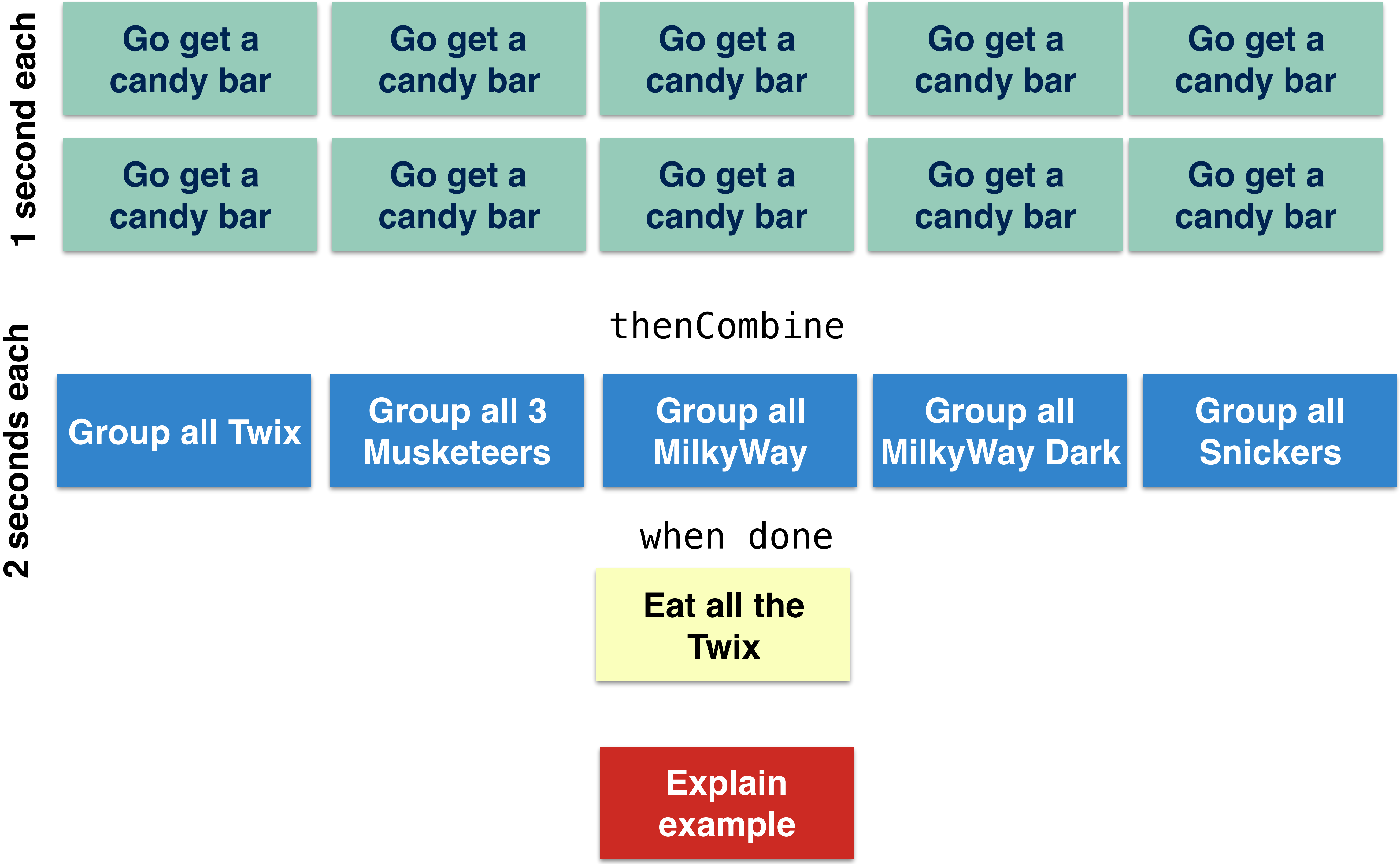
Backend Development

SWE 432, Fall 2019

Web Application Development

GEORGE MASON UNIVERSITY

Review: Async Programming Example



1 second each

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`

2 seconds each

Group all Twix

Group all 3 Musketeers

Group all MilkyWay

Group all MilkyWay Dark

Group all Snickers

`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!

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

Logistics

- HW2 released
    - Due 10/7 before class

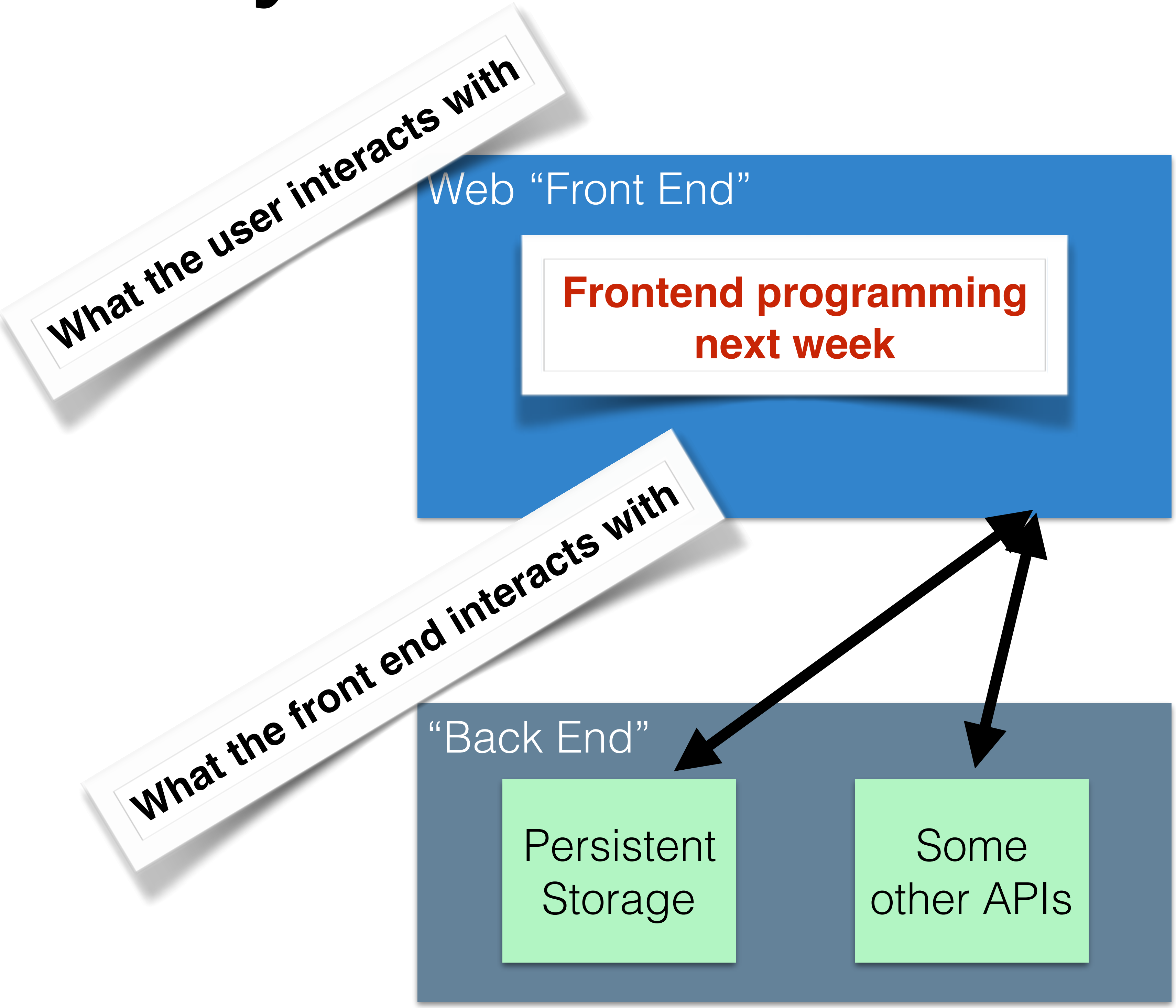
Today

- 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"
Frontend programming next week
What the front end interacts with
"Back End"
Persistent Storage
Some other APIs

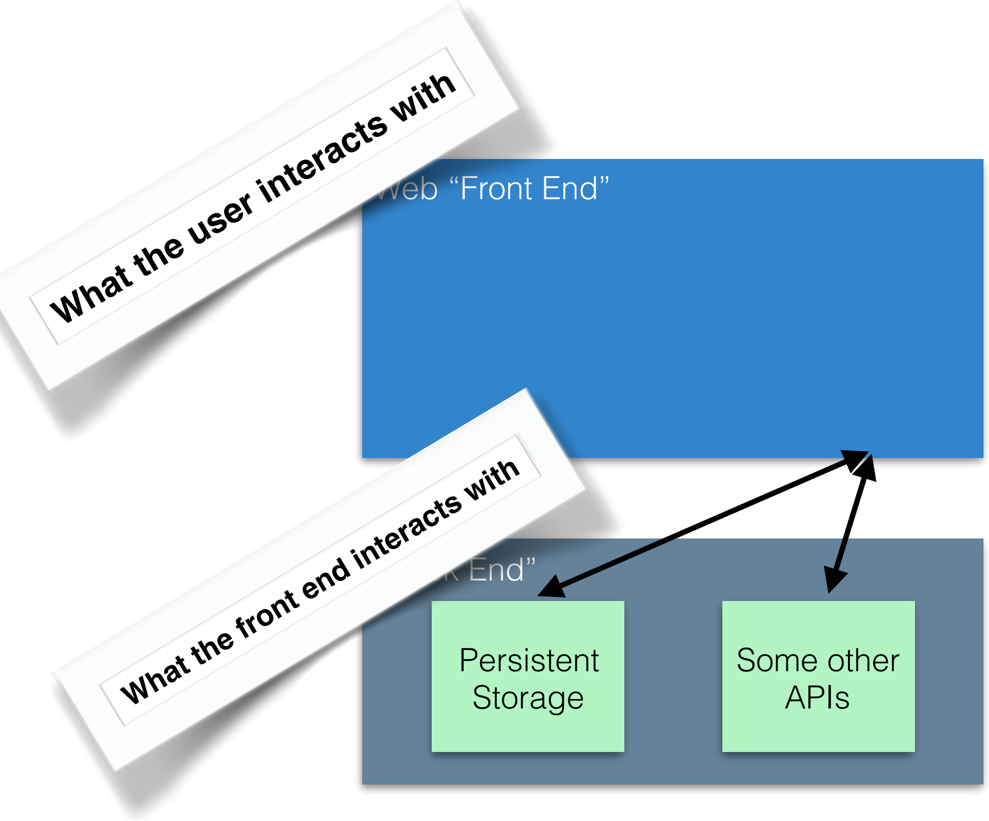
Presentation

Some logic

Data storage

Some other logic

Where do we put the logic?



What the user interacts with

Web “Front End”

What the front end interacts with

k End”

Persistent Storage

Some other APIs

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

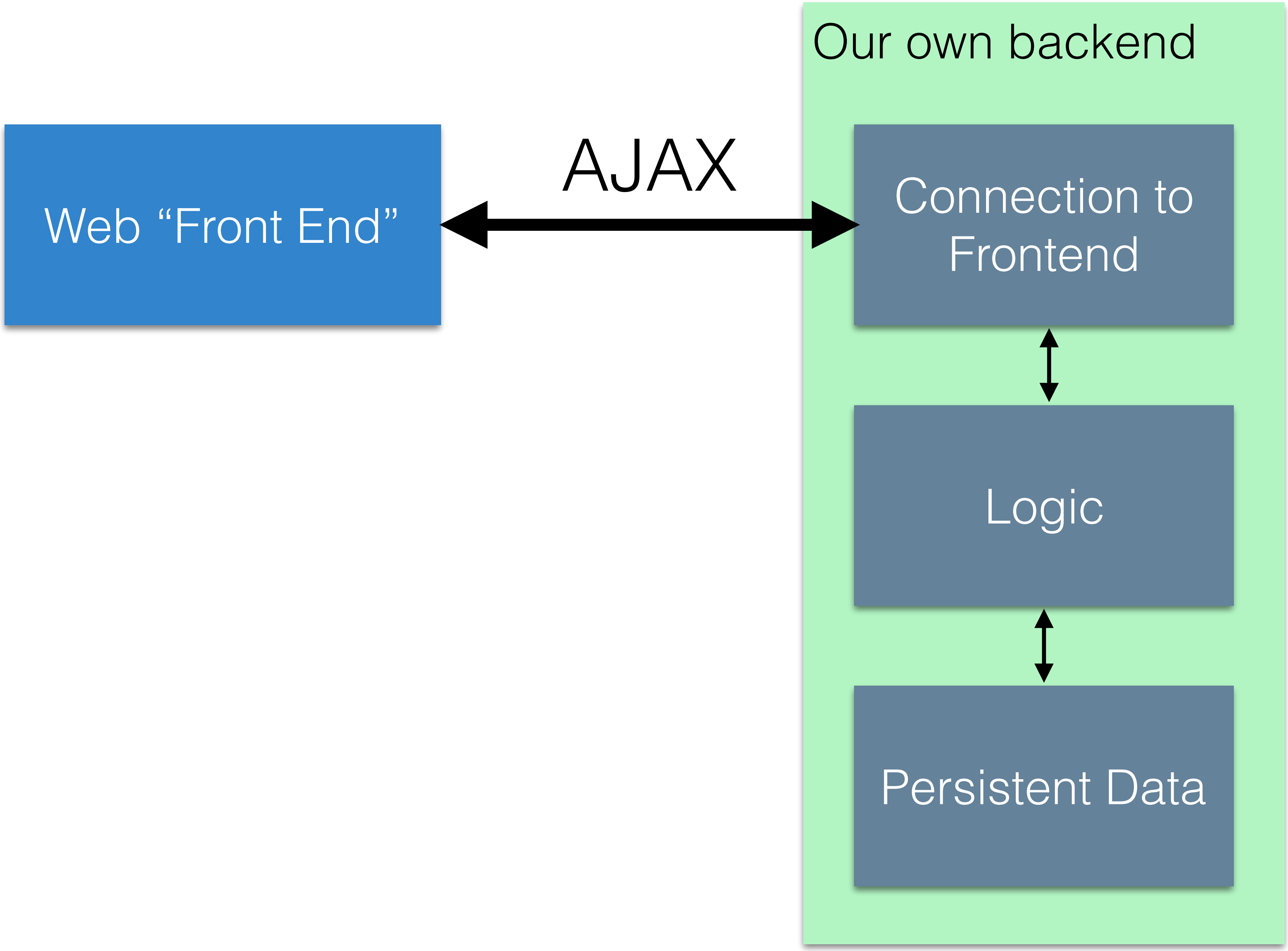
- Example: Banking app
  Imagine a banking app where the following code runs in the browser:

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

- What’s wrong?

- How do you fix that?

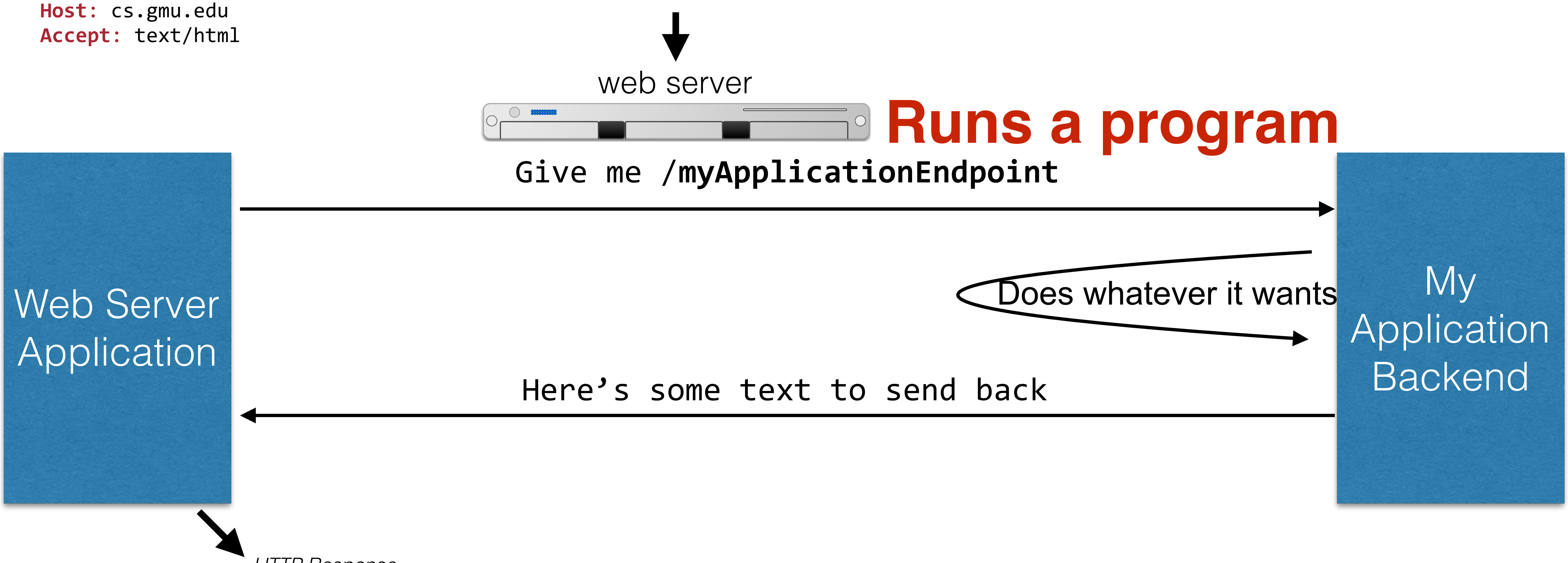
What does our backend look like?



The “good” old days of backends

HTTP Request

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



web server

Runs a program

Web Server Application

Give me /myApplicationEndpoint

Does whatever it wants

Here’s some text to send back

My Application Backend

HTTP Response

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

<html><head>...
```

What’s wrong with this picture?

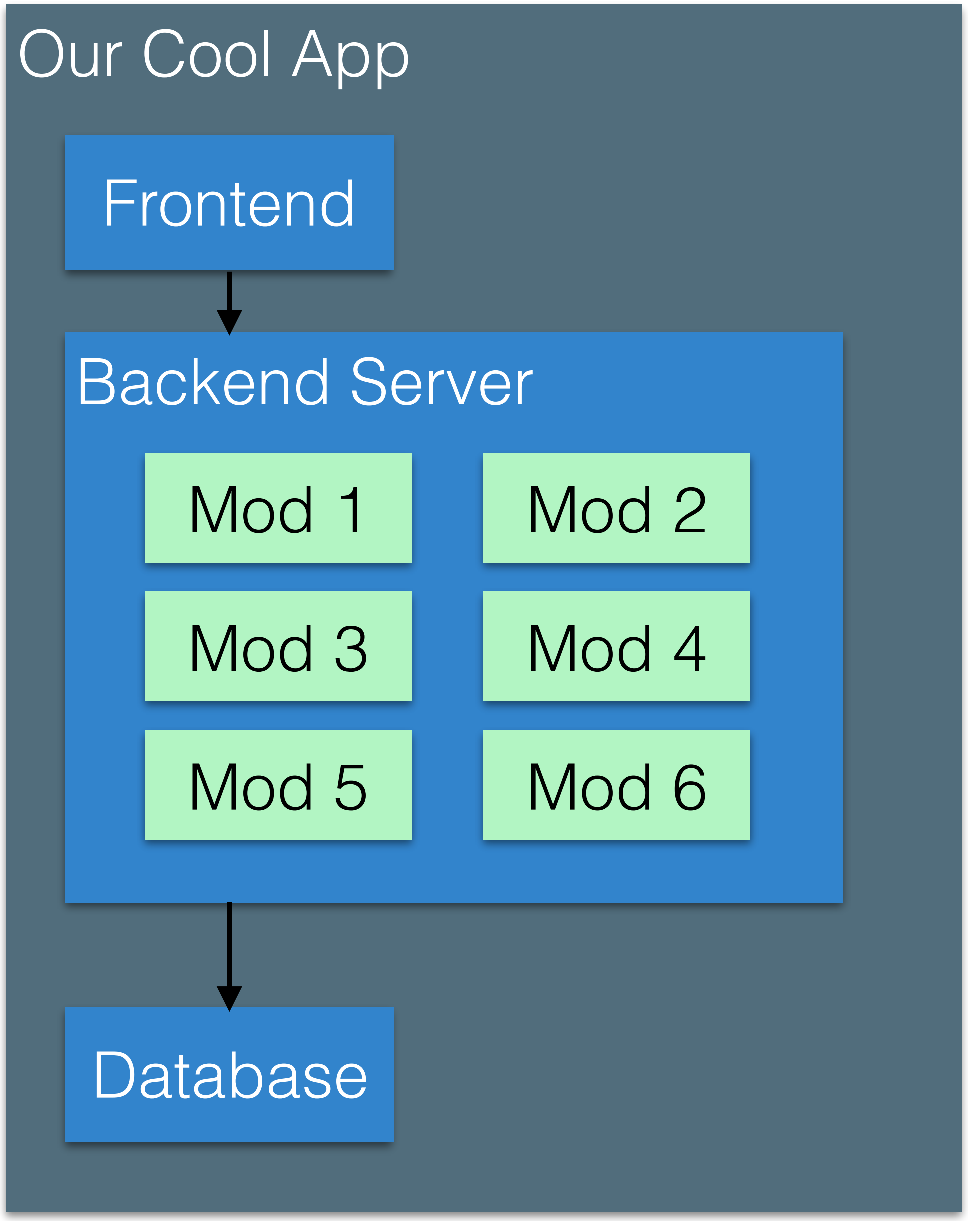
History of Backend Development

- 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

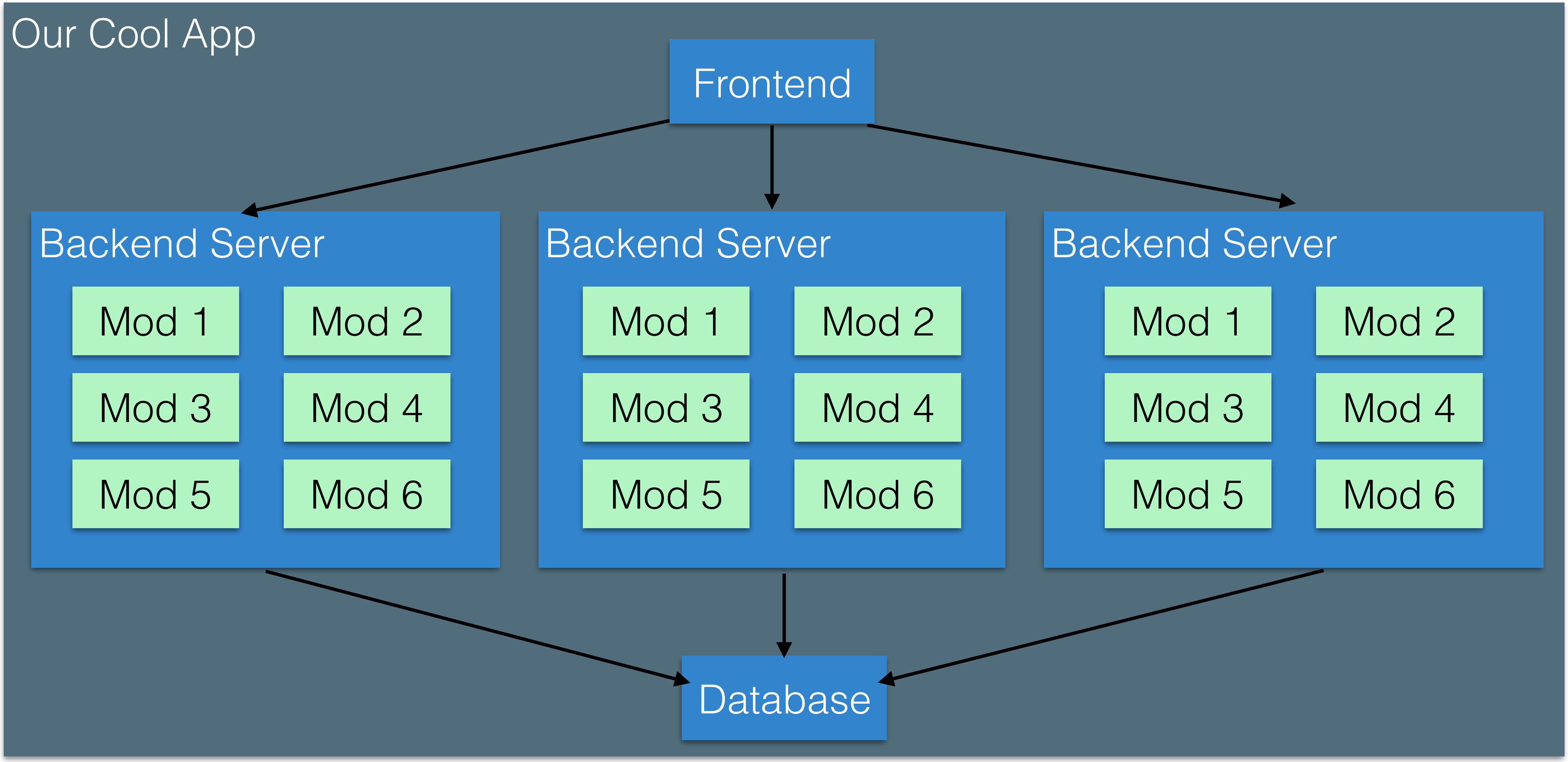
Microservices vs. Monoliths

- 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



Now how do we scale it?



We run multiple copies of the backend, each with each of the modules

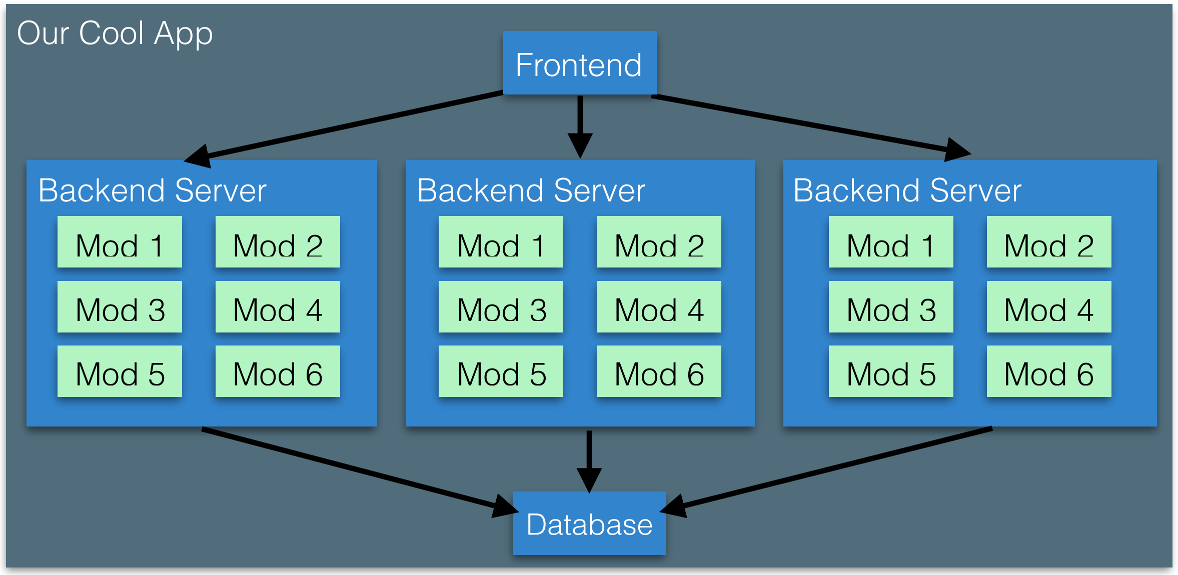
What's wrong with this picture?

- This is called the "monolithic" app

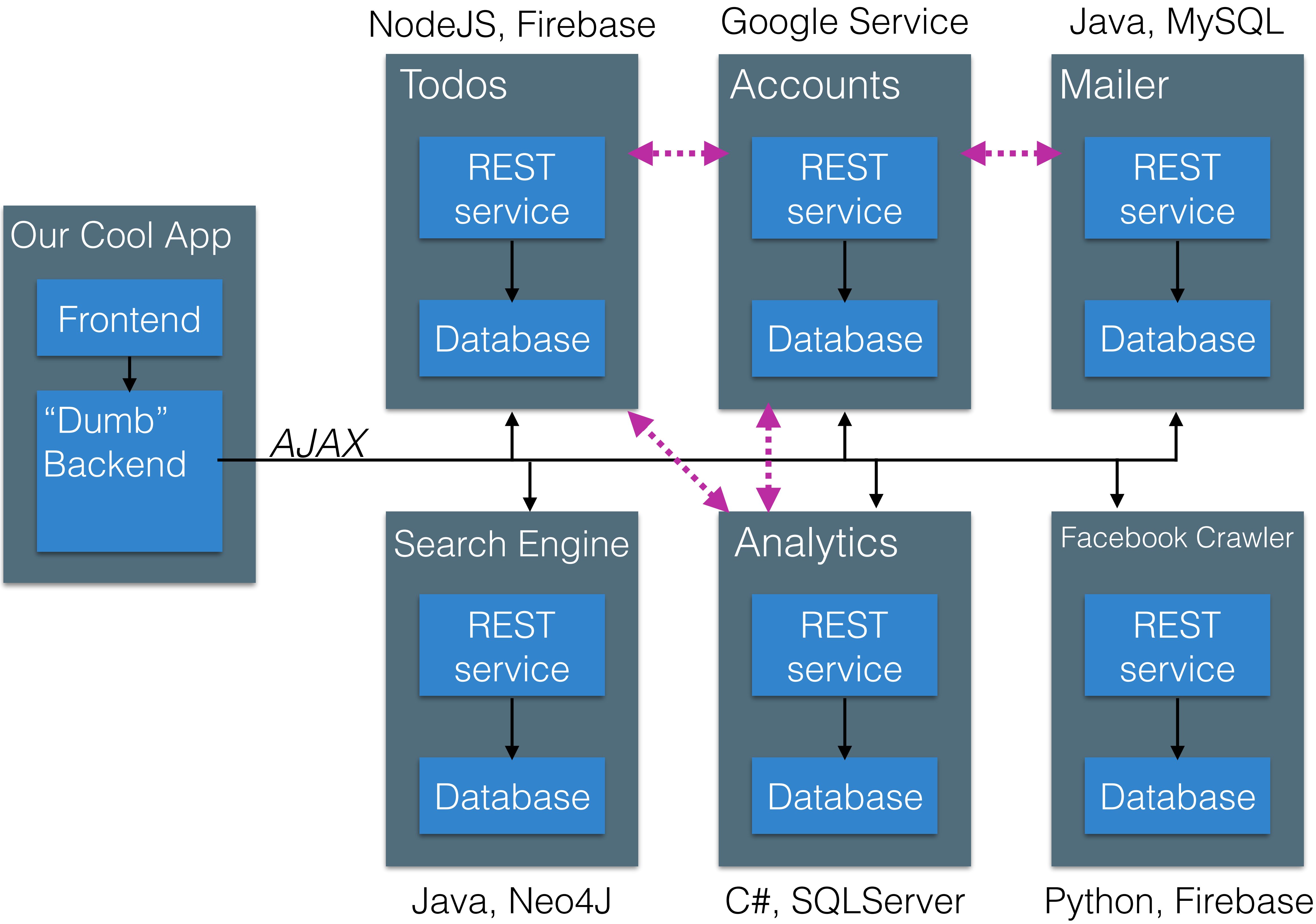
- If we need 100 servers…

- Each server will have to run EACH module

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



Microservices



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
    - Very large ecosystem of packages, as we've seen
- Why not use Node?
    - Bad for CPU heavy stuff

Express

- Basic setup:
    - For get:

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

- For post:

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

- Serving static files:

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

- Make sure to declare this *last*

- Additional helpful module - bodyParser (for reading POST data)

https://expressjs.com/

Demo: Hello World Server

1: Make a directory, myapp

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

Creates a configuration file for your project

3: Type **`npm install express --save`**

Tells NPM that you want to use express, and to save that in your project config

4: Create text file app.js:

```
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`**

Runs your app

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

```
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

```
app.get('/', function (req, res) {
  res.send('root');
});
```

- Matches request to /about

```
app.get('/about', function (req, res) {
  res.send('about');
});
```

- Matches request to /abe and /abcde

```
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" }`

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

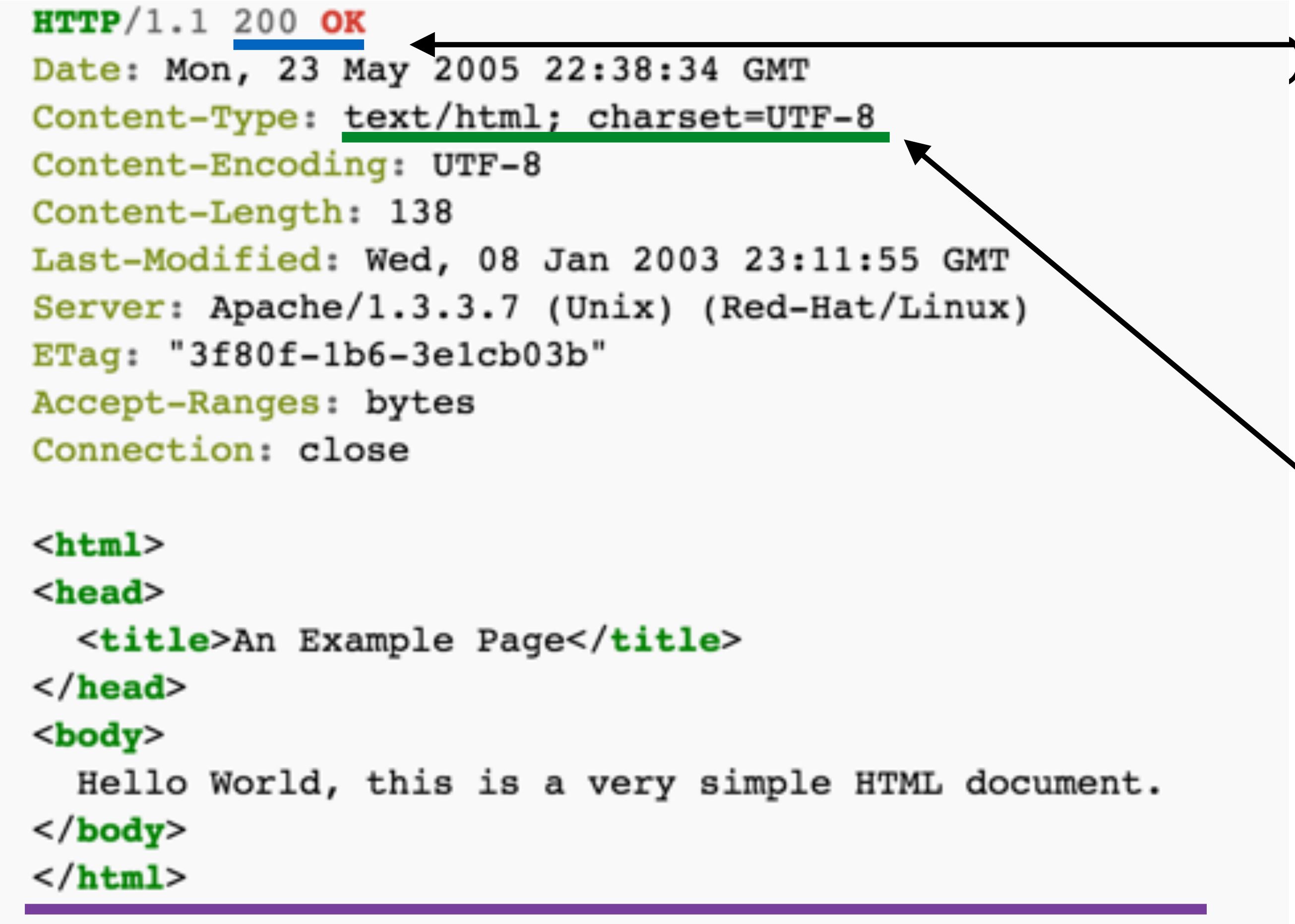
Request object

- 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)

with certain response



```
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
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
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

[HTML data]

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 w/ 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

```
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

https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

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);`

Persisting data in memory

- Can declare a global variable in node
    - i.e., a variable that is not declared inside a class or function
- Global variables persist between requests
- Can use them to store state in memory
- Unfortunately, if server crashes or restarts, state will be lost
    - Will look later at other options for persistence

Making HTTP Requests

- May want to request data from other servers from backend
- Fetch
  - Makes an HTTP request, returns a Promise for a response
  - Part of standard library in browser, but need to install library to use in backend
- Installing:

```
npm install node-fetch --save
```

- Use:

```
const fetch = require('node-fetch');

fetch('https://github.com/')
    .then(res => res.text())
    .then(body => console.log(body));

var res = await fetch('https://github.com/');
```

https://www.npmjs.com/package/node-fetch

Responding later

- What happens if you'd like to send data back to client in response, but not until something else happens (e.g., your request to a different server finishes)?

- Solution: wait for event, then send the response!

```
fetch('https://github.com/')
    .then(res => res.text())
    .then(body => res.send(body));
```