function (error) {
  // Error case
});
```


Promise Chaining

```
return myReadFile(fileName)
  .then(function (fileData) { return modifyData(fileData); })
  .then(function (data) { return finalizeData(data); })
  .catch(errorHandler);
```

Or with arrow functions:

```
return myReadFile(fileName)
  .then((fileData) => modifyData(fileData))
  .then((data) => finalizeData(data))
  .catch(errorHandler);
```

Much easier to read than the Pyramid of Doom ☺

`async` and `await` keywords

- *'Syntactic Sugar'* for Promises
- Declare a function to return a Promise

```
async function returnOne() {
  return 1;
}
```

- Resolve the Promise and extract its value

```
let one = await returnOne();
console.log(one);
```
(Prints 1)

- Note: `await` is only valid inside `async` functions

Node.js

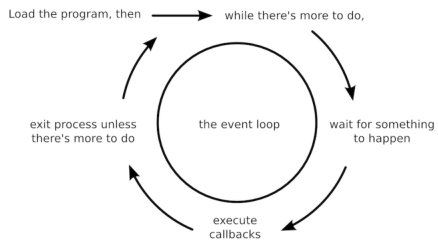
Event Driven Programming

- GUI Programming
  - Respond to user events
    - Click a button, scroll the mouse, etc.
- Java Examples:

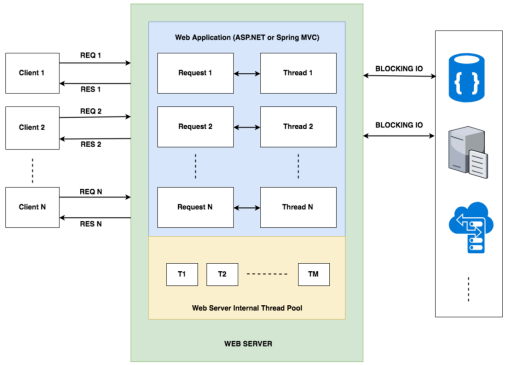
```
myButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        myFrame.toFront();
    }
});
myOtherButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        new Thread() {
            public void run(){
                // some long running action
            }
        }.start();
    }
});
```


Event Driven Programming

- Single Threaded Event Loop
  - Implicitly Asynchronous



Traditional Web Application



Multi Threaded vs Event Driven

Multiple Threads	Event-driven
Main application thread with listener / worker threads	Single thread, repeatedly fetching events one-at-a-time
Responds to incoming requests	Queues then processes events
Can block requests that involve multiple events	Manually saves state and goes on to process the next event
Thread and/or process context switching required	No thread contention and no context switches
Synchronous I/O in multiple threads: send, recv, poll, select, etc.	Asynchronous I/O via callbacks

Node.js : Big Ideas

- Perform asynchronous processing in a single thread instead of classical multithreaded processing
  - Minimize overhead & latency
  - Maximize scalability
  - Completely bypass concurrency problems
- Intended for applications that serve many clients but do not need lots of computational power for each request
  - Ideal for Web Apps
  - Not good for heavy calculations, e.g. massive parallel computing
- What Node.js is not:
  - A language
  - A web framework

Node.js : Principles

- Standard Library and Modules
- Callbacks
- The Event Loop
- Blocking vs. Non-Blocking
- Timers

Node.js : Standard Library Modules

assert	Assertion tests
buffer	Handle binary data
child_process	Run a child process
cluster	Split a single Node process into multiple processes
crypto	Handle OpenSSL cryptographic functions
dgram	Provides implementation of UDP datagram sockets
dns	DNS lookups and name resolution functions
events	Handle events
fs	Interact with the file system
http	HTTP server
https	HTTPS server
net	Create servers and clients
os	Provides information about the operation system
path	Handle file paths
querystring	Handle URL query strings
readline	Handle readable streams one line at the time
stream	Handle streaming data
string_decoder	Decode buffer objects into strings
timers	Execute a function after a given number of milliseconds
tls	Implement TLS and SSL protocols
tty	Provides classes used by a text terminal
url	Parse URL strings
util	Access utility functions
v8	Access information about V8 (the JavaScript engine)
vm	Compile JavaScript code in a virtual machine
zlib	Compress or decompress files

Node.js : Using Modules

- Install:

```
$ npm install <nodule name>
```

- Use:

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


Node.js : Callbacks (callback functions)

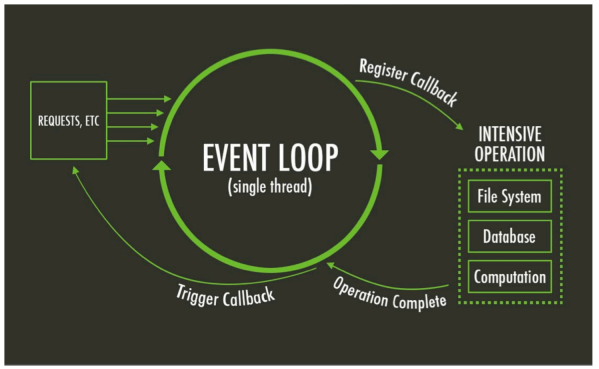
- Traditional synchronous approach:

```
function processData () {
    var data = fetchData ();
    data += 1;
    return data;
}
```

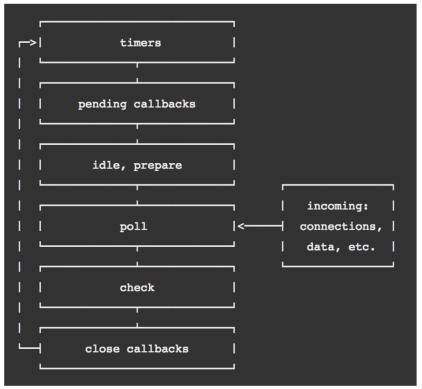
- Asynchronous approach with a callback function:

```
function processData (doneCallback) {
    fetchData(function (err, data) {
        if (err) {
            console.log("An error has occurred!");
            doneCallback(err);
        } else {
            data += 1;
            doneCallback(data);
        }
    });
}
```


Node.js : The Event Loop



Node.js : The Event Loop



Node.js : Blocking vs. Non-Blocking

- Blocking
  - Execution of JavaScript in the Node.js process must wait until a non-JavaScript operation completes

```
const fs = require('fs');
const data = fs.readFileSync('big-file');
console.log(data);
console.log('doing something useful');
```

```
<Buffer 68 65 6c 6c 6f 0a>
doing something useful
```

- Non-Blocking
  - All Node.js standard library I/O methods are non-blocking and accept callbacks - some have blocking equivalents ending in `Sync`

```
const fs = require('fs');
fs.readFile('big-file', (err, data) => {
    if (err) throw err;
    console.log(data);
});
console.log('doing something useful');
```

```
doing something useful
<Buffer 68 65 6c 6c 6f 0a>
```


Node.js : Blocking vs. Non-Blocking

- Dangers of mixing blocking and non-blocking calls:

```
const fs = require('fs');
fs.readFile('big-file', (err, data) => {
    if (err) throw err;
    console.log(data);
});
fs.unlinkSync('big-file');
```

'big-file' may be removed before being read ☹

```
const fs = require('fs');
fs.readFile('big-file', (readFileErr, data) => {
    if (readFileErr) throw readFileErr;
    console.log(data);
    fs.unlink('big-file', (unlinkErr) => {
        if (unlinkErr) throw unlinkErr;
    });
});
```


Node.js : Timers (delayed execution)

- Timers in Single Threaded Event Loop Systems can be problematic
- Node.js provides several mechanism:
  - **When I Say So**:
    - Execute once at some point in the future

```
setTimeout(some_func, 1500, 'some argument');
```

  - **Right After This**:
    - Execute once at the end of the current event loop

```
setImmediate(some_func, 'some argument');
```

  - **Infinite Loop:**
    - Execute forever at set intervals

```
setInterval(some_func, 1500, 'some argument');
```

Node.js Success Stories

Ruby on Rails to Node.js
Reduced number of servers by an order of magnitude
Some services ran 20x faster
Mobile App front-end and back-end teams combined

Java to Node.js
Built twice as fast with fewer developers
Doubled the number of requests handled per second
35% decrease in average response times

Express

Express

- A Web Framework for Node.js
  - Fast, Unopinionated, Minimalist ☺
- Principal Services:
  - **HTTP Server**
    - Accept TCP Connections
    - Process HTTP Requests
    - Send HTTP Replies
  - **Routing**
    - Map URLs to JavaScript functions
    - Like React Router in some ways
  - **Middleware**
    - Support for HTTP Request pre-processors and Response post-processors
    - Makes it easy to inject support for:
      - Cookies, sessions, authentication, compression, decompression, …
      - API Validation (we'll do this)

Simple Express Web Server

```
1  const express = require('express');
2
3  var app = express();
4
5  app.get('/', function (req, res) {
6    res.send('Hello CSE183 World');
7  });
8
9  app.listen(3010, () => {
10   console.log('Server Running on port 3010');
11 });
```

localhost:3010

Hello CSE183 World

localhost:3010/fred

Cannot GET /fred

Express Routing

- Numerous mechanisms
- Most common is by HTTP Method:

```
app.get(urlPath, requestProcessingFunction);
app.post(urlPath, requestProcessingFunction);
app.put(urlPath, requestProcessingFunction);
app.delete(urlPath, requestProcessingFunction);
app.all(urlPath, requestProcessingFunction);
```

Note: `urlPath` can contain parameters (e.g. `'/user/:id'`)

The `req` Object

- Shorthand for `httpRequest`

```
app.get('/user/:id', function (req, res){ … });
```

- Object with many properties
  - Commonly used properties:
    - `req.params`
      URL route parameters (`id` in the case above)
    - `req.query`
      Query parameters (e.g. `&foo=9` in URL ⇒ `{foo: '9'}` in `req.query`)
    - `req.body`
      The parsed request body
    - `req.get(field)`
      Value of the specified HTTP header field
- Middleware can add additional properties
  - E.g. JSON body parser, session manager, API validator, etc.

The `res` Object

- Shorthand for `httpResponse`

```
app.get('/user/:id', function (req, res){ … });
```

- Object methods for setting HTTP response fields:

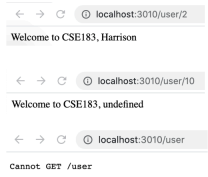
`res.write(content)`	Build up the response body with content
`res.status(code)`	Set the HTTP status code of the response
`res.set(prop, value)`	Set a response header property value
`res.end()`	End the request by responding to it
`res.end(msg)`	End the request by responding with msg
`res.send(content)`	Do a `write()` and an `end()`

- Methods return the `res` object so they can "stack"

```
res.status(code)
  .write('something')
  .write('something else')
  .end();
```


Parameterised Express Web Server

```
1  const express = require('express');
2
3  var app = express();
4
5  const users = new Map();
6  users.set(1, 'David');
7  users.set(2, 'Harrison');
8
9  app.get('/user/:id', function (req, res) {
10   res.send(`Welcome to CSE183, ${users.get(parseInt(req.params.id))}`);
11 });
12
13 app.listen(3010, () => {
14   console.log('Server Running on port 3010');
15 });
```



Request Processing Function for Route `/user/:id` extracts the `id` parameter from the HTTP request and finds the user with that id in the global Map of users

Express for RESTful APIs

- Express Principally used to implement REST services
- Consider a simple book API:
  - Retrieve all the known books
  - Retrieve one book by its ISBN
  - Add a new book
- Express Routes:

  GET `/v0/books`

  GET `/v0/book/:isbn`

  POST `/v0/book`
- API Versioning:

  `/v0` Development version
  `/v1` First production version
  `/v1.1` Minor changes to first production version
  `/v2.0` Second production version – major changes from first production version

The Books Example

We need Express and a custom written module to handle requests to the API

```
const express = require('express');
const books = require('./books');
```

Start Express and add Middleware to parse HTTP request bodies as JSON and support URL Encoding

```
const app = express();
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
```

Setup the Express Routes by mapping them to request handler functions implemented in books.js

```
app.get('/v0/books',       books.getAll);
app.get('/v0/books/:isbn', books.getByISBN);
app.post('/v0/books',      books.post);
```

Load JSON containing all the known books

```
const books = require('../data/books.json');
```

Return all the books as a large chunk of JSON with an HTTP 200 to indicate success

```
exports.getAll = async (req, res) => {
  res.status(200).json({books: books});
}
```

Return the book identified by the ISBN parameter with an HTTP 200 to indicate success

Or return an HTTP 404 if the book is unknown

```
exports.getByISBN = async (req, res) => {
  const book = books.find(book => book.isbn == req.params.isbn);
  if (book) {
    res.status(200).json(book);
  } else {
    res.status(404).send();
  }
}
```

Insert a new book into the 'database' JSON and return it with an HTTP 201 to indicate a successful insertion

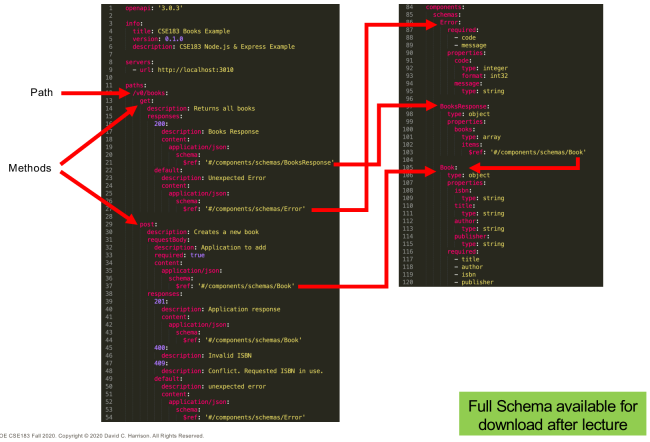
Or return an HTTP 409 if the book already exists

```
exports.post = async (req, res) => {
  const book = books.find(book => book.isbn == req.body.isbn);
  if (book) {
    res.status(409).send();
  } else {
    books.push(req.body);
    res.status(201).send(req.body);
  }
}
```


OpenAPI

- How do consumers of our API know:
  - What end-points exist?
  - What parameters and queries are accepted by each end-point?
  - What the types of all the parameters are?
  - What HTTP codes are returned and under what circumstances?
  - What the format of request and response bodies are?
- All these questions, and more, can be answered by creating an OpenAPI **Schema** that defines our API ☺
- By adding some middleware to Express we can ensure requests and responses adhere to the Schema ☺☺☺

Books Example Schema



Full Schema available for download after lecture

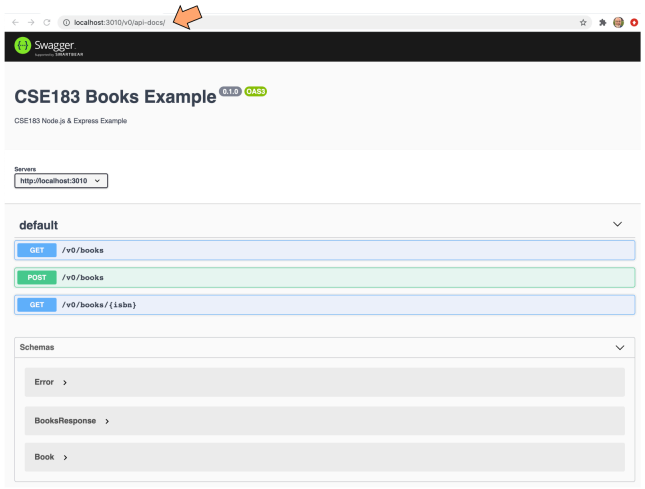
Books Example Schema Validation

Extra packages to locate the OpenAPI Schema and Middleware to validate it

Installing the OpenAPI Validator Middleware
Requests *and* Responses will be checked

Middleware to ensure OpenAPI Schema Validation errors are sent back to the caller

```
const express = require('express');
const path = require('path');
const OpenApiValidator = require('express-openapi-validator');

const books = require('./books');

const app = express();
app.use(express.json());
app.use(express.urlencoded({ extended: false }));

const apiSpec = path.join(__dirname, '../api/openapi.yaml');

app.use(
  OpenApiValidator.middleware({
    apiSpec: apiSpec,
    validateRequests: true,
    validateResponses: true,
  }),
);

app.get('/v0/books',       books.getAll);
app.get('/v0/books/:isbn', books.getByISBN);
app.post('/v0/books',      books.post);

app.use((err, req, res, next) => {
  res.status(err.status).json({
    message: err.message,
    errors: err.errors,
    status: err.status,
  });
});
```

Testing a RESTful API

```
const supertest = require('supertest');
const http = require('http');

const app = require('../src/app');

let server;

beforeAll(() => {
  server = http.createServer(app);
  server.listen();
  request = supertest(server);
});

afterAll((done) => {
  server.close(done);
});

test('GET Invalid URL', async () => {
  await request.get('/v0/bookie-wookie/')
    .expect(404)
});

test('GET All', async () => {
  await request.get('/v0/books')
    .expect(200)
    .expect('Content-Type', /json/)
    .then(data => {
      expect(data).toBeDefined();
      expect(data.body).toBeDefined();
      expect(data.body.books).toBeDefined();
      expect(data.body.books.length).toEqual(256);
    })
});
```

Supertest enhanced Jest for easy access to our API server over HTTP

Start our Express server (code is in `../src/app.js`) before the tests run and save a reference to it so it can be shutdown afterwards

Test to ensure invalid URLs are rejected

Test to ensure the correct number of known books are returned

Note this is an example or Promise Chaining

Many other tests available for download after lecture

Books Example Test Results

```
PASS test/books.test.js
  ✓ GET Invalid URL (97 ms)
  ✓ GET All (12 ms)
  ✓ GET One (4 ms)
  ✓ GET Missing (1 ms)
  ✓ GET Invalid ISBN  (2 ms)
  ✓ POST New (11 ms)
  ✓ GET After POST (2 ms)
  ✓ POST Invalid ISBN (1 ms)
  ✓ POST Exisiting ISBN (2 ms)

----------|---------|----------|---------|---------|-------------------
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------|---------|----------|---------|---------|-------------------
All files |     100 |      100 |     100 |     100 |
 app.js   |     100 |      100 |     100 |     100 |
 books.js |     100 |      100 |     100 |     100 |
----------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       9 passed, 9 total
Snapshots:   0 total
Time:        1.419 s, estimated 2 s
Ran all test suites.
```

Assignment 7

- E-Mail API in Node.js / Express / OpenAPI
- You're given three mailboxes full of E-Mails
  - Inbox, Sent, Trash
- Endpoints:

`GET /v0/mail`	Retrieve all the mail
`GET /v0/mail?mailbox={mailbox}`	Retrieve one mailbox
`GET /v0/mail/{id}`	Retrieve one E-Mail
`POST /v0/mail`	Send an E-Mail (goes in Sent)
`PUT /v0/mail/{id}?mailbox={mailbox}`	Move an E-Mail to different mailbox

- Base your submission on the Books Example ☺

Assignment 7 - Starter Code

Assignment 7 - Requirements

- **Basic** - 4 Points
  - Write the OpenAPI Schema
  - Implement all the endpoints
- **Advanced** - 2 Points
  - Write tests for all the endpoints
  - 100% code coverage
  - Zero linter errors
- **Stretch** - 1 Point
  - Save the in-memory mailboxes to disk when they change
  - When server re-starts, mailboxes are restored from disk
  - Write tests to demonstrate this works
  - Maintain 100% code coverage and Zero linter errors

Running `npm zip` will create a correctly formatted submission archive for upload to Canvas ☺

Upcoming Lectures

- **Monday 16th** **In-Class Quiz 3** (Plus Assignments 5 & 6 Review)
- Wednesday 18th Storage Tier
- Friday 20th Cookies & Session Data

Tasks

- **Assignment 7** due 23:59 **Thursday November 19**