Backend Development

SWE 432, Fall 2018
Web Application Development
Review: Async Programming Example

- Go get a candy bar
- Go get a candy bar
- Go get a candy bar
- Go get a candy bar
- Go get a candy bar
- Go get a candy bar
- Go get a candy bar
- Go get a candy bar

Go get a candy bar
Go get a candy bar
Go get a candy bar
Go get a candy bar
Go get a candy bar
Go get a candy bar
Go get a candy bar
Go get a candy bar

thenCombine

1 second each
Group all Twix
Group all 3 Musketeers
Group all MilkyWay
Group all MilkyWay Dark
Group all Snickers

2 seconds each

when done

Eat all the Twix

Explain example
Review: Async/Await

• Rules of the road:

  • You can only call **await** from a function that is **async**

  • You can only **await** on functions that return a **Promise**

  • Beware: await makes your code synchronous!

```javascript
async function getAndGroupStuff() {
  ...
  ts = await lib.groupPromise(stuff,"t");
  ...
}
```
Today

• What is a backend for?
• History of backend web programming
• NodeJS backends with Express
Why we need backends

• Security: *SOME* part of our code needs to be “trusted”
  • Validation, security, etc. that we don’t want to allow users to bypass

• Performance:
  • Avoid duplicating computation (do it once and cache)
  • Do heavy computation on more powerful machines
  • Do data-intensive computation “nearer” to the data

• Compatibility:
  • Can bring some dynamic behavior without requiring much JS support
Dynamic Web Apps

What the user interacts with
Web “Front End”

Presentation
Some logic

What the front end interacts with

Frontend programming
next week

“Back End”

Data storage
Some other logic

Persistent Storage
Some other APIs
Where do we put the logic?

**Web “Front End”**
- Presentation
- Some logic
- Data storage
- Some other logic

**Frontend**

**Pros**
- Very responsive (low latency)

**Cons**
- Security
- Performance
- Unable to share between front-ends

**Backend**

**Pros**
- Easy to refactor between multiple clients
- Logic is hidden from users (good for security, compatibility, and intensive computation)

**Cons**
- Interactions require a round-trip to server
Why Trust Matters

• Example: Transaction app

```javascript
function updateBalance(user, amountToAdd) {
    user.balance = user.balance + amountToAdd;
}
```

• What’s wrong?

• How do you fix that?
What does our backend look like?

Web “Front End”

AJAX

Our own backend

Connection to Frontend

Logic

Persistent Data
The “good” old days of backends

HTTP Request
GET /myApplicationEndpoint HTTP/1.1
Host: cs.gmu.edu
Accept: text/html

Runs a program

Web Server
Application

My Application Backend

HTTP Response
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8

<html><head>...

Give me /myApplicationEndpoint

Does whatever it wants

Here’s some text to send back
What’s wrong with this picture?
History of Backend Development

• In the beginning, you wrote whatever you wanted using whatever language you wanted and whatever framework you wanted

• Then… PHP and ASP
  • Languages “designed” for writing backends
  • Encouraged spaghetti code
  • A lot of the web was built on this

• A whole lot of other languages were also springing up in the 90’s…
  • Ruby, Python, JSP
MVC & Backend Servers

• There are a ton of backend frameworks that support MVC
  • SailsJS, Ruby on Rails, PHP Symfony, Python Django, ASP.NET, EJB…

• Old days: View was server-generated HTML

• New days: View is an API

• Today we’ll talk about Node.JS backend development

• We will not talk about making MVC backends and will not require you to do so
Microservices vs. Monoliths

• Advantages of microservices over monoliths include
  • Support for scaling
    • Scale vertically rather than horizontally
  • Support for change
    • Support hot deployment of updates
  • Support for reuse
    • Use same web service in multiple apps
    • Swap out internally developed web service for externally developed web service
  • Support for separate team development
    • Pick boundaries that match team responsibilities
  • Support for failure
Support for scaling

Our Cool App

Frontend

Backend Server

Mod 1  Mod 2
Mod 3  Mod 4
Mod 5  Mod 6

Database
Now how do we scale it?

We run multiple copies of the backend, each with each of the modules.
What's wrong with this picture?

- This is called the “monolithic” app

- If we need 100 servers…

- Each server will have to run EACH module

- What if we need more of some modules than others?
Microservices

Our Cool App

Frontend

“Dumb” Backend

NodeJS, Firebase

Todos

REST service

Database

Google Service

Accounts

REST service

Database

Java, MySQL

Mailer

REST service

Database

AJAX

Search Engine

REST service

Database

Java, Neo4J

Analytics

REST service

Database

C#, SQLServer

Facebook Crawler

REST service

Database

Python, Firebase
Goals of microservices

• Add them independently
• Upgrade the independently
• Reuse them independently
• Develop them independently

• ==> Have ZERO coupling between microservices, aside from their shared interface
Node.JS

- We’re going to write backends with Node.JS

- Why use Node?
  - Event based: really efficient for sending lots of quick updates to lots of clients

- Why not use Node?
  - Bad for CPU heavy stuff
  - It’s relatively immature
Node.JS

• Node.JS is a runtime that lets you run JS outside of a browser

• How we’ve been running JS so far, mostly (browser will start next week)

• Node.JS has a very large ecosystem of packages as we’ve seen

  • Very relevant example here: express (web server)
Express

• Basic setup:

  • For get:

    ```javascript
    app.get("/somePath", function(req, res){
      //Read stuff from req, then call res.send(myResponse)
    });
    ```

  • For post:

    ```javascript
    app.post("/somePath", function(req, res){
      //Read stuff from req, then call res.send(myResponse)
    });
    ```

  • Serving static files:

    ```javascript
    app.use(express.static('myFileWithStaticFiles'));
    ```

    • Make sure to declare this *last*

• Additional helpful module - bodyParser (for reading POST data)
Demo: Hello World Server

1: Make a directory, myapp

2: Enter that directory, type `npm init` (accept all defaults)

3: Type `npm install express --save`

4: Create text file `app.js`:

```javascript
var express = require('express');
var app = express();
var port = process.env.port || 3000;
app.get('/', function (req, res) {
    res.send('Hello World!');
});
app.listen(port, function () {
    console.log('Example app listening on port ' + port);
});
```

5: Type `node app.js`

6: Point your browser to `http://localhost:3000`
Demo: Hello World Server

```
var express = require('express');
  // Import the module express

var app = express();
  // Create a new instance of express

var port = process.env.port || 3000;
  // Decide what port we want express to listen on

app.get('/', function (req, res) {
  res.send('Hello World!');
});
  // Create a callback for express to call when we have a “get” request to “/“. That callback has access to the request (req) and response (res).

app.listen(port, function () {
  console.log('Example app listening on port ' + port);
});
  // Tell our new instance of express to listen on port, and print to the console once it starts successfully
```
Core concept: Routing

• The definition of end points (URIs) and how they respond to client requests.

• `app.METHOD(PATH, HANDLER)`

• `METHOD`: all, get, post, put, delete, [and others]

• `PATH`: string

• `HANDLER`: call back

```javascript
app.post('/', function (req, res) {
  res.send('Got a POST request');
});
```
Route paths

• Can specify strings, string patterns, and regular expressions
  • Can use ?, +, *, and ()

• Matches request to root route
  ```javascript
  app.get('/', function (req, res) {
    res.send('root');
  });
  ```

• Matches request to /about
  ```javascript
  app.get('/about', function (req, res) {
    res.send('about');
  });
  ```

• Matches request to /abe and /abcde
  ```javascript
  app.get('/ab(cd)?e', function (req, res) {
    res.send('ab(cd)?e');
  });
  ```
Route parameters

- Named URL segments that capture values at specified location in URL
  - Stored into `req.params` object by name

- Example
  - Route path `/users/:userId/books/:bookId`
  - Request URL `http://localhost:3000/users/34/books/8989`
  - Resulting `req.params`: `{ "userId": "34", "bookId": "8989" }`

```javascript
app.get('/users/:userId/books/:bookId', function(req, res) {
  res.send(req.params);
});
```
Request object

- Enables reading properties of HTTP request
  - `req.body`: JSON submitted in request body (must define body-parser to use)
  - `req.ip`: IP of the address
  - `req.query`: URL query parameters
HTTP Responses

- Larger number of response codes (200 OK, 404 NOT FOUND)

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Content-Type: text/html; charset=UTF-8
Content-Encoding: UTF-8
Content-Length: 138
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
ETag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Connection: close

<html>
<head>
  <title>An Example Page</title>
</head>
<body>
  Hello World, this is a very simple HTML document.
</body>
</html>
```

“OK response”
Response status codes:
- 1xx Informational
- 2xx Success
- 3xx Redirection
- 4xx Client error
- 5xx Server error

“HTML returned content”
Common MIME types:
- application/json
- application/pdf
- image/png
Response object

- Enables a response to client to be generated
  - `res.send()` - send string content
  - `res.download()` - prompts for a file download
  - `res.json()` - sends a response with `application/json` Content-Type header
  - `res.redirect()` - sends a redirect response
  - `res.sendStatus()` - sends only a status message
  - `res.sendFile()` - sends the file at the specified path

```javascript
app.get('/users/:userId/books/:bookId', function(req, res) {
  res.json({ "id": req.params.bookID });
});
```
Describing Responses

- What happens if something goes wrong while handling HTTP request?
  - How does client know what happened and what to try next?
- HTTP offers response status codes describing the nature of the response
  - 1xx Informational: Request received, continuing
  - 2xx Success: Request received, understood, accepted, processed
  - 200: OK
  - 3xx Redirection: Client must take additional action to complete request
    - 301: Moved Permanently
    - 307: Temporary Redirect

Describing Errors

- 4xx Client Error: client did not make a valid request to server. Examples:
  - 400 Bad request (e.g., malformed syntax)
  - 403 Forbidden: client lacks necessary permissions
  - 404 Not found
  - 405 Method Not Allowed: specified HTTP action not allowed for resource
  - 408 Request Timeout: server timed out waiting for a request
  - 410 Gone: Resource has been intentionally removed and will not return
  - 429 Too Many Requests
Describing Errors

- 5xx Server Error: The server failed to fulfill an apparently valid request.
  - 500 Internal Server Error: generic error message
  - 501 Not Implemented
  - 503 Service Unavailable: server is currently unavailable
Error handling in Express

- Express offers a default error handler

- Can specific error explicitly with status
  - `res.status(500);`
Making HTTP Requests

- Writing clients that talk to backends

- Two good options: request, request-promise (need to install both to use request-promise)

```javascript
var rp = require('request-promise');

rp("http://localhost:3000/").then(v => {
    console.log("Response from server:");
    console.log(v);
}).catch(e => {
    console.log("Error");
    console.log(e);
});
```