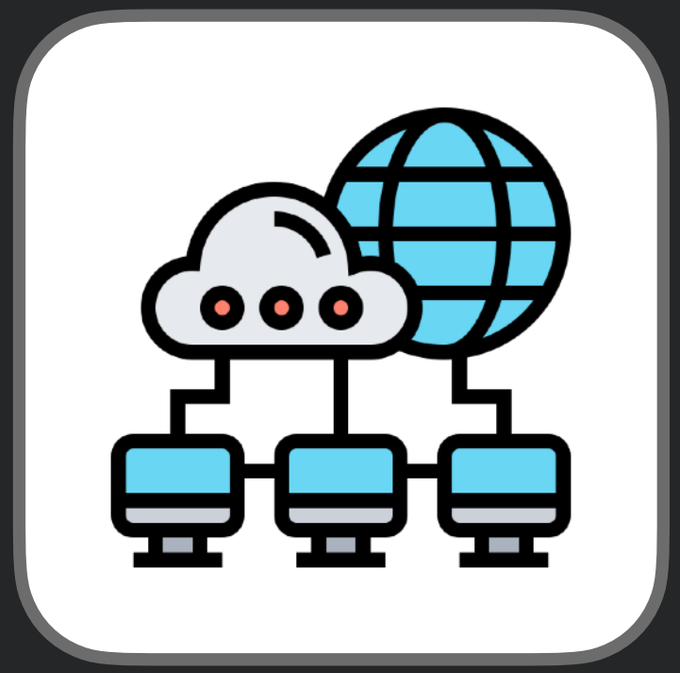
SWE 432 -Web Application Development

Fall 2021

George Mason University

Dr. Kevin Moran

Week 4: Backend Development & HTTP Requests



Administrivia

- *HW Assignment 1* - Grades Available on Blackboard - Detailed Comments in Replit

- *HW Assignment 2* - Due September 28th Before Class

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

2 seconds each

thenCombine

Group all Twix	Group all 3 Musketeers	Group all MilkyWay	Group all MilkyWay Dark	Group all Snickers

when done

Eat all the Twix

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");
...
}
```

Class Overview

- *Part 1 - Backend Programming:* A Brief History and Intro to Express with Node.js.
- *10 minute Break*
- *Part 2 -Handling HTTP Requests:* Exploring HTTP and REST
- *Part 3 - In-Class Activity:* E Getting Started with Backend Development & REST

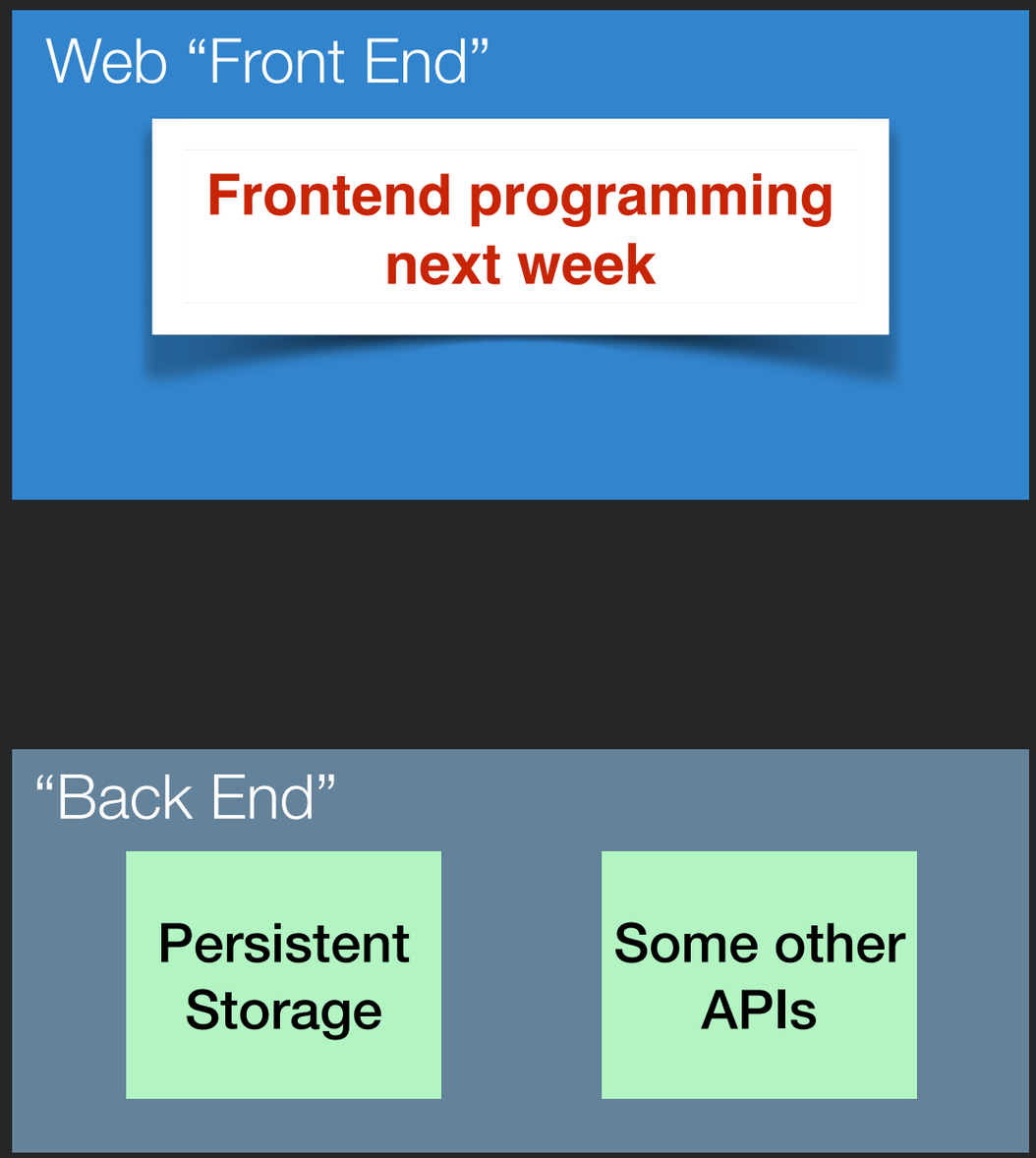
Why we need backends

- 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

Backend Web Development

A Brief History of Backend Programming

Dynamic Web Apps



Dynamic Web Apps

What the user interacts with

Web “Front End”

Frontend programming next week

“Back End”

Persistent Storage

Some other APIs

Dynamic Web Apps

What the user interacts with

Web "Front End"

Frontend programming next week

Presentation

"Back End"

Persistent Storage

Some other APIs

Dynamic Web Apps

What the user interacts with

Web "Front End"

Frontend programming next week

"Back End"

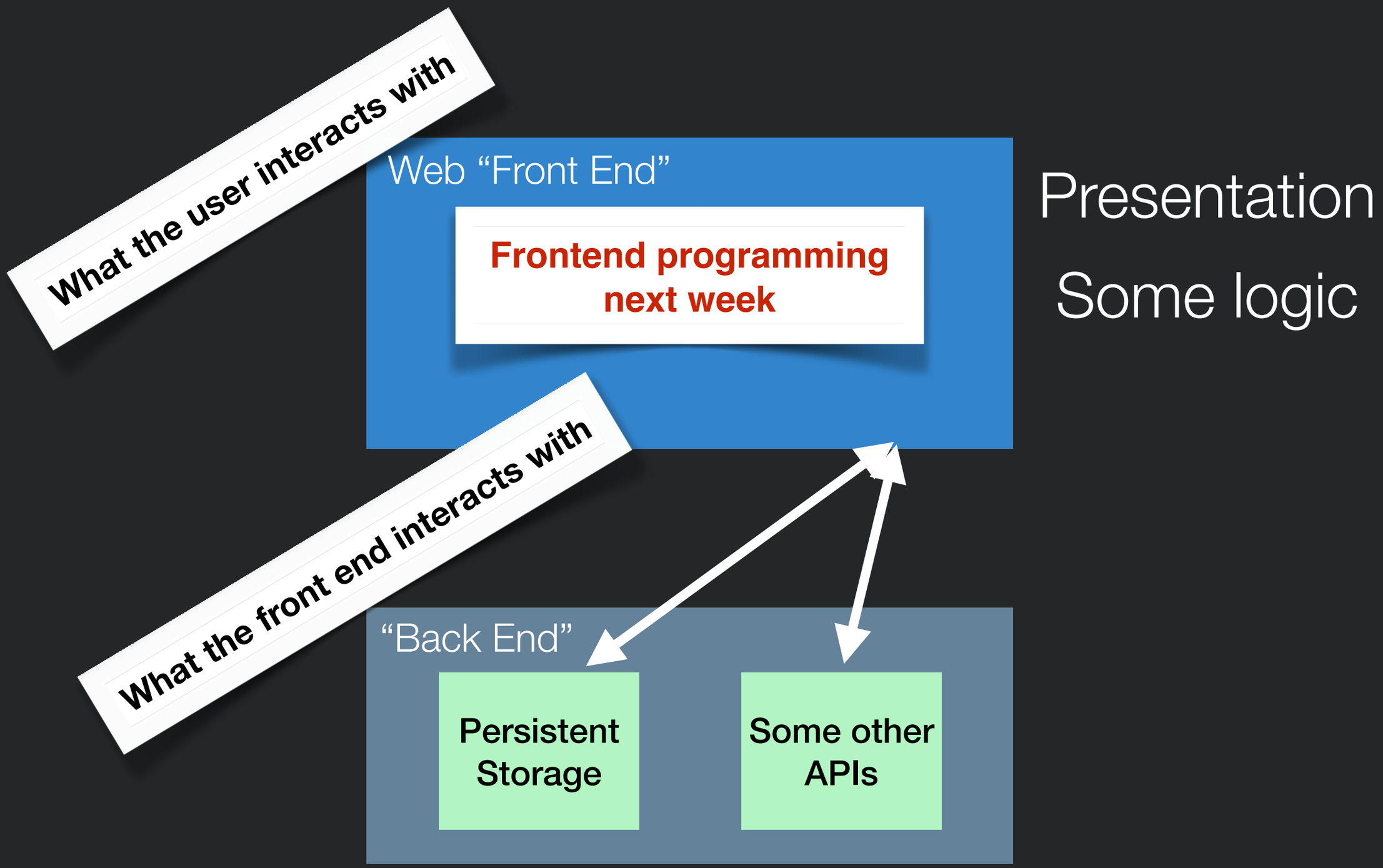
Persistent Storage

Some other APIs

Presentation

Some logic

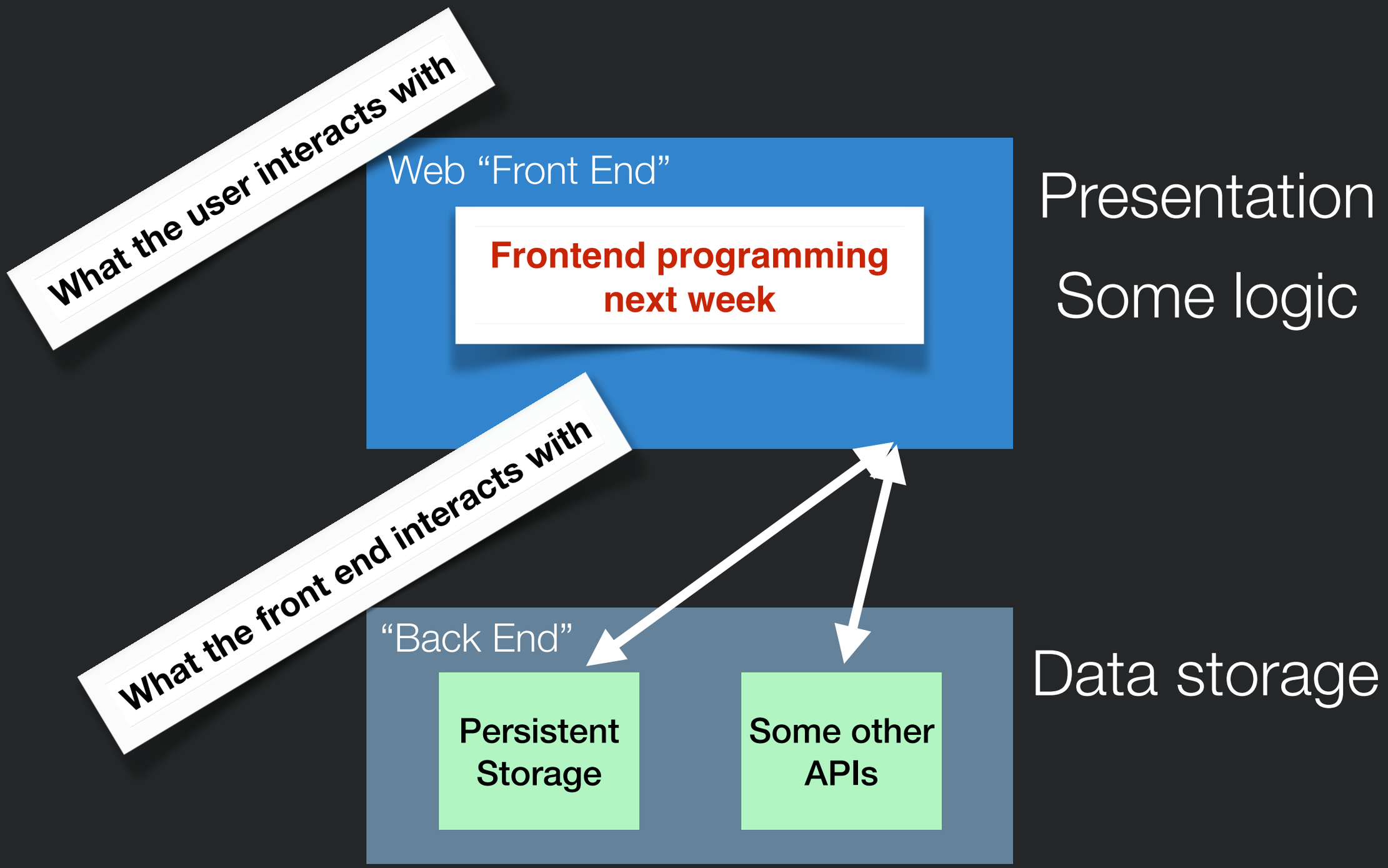
Dynamic Web Apps



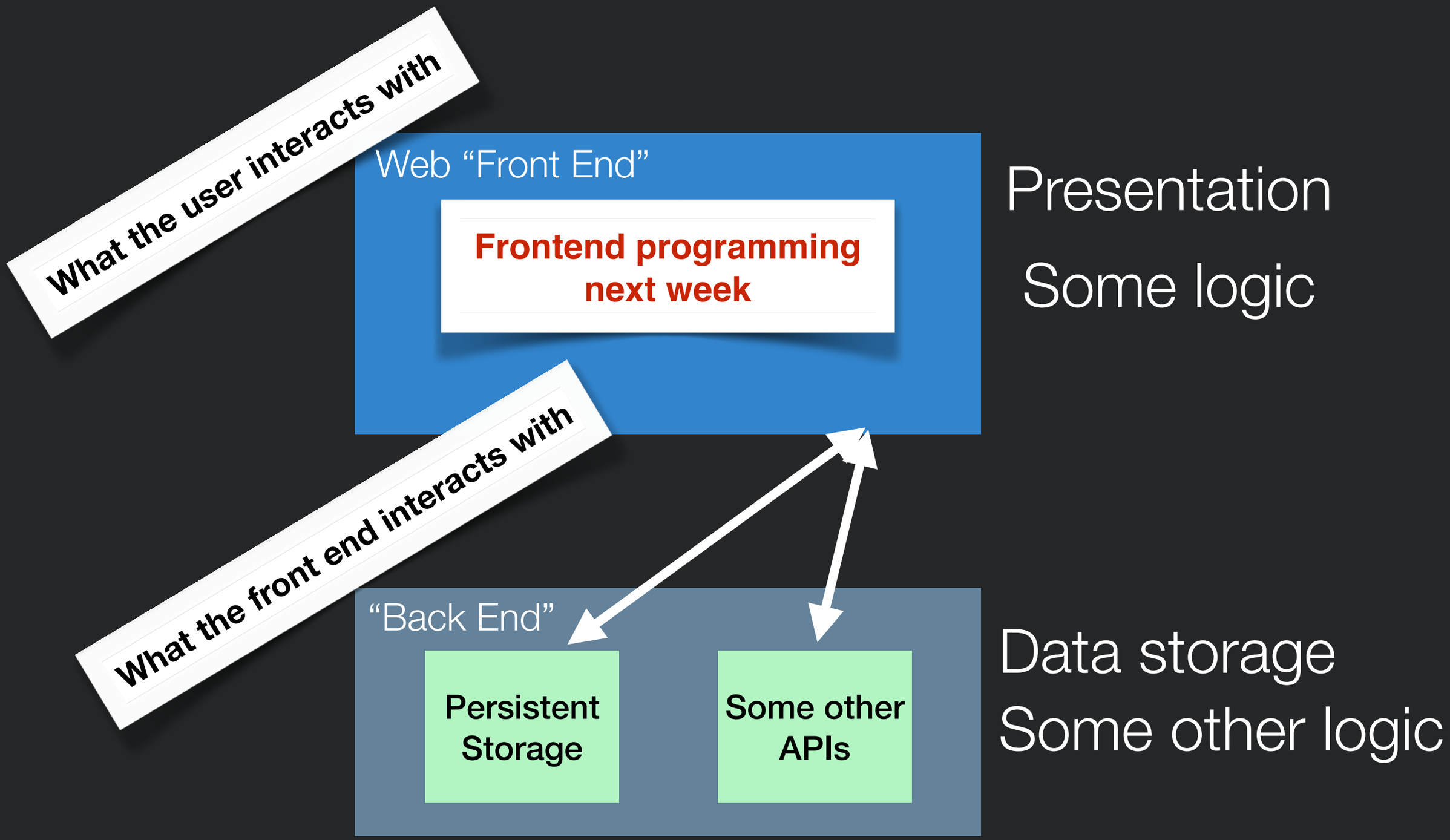
Presentation

Some logic

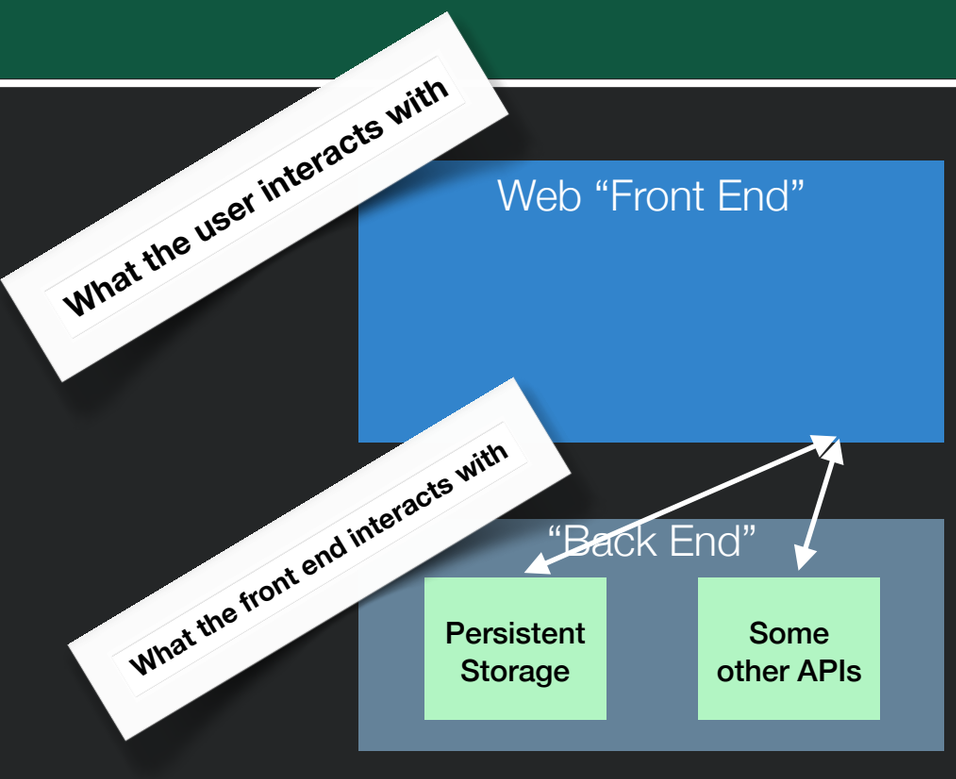
Dynamic Web Apps



Dynamic Web Apps



Where Do We Put the Logic?



Presentation

Some logic

Data storage

Some other logic

<u>Frontend</u> Pros

Very responsive (low latency)

<u>Frontend</u> Cons

Security

Performance

Unable to share between front-ends

<u>Backend</u> Pros

Easy to refactor between multiple clients

Logic is hidden from users (good for security, compatibility, etc.)

<u>Backend</u> 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:

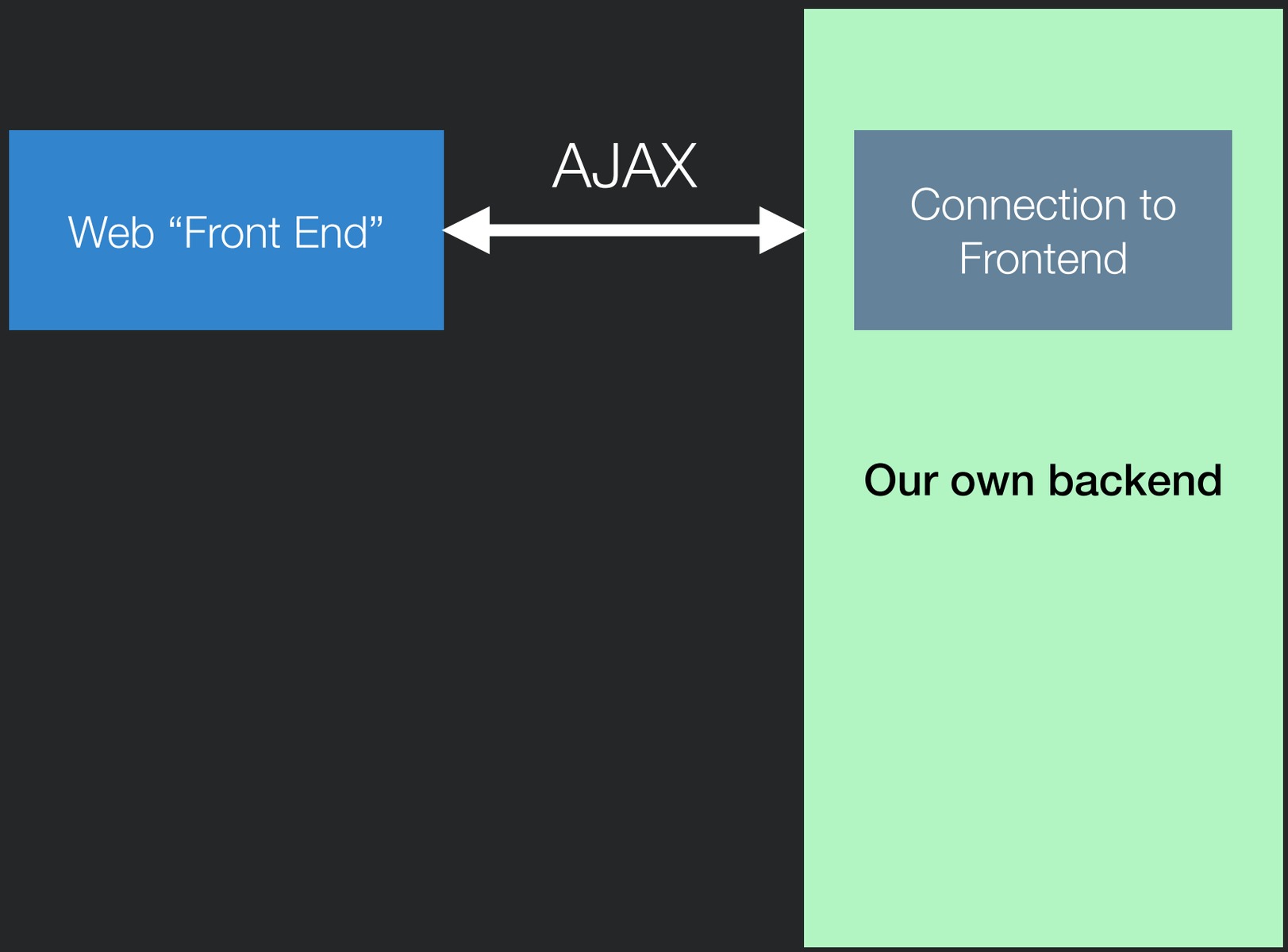
```
function updateBalance(user, amountToAdd)
{
    user.balance = user.balance + amountToAdd;
}
```

- What's wrong?
- How do you fix that?

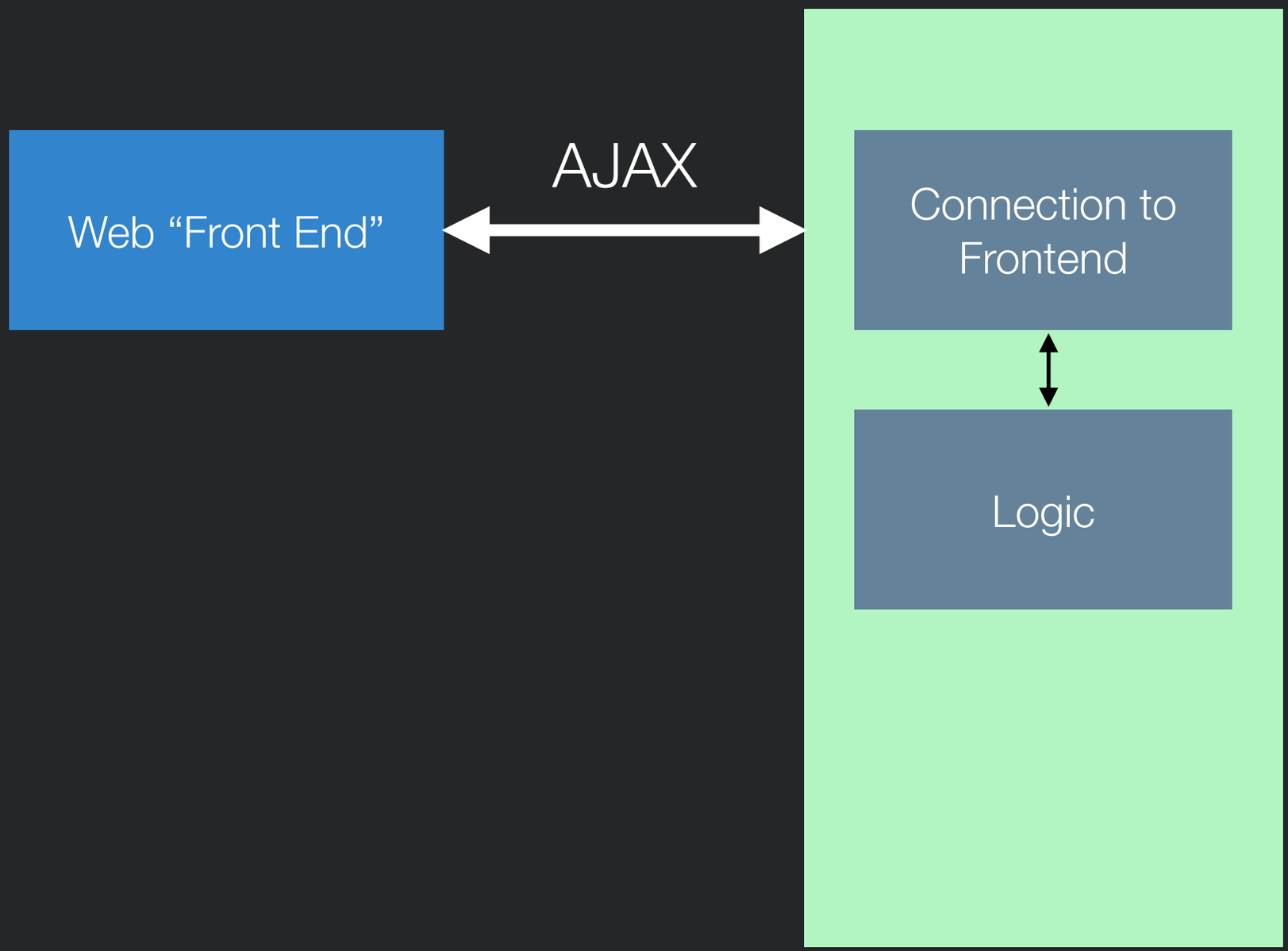
What Does our Backend Look Like?

Our own backend

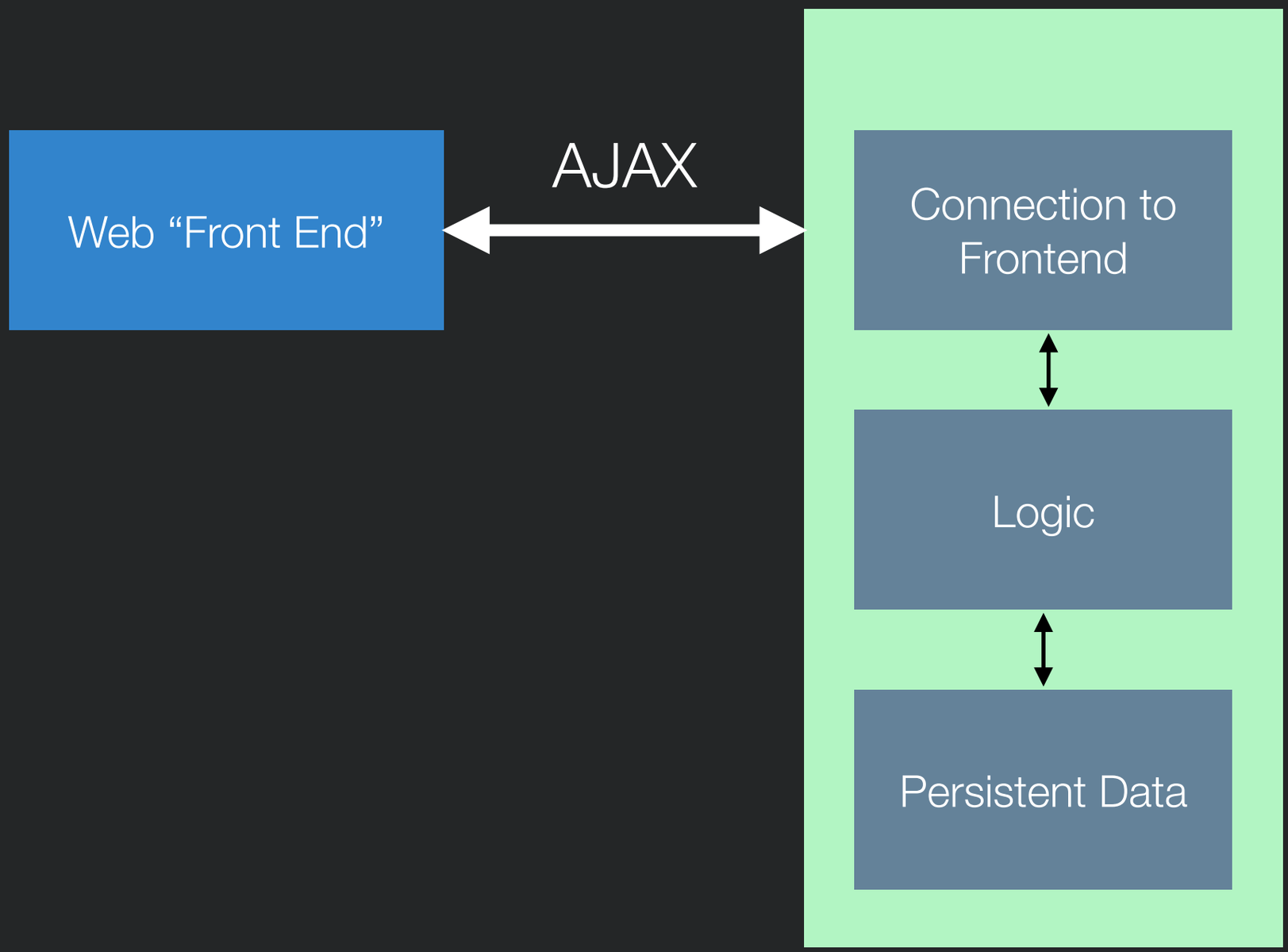
What Does our Backend Look Like?



What Does our Backend Look Like?



What Does our Backend Look Like?

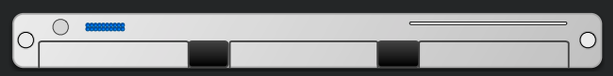


The "Good" Old Days of Backends

HTTP Request

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

web server



Runs a program

Web Server Application

My Application Backend

HTTP Response

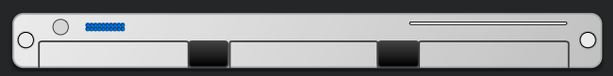
```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8

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

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

Give me **/myApplicationEndpoint**

Web Server Application

My Application Backend

HTTP Response

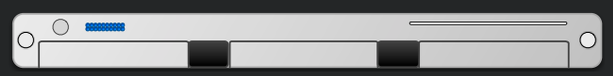
```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8

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

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

Give me **/myApplicationEndpoint**

Web Server Application

Does whatever it wants

My Application Backend

HTTP Response

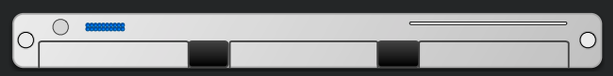
```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8

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

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

My Application Backend

Here's some text to send back

HTTP Response

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

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

What's wrong with this picture?

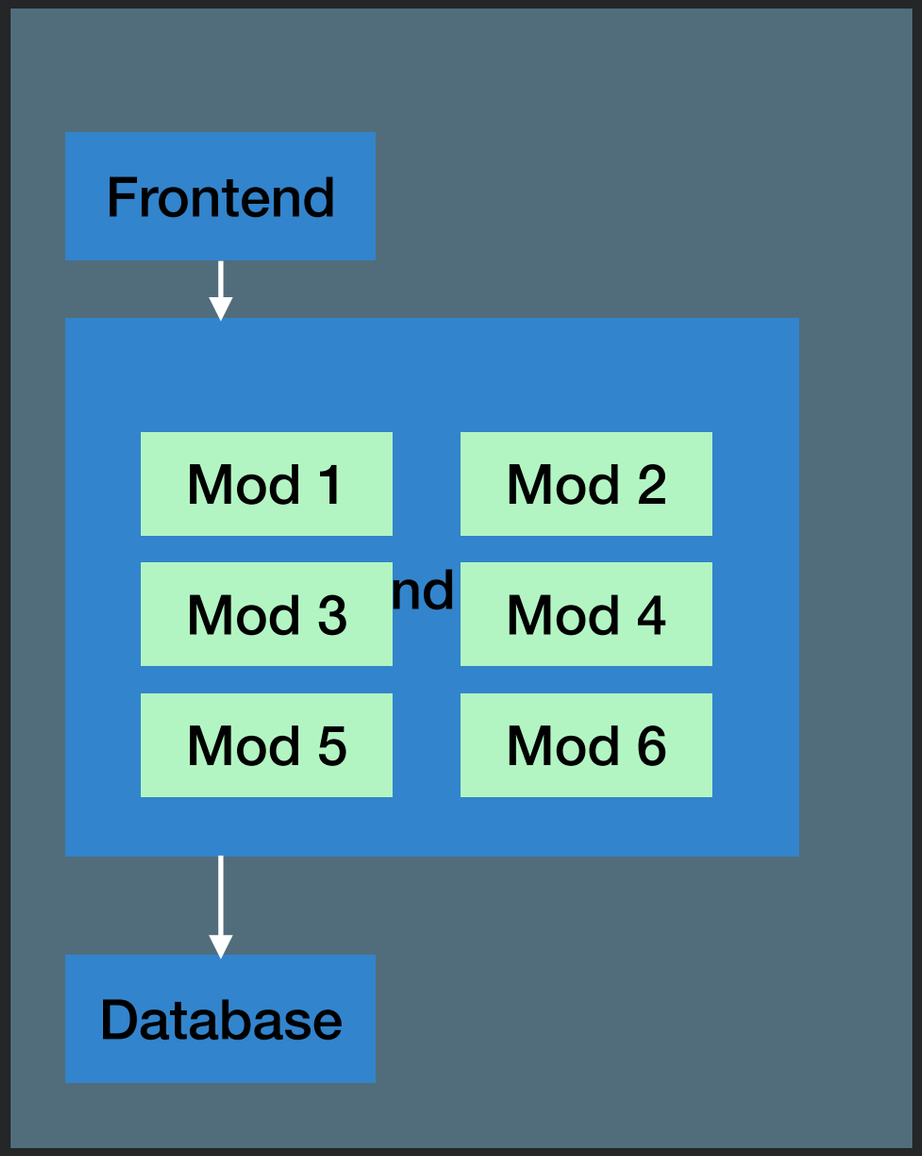
History of Backend Development

- 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

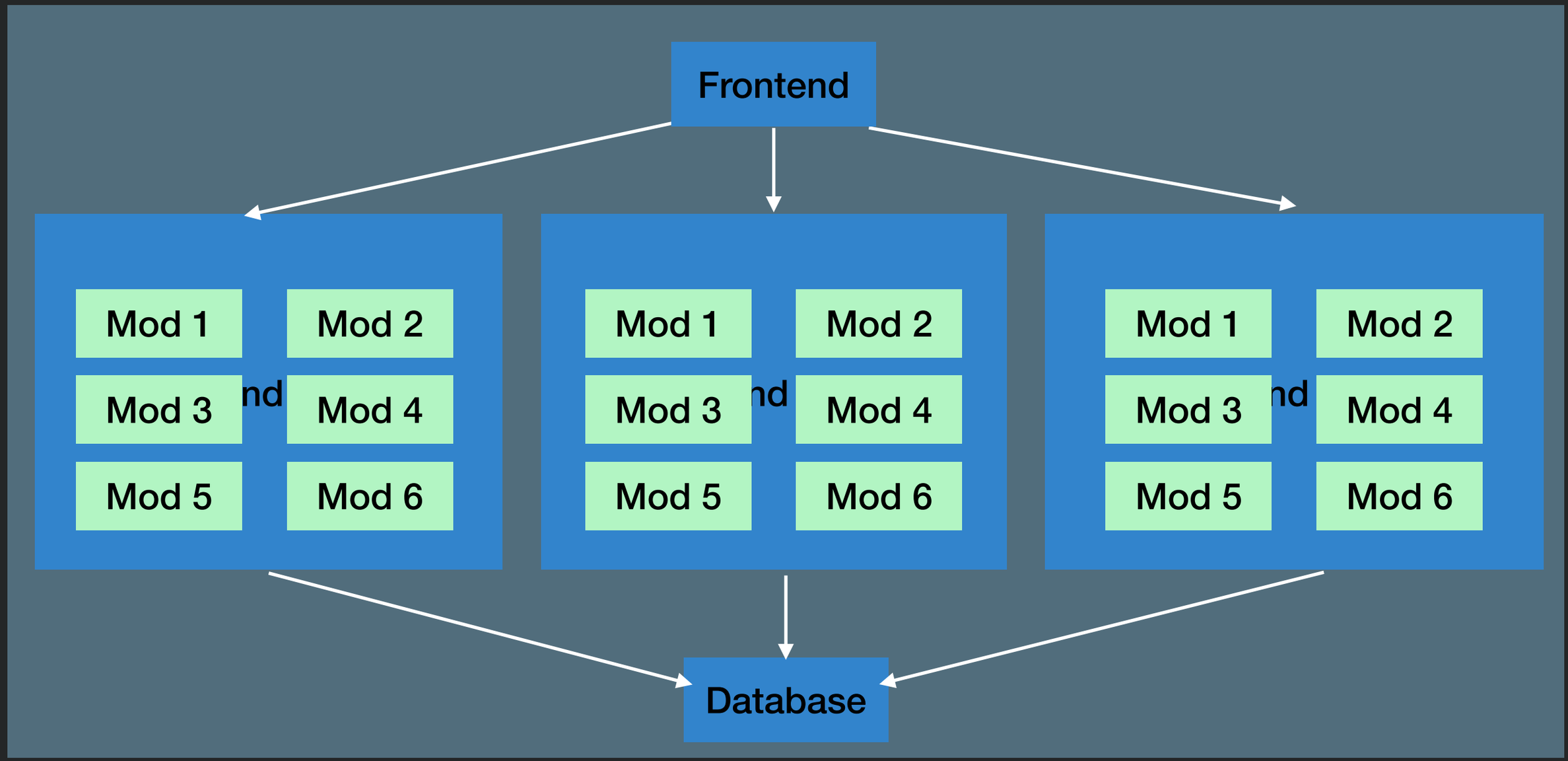
Microservices vs. Monoliths

- 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

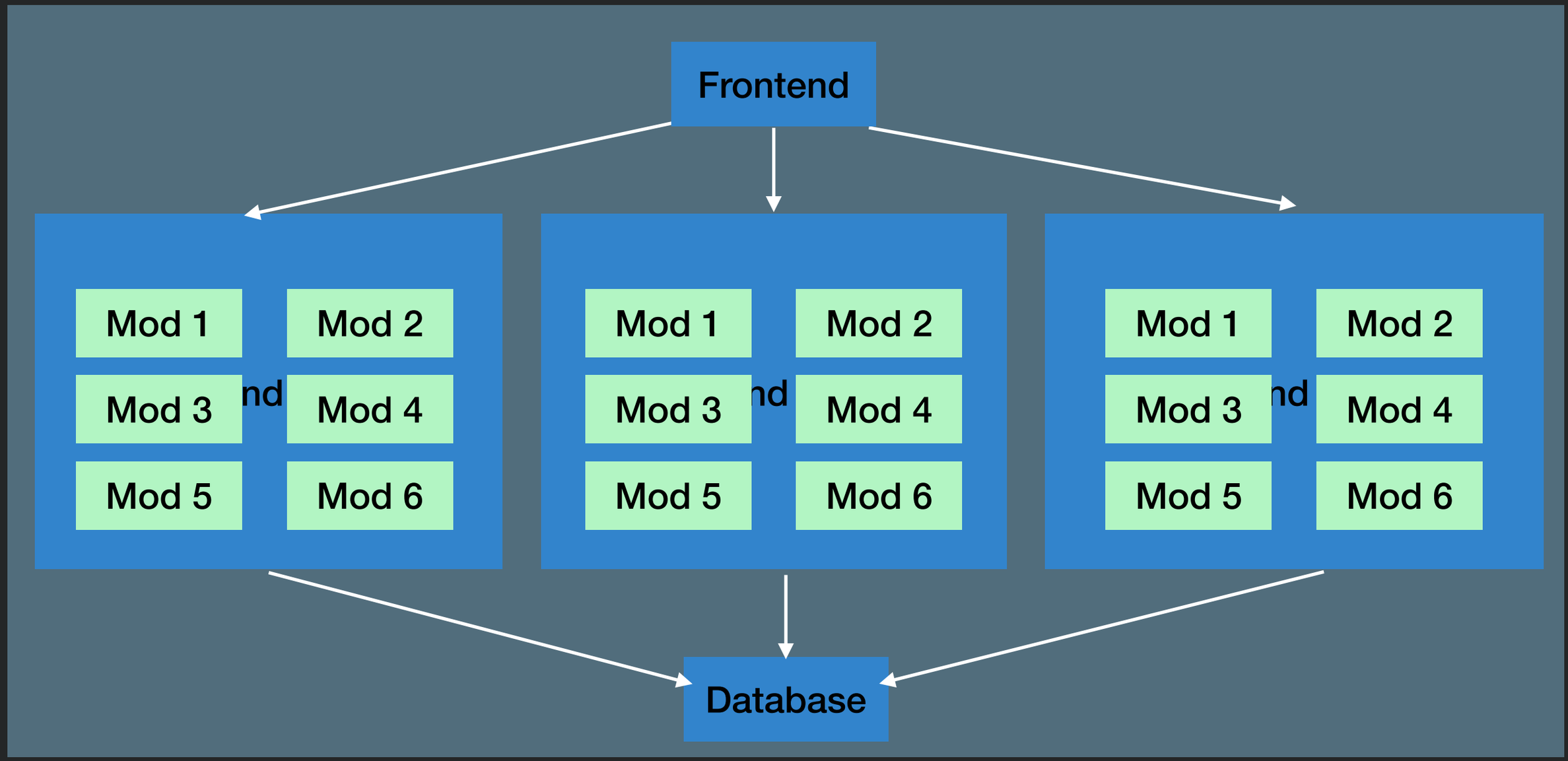
Support for Scaling



Now How Do We Scale It?



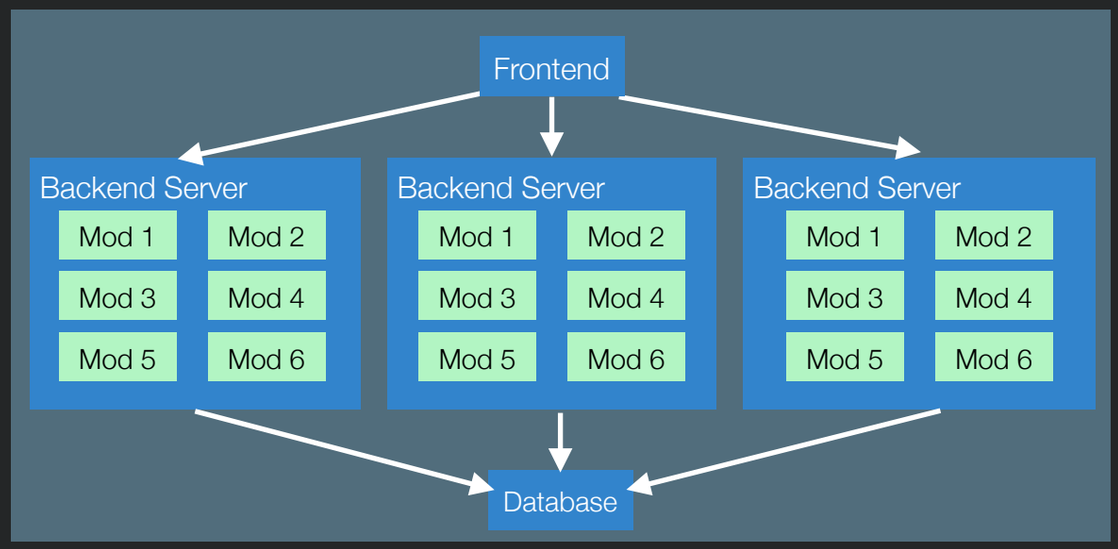
Now How Do We Scale It?



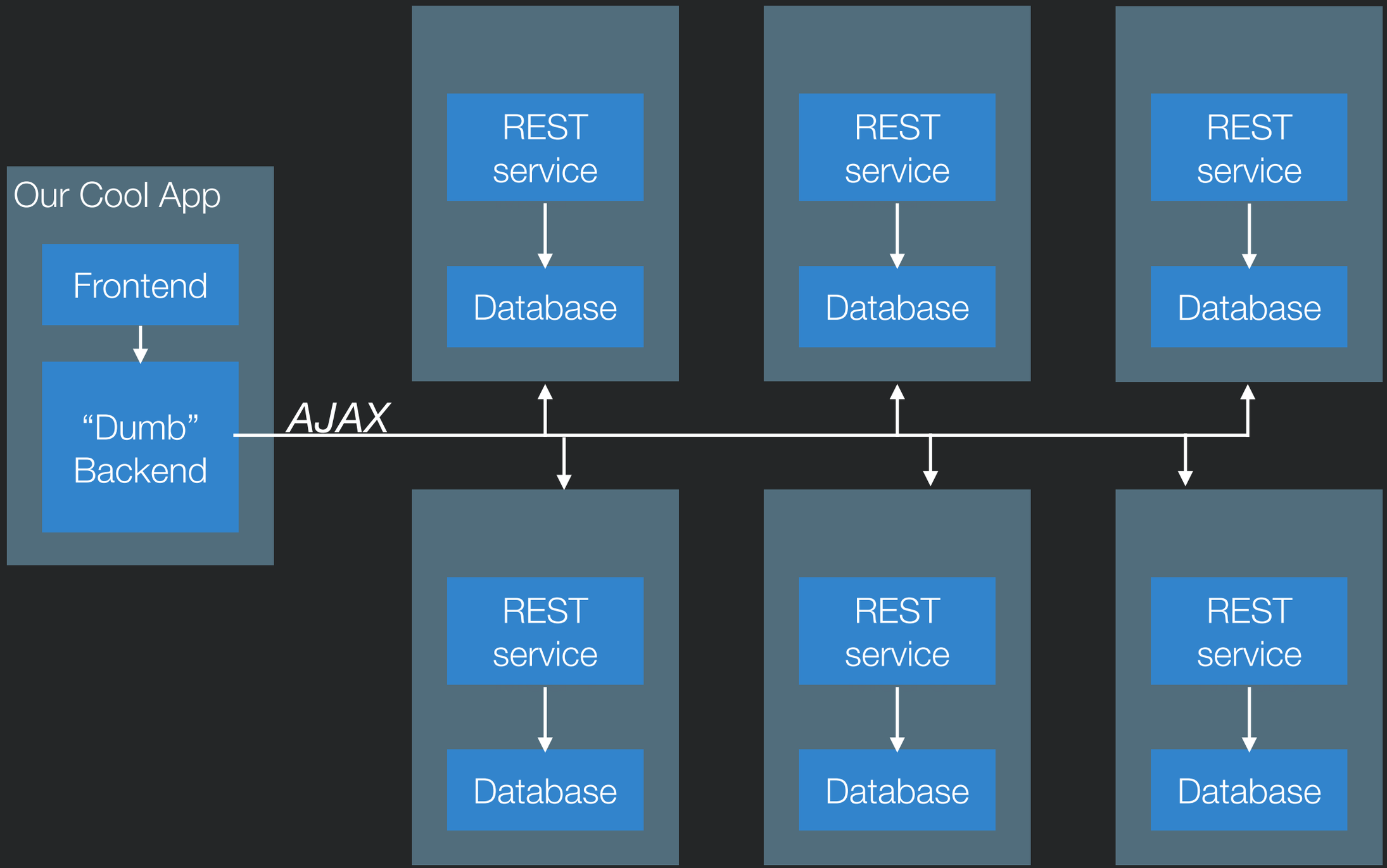
We run multiple copies of the backend, each with each of the modules

What's wrong with this picture?

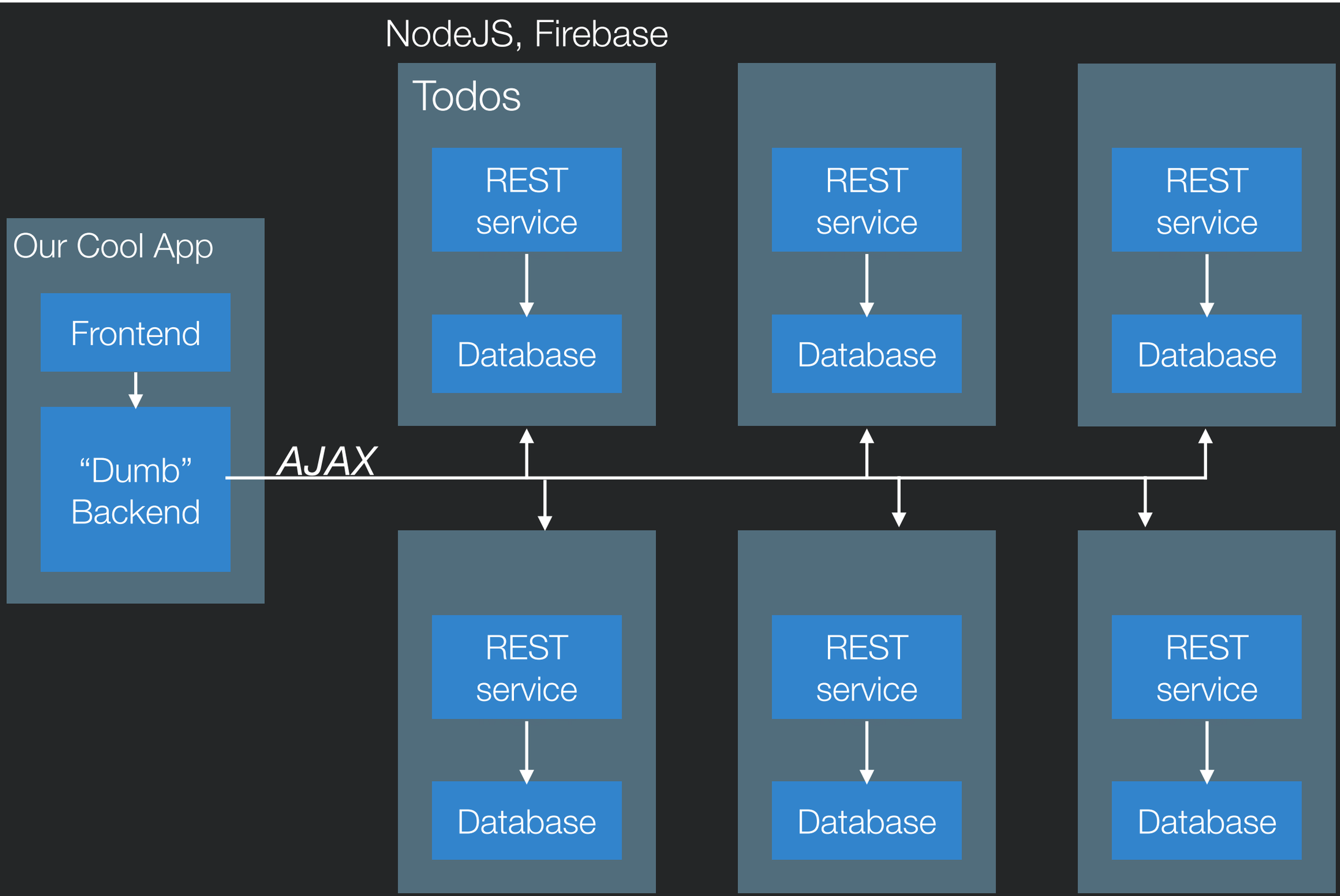
- This is called the "monolithic" app
- If we need 100 servers…
- Each server will have to run EACH module
- What if we need more of some modules than others?



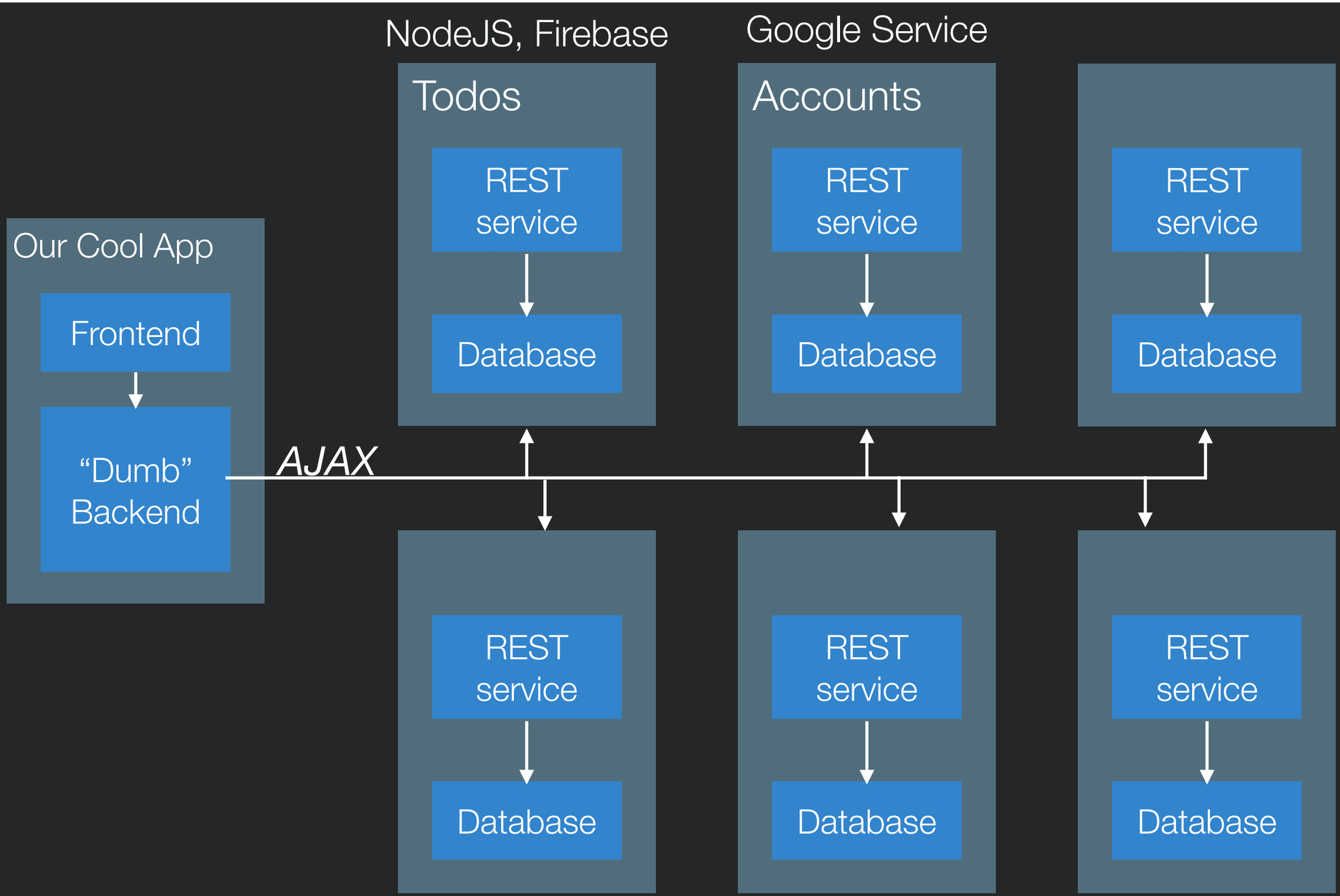
Microservices



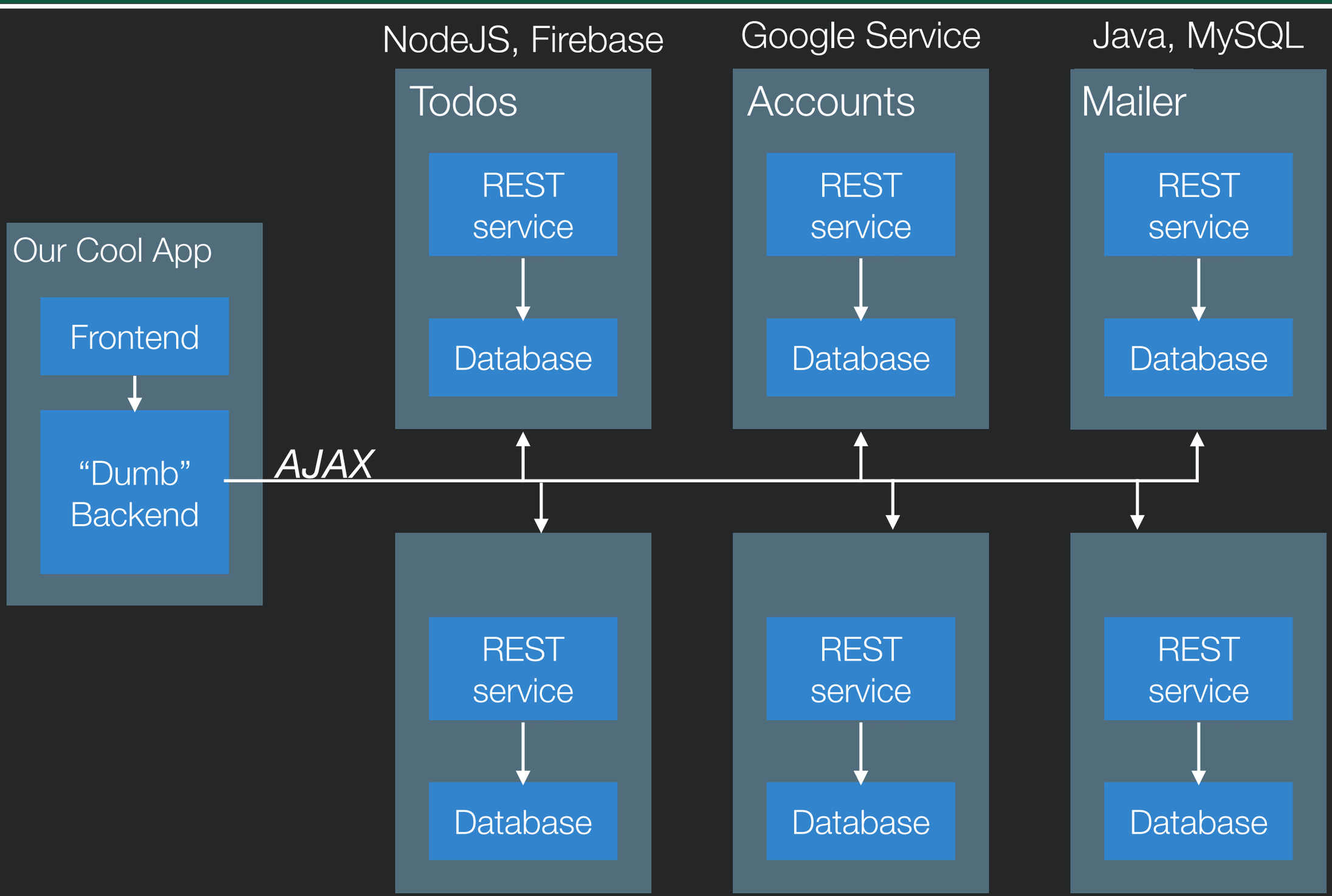
Microservices



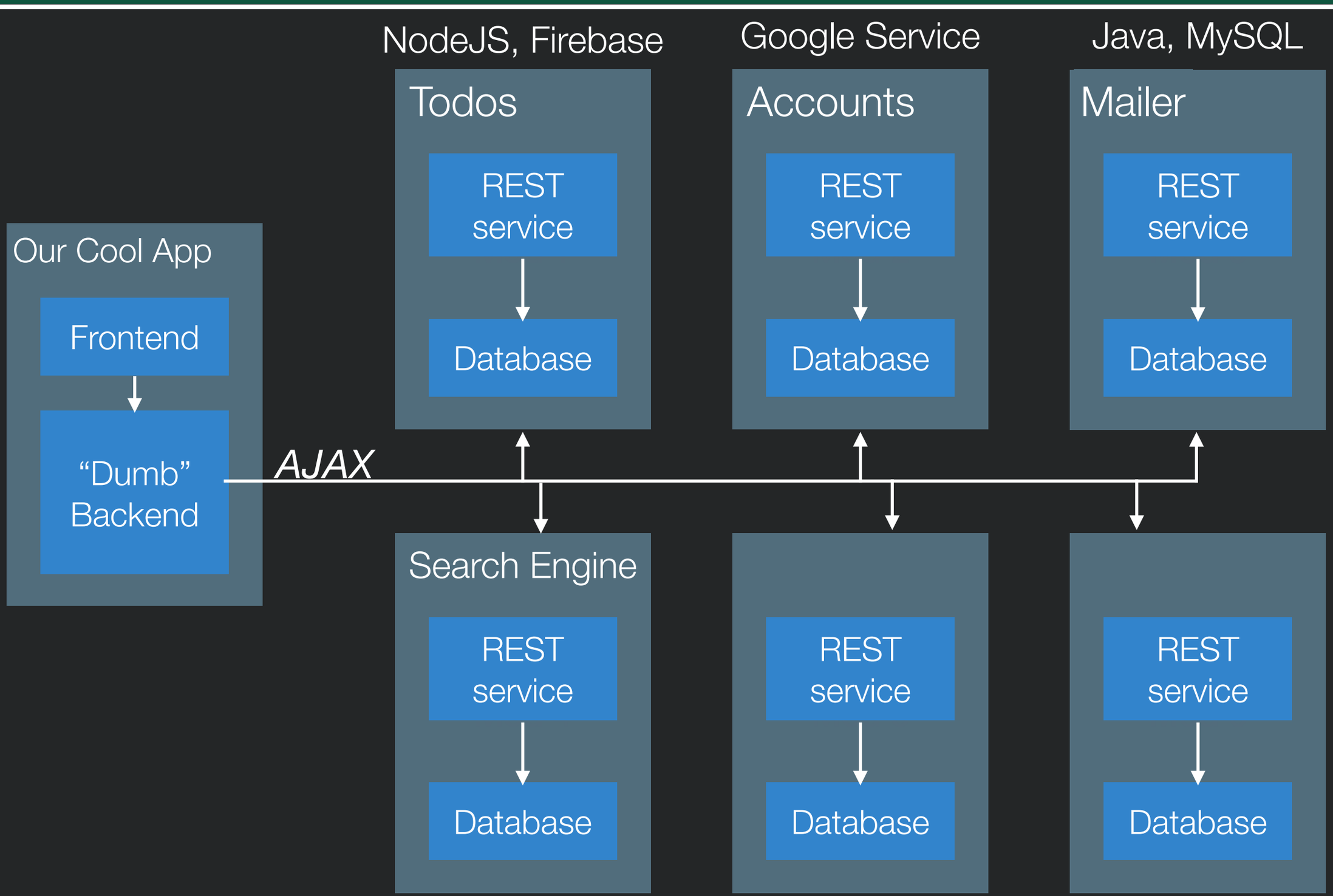
Microservices



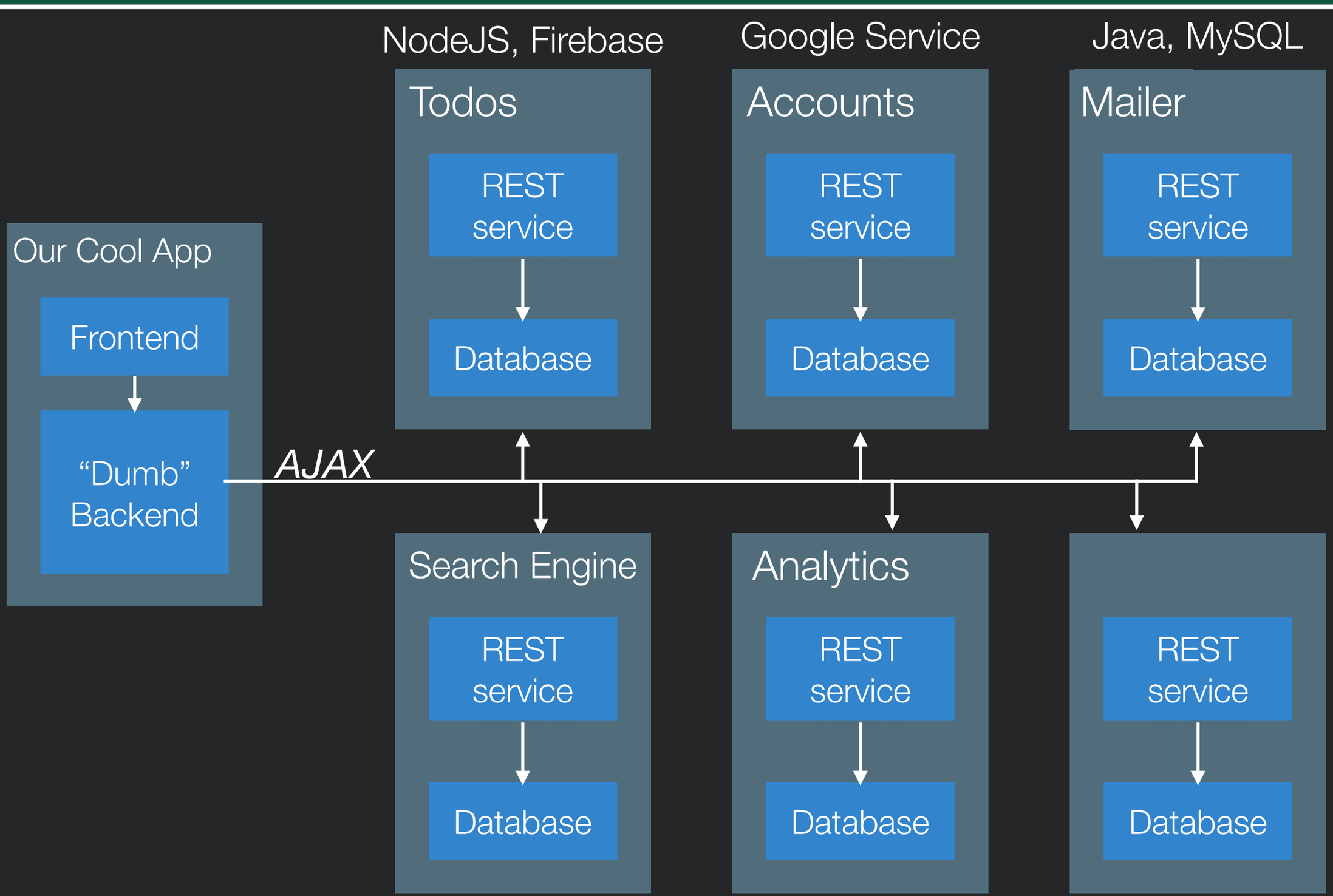
Microservices



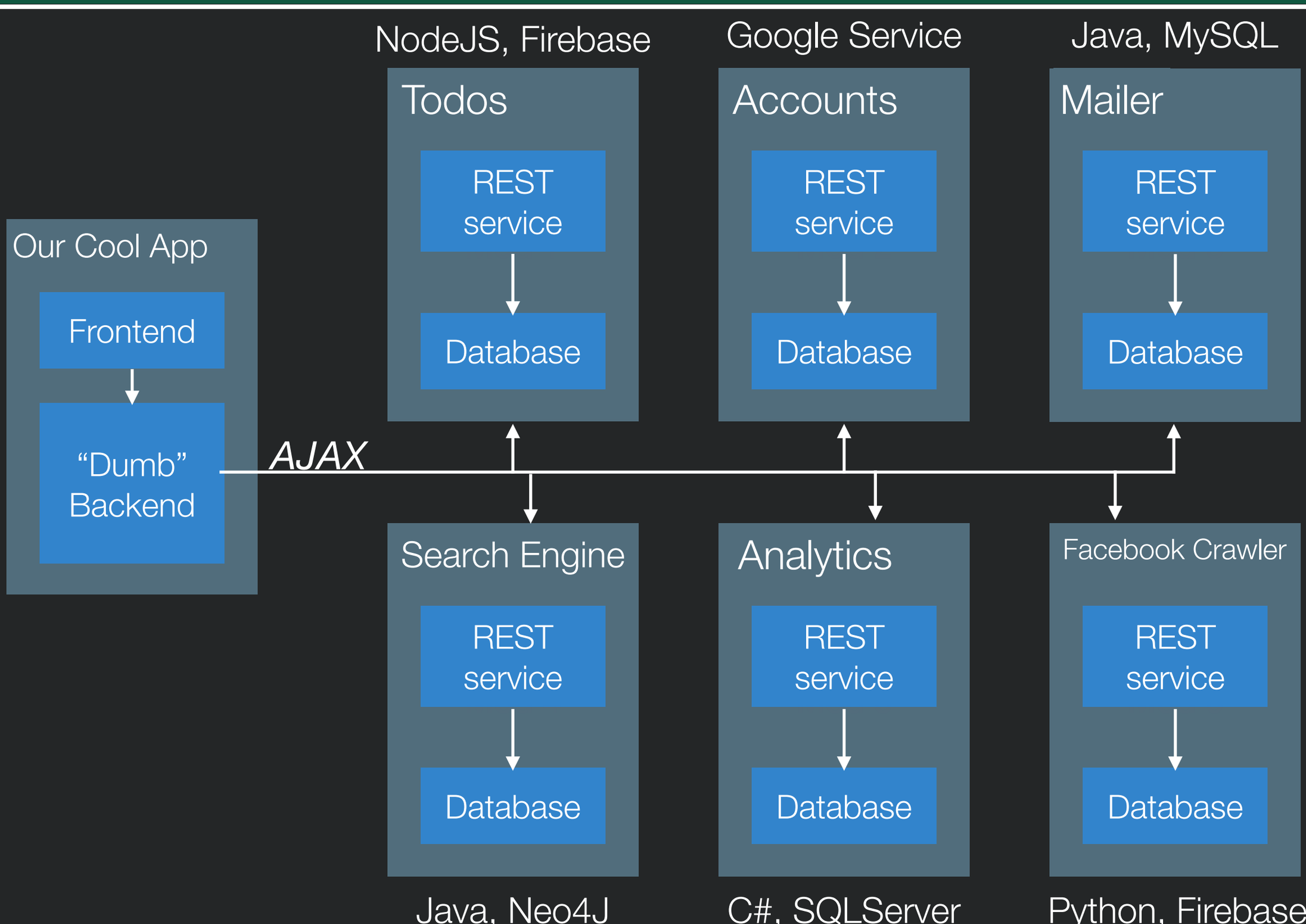
Microservices



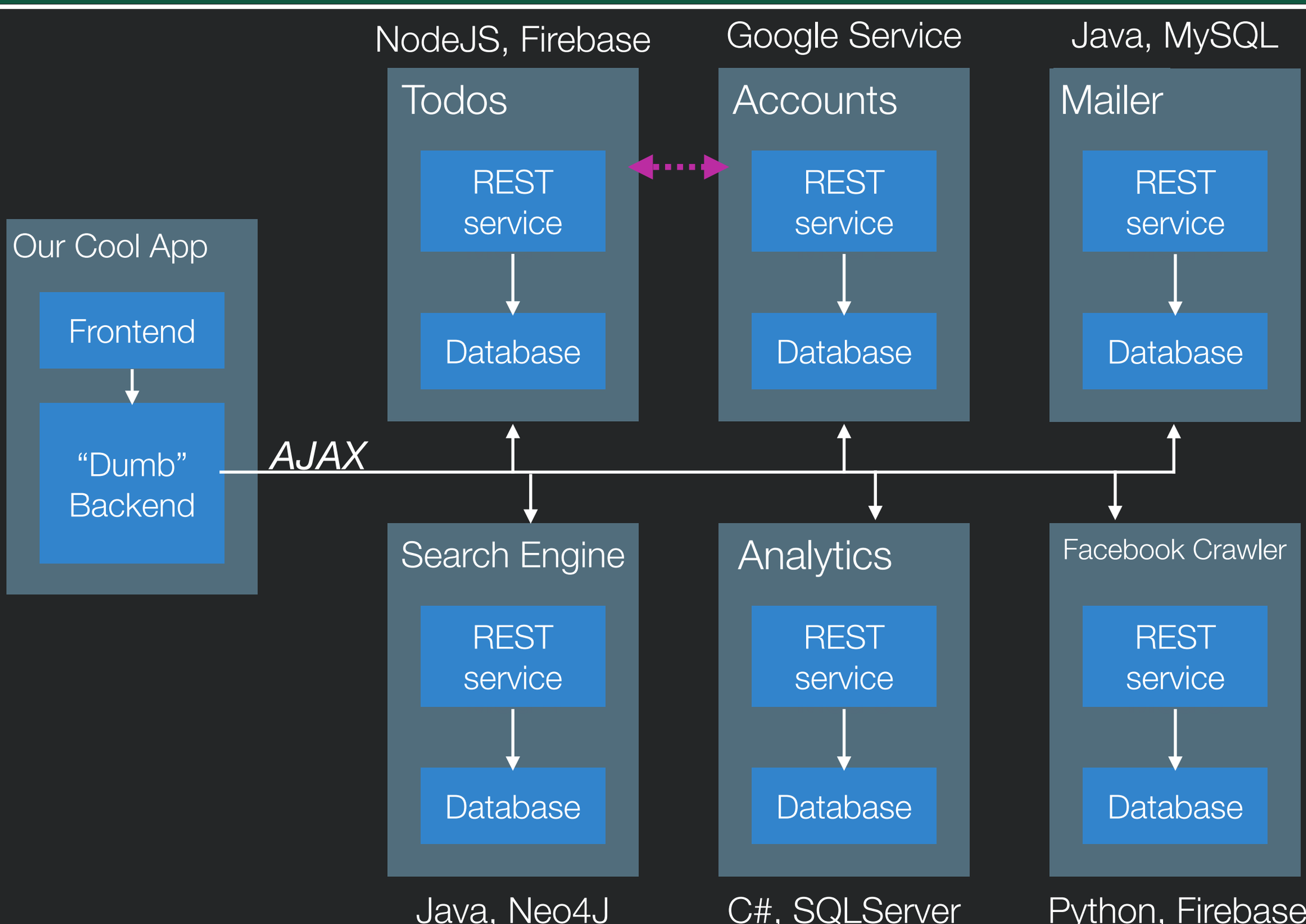
Microservices



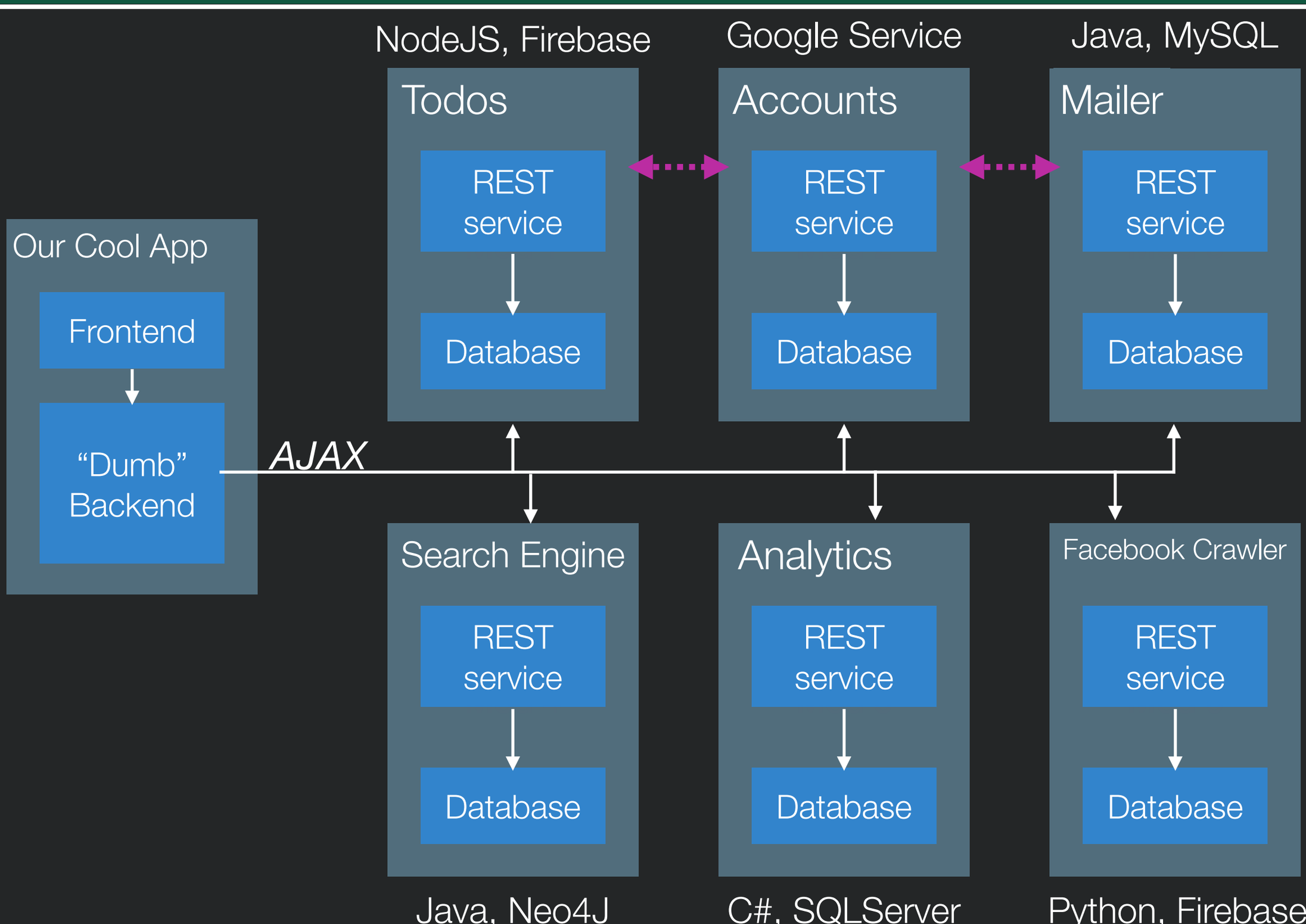
Microservices



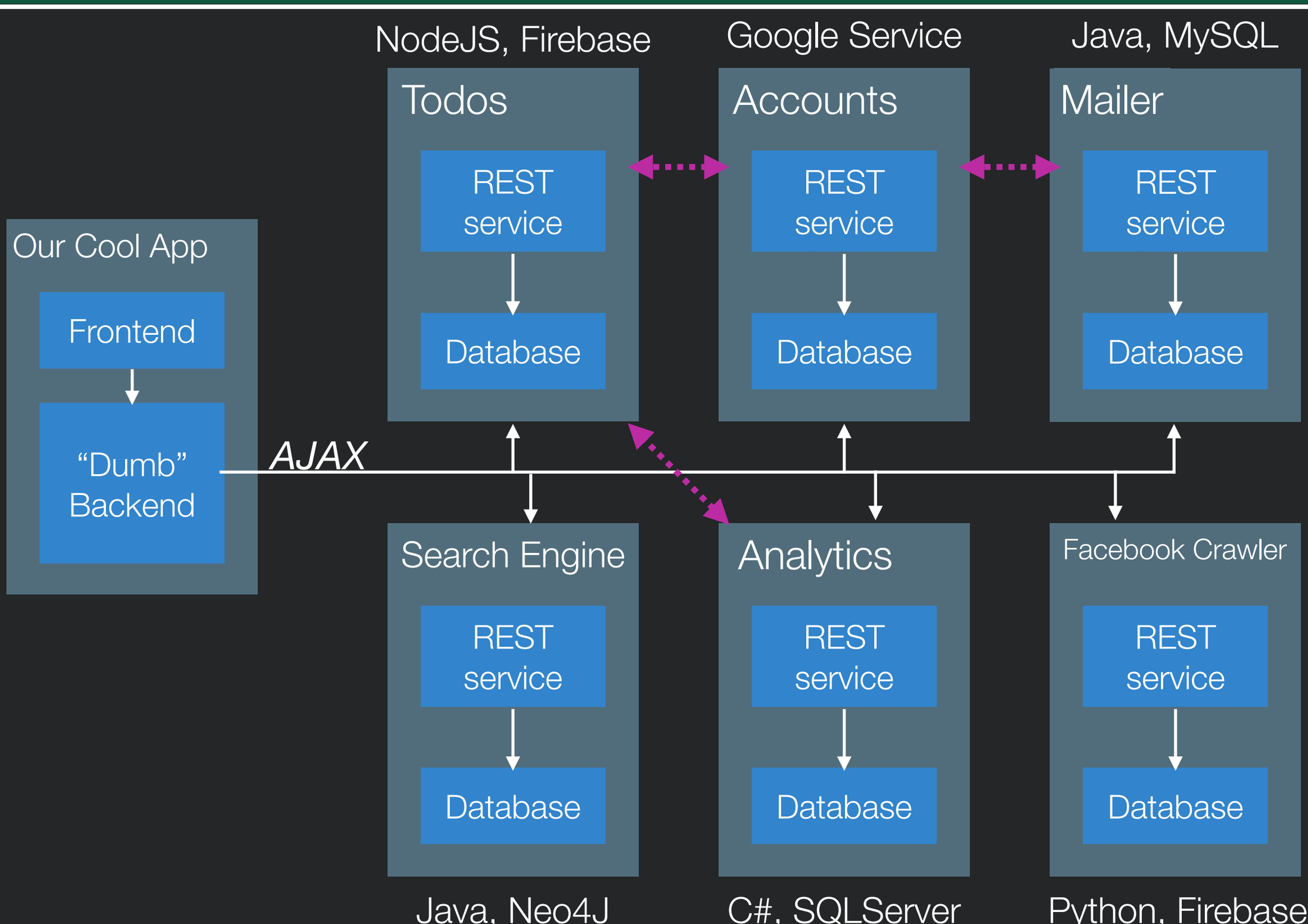
Microservices



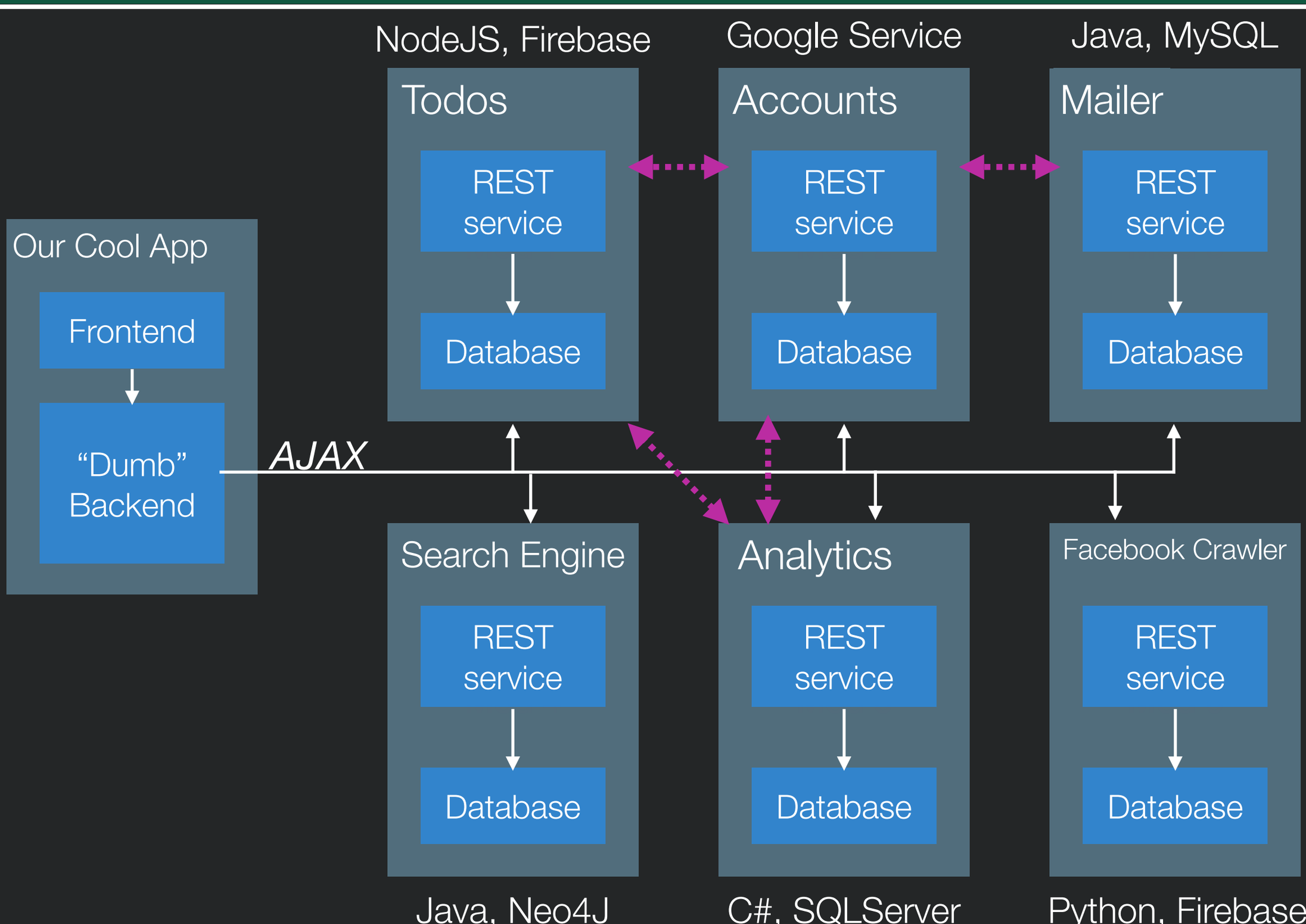
Microservices



Microservices



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

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

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`

Creates a configuration file for your project

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

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`:

```
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);
});
```

Creates a configuration file for your project

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

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`:

```
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

Creates a configuration file for your project

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

Runs your app

Demo: Hello World Server

```
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);
});
```

Demo: Hello World Server

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

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;


app.get('/', function (req, res) {
  res.send('Hello World!');
});



app.listen(port, function () {
  console.log('Example app listening on port' + port);
});
```

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!');
});



app.listen(port, function () {
  console.log('Example app listening on port' + port);
});
```

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);
});
```

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)
- Message body only allowed with certain response status codes

```
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>
```

Response status codes:
1xx Informational
2xx Success
3xx Redirection
4xx Client error
5xx Server error

Common MIME types:
application/json
application/pdf
image/png

HTTP Responses

- Larger number of response codes (200 OK, 404 NOT FOUND)
- Message body only allowed with certain response status codes

```
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>
```

[HTML data]

"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 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));
```

SWE 432 - Web Application Development

George Mason University

Instructor:
Dr. Kevin Moran

Teaching Assistant:
David Gonzalez Samudio

Class will start in:

10:01

SWE 432 - Web Application Development

George Mason University

Instructor:
Dr. Kevin Moran

Teaching Assistant:
David Gonzalez Samudio

Class will start in:

10:01

Handling HTTP Requests

Quiz

Go to:
b.socrative.com, Click student login
Room name: SWE432
Student Name: Your G-number (Including the G)

Reminder: Survey can only be completed if you are in class. If you are not in class and do it you will be referred directly to the honor code board, no questions asked, no warning.

Review: Express

```
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);
});
```

Review: Express

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

Review: Express

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

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

Review: Express

```
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!');
});



app.listen(port, function () {
  console.log('Example app listening on port' + port);
});
```

Review: Express

```
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);
});
```

Review: Express

```
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

Review: 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);
});
```

Review: 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

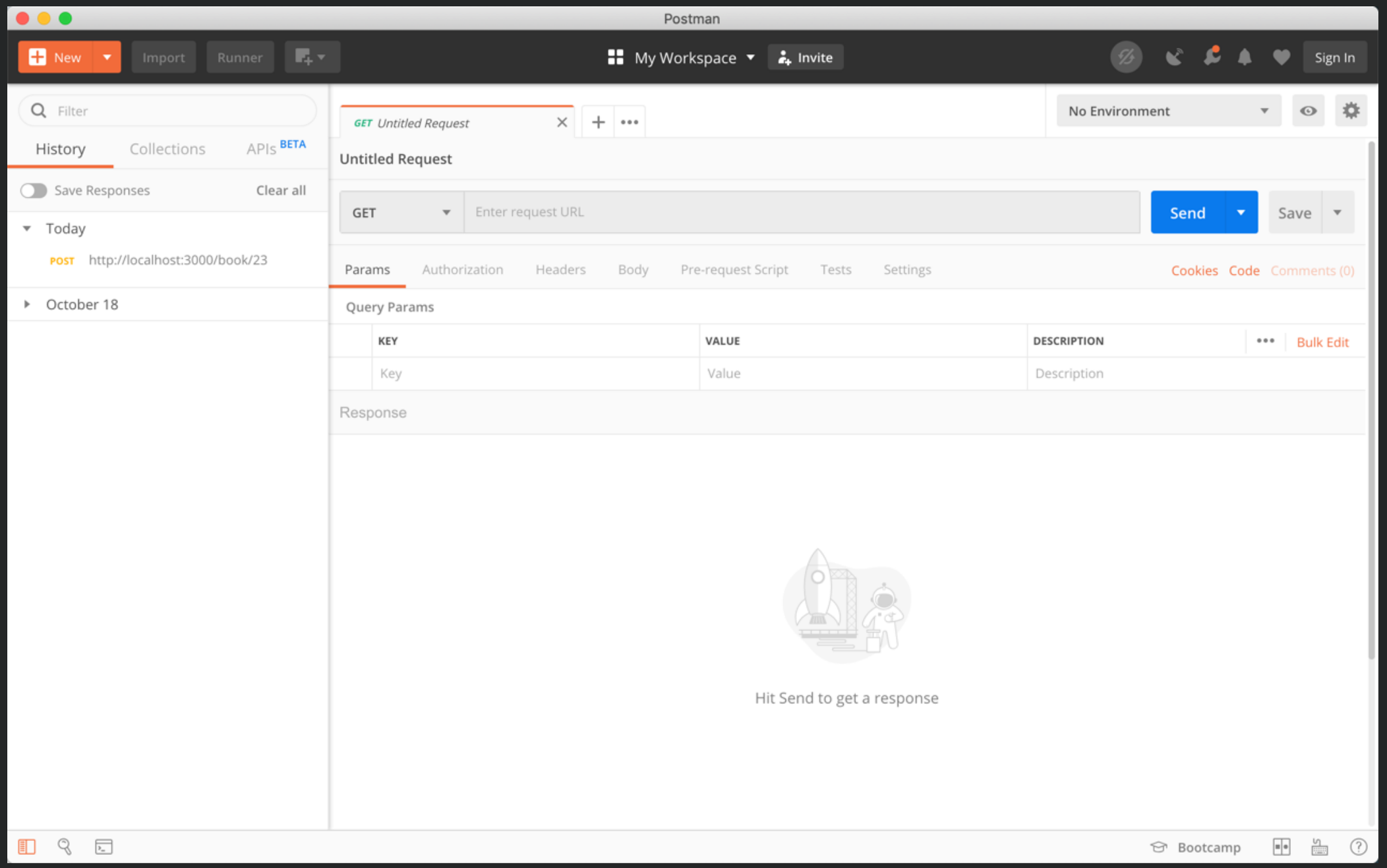
Today

- Design considerations in identifying resources
- REST
  - What is it?
  - Why use it?

Demo: Using fetch to post data

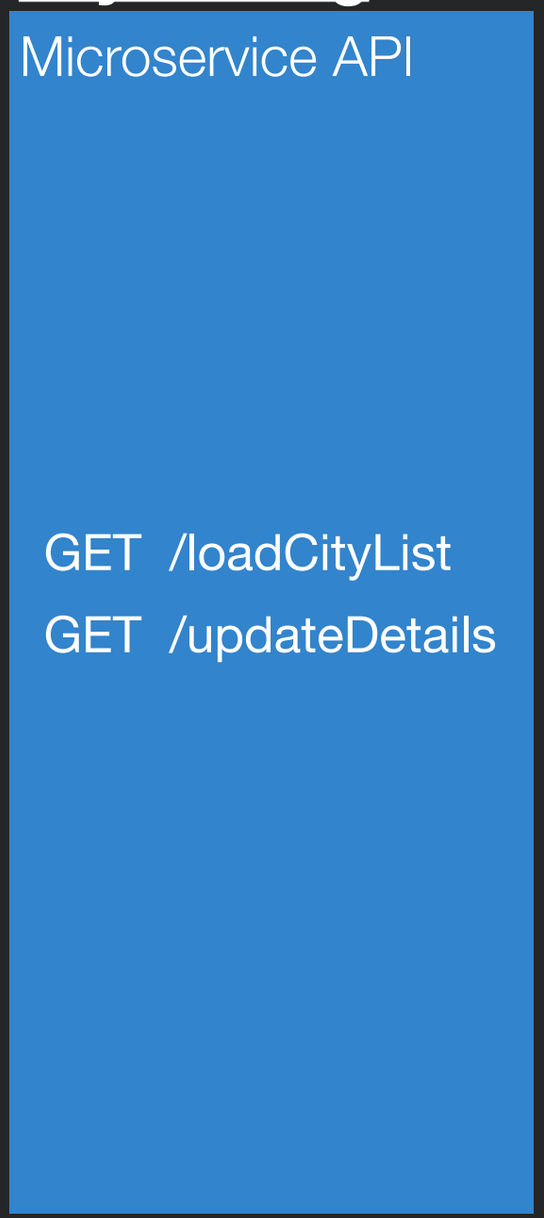
```
var express = require('express');
var app = express();
const fetch = require('node-fetch');

const body = { 'a': 1 };

fetch('http://localhost:3000/book/23', {
    method: 'post',
    body:    JSON.stringify(body),
    headers: { 'Content-Type': 'application/json' },
})
    .then(res => res.json())
    .then(json => console.log(json));
```

Demo: Making HTTP Request with Postman



https://www.getpostman.com/

Demo: Building a Microservice w/ Express



API: Application Programming Interface

cityinfo.org

Microservice API

GET /loadCityList

GET /updateDetails

- Microservice offers public **interface** for interacting with backend
  - Offers abstraction that hides implementation details
  - Set of endpoints exposed on micro service
- Users of API might include
  - Frontend of your app
  - Frontend of other apps using your backend
  - Other servers using your service

APIs for Functions and Classes

V1

```
function sort(elements)
{
    [sort algorithm A]
}
```

```
class Graph
{
    [rep of Graph A]
}
```

Implementation change

Consistent interface

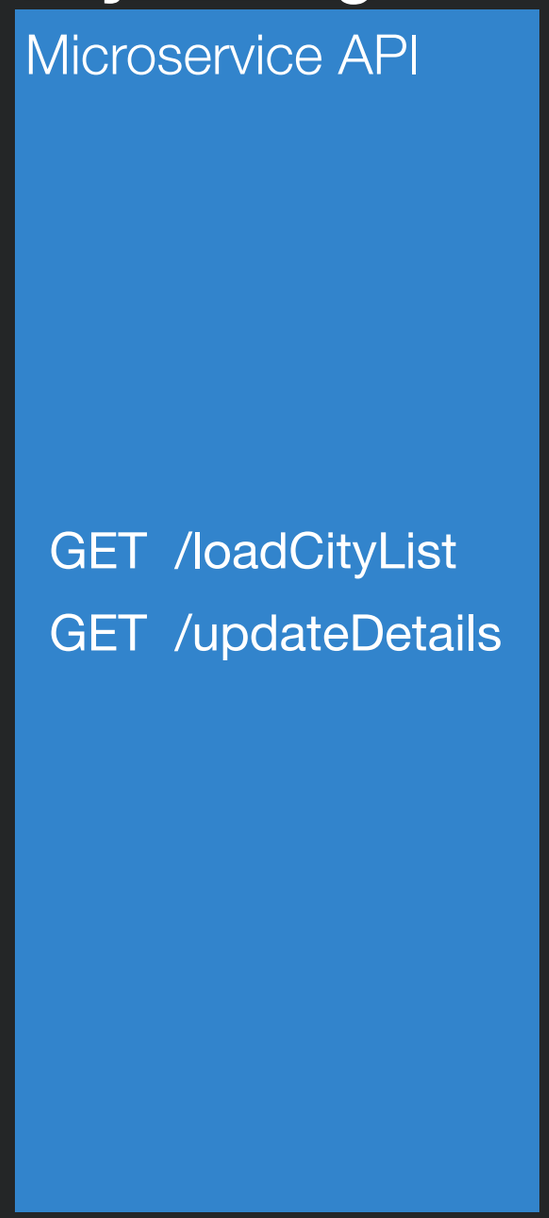
V2

```
function sort(elements)
{
    [sort algorithm B]
}
```

```
class Graph
{
    [rep of Graph B]
}
```

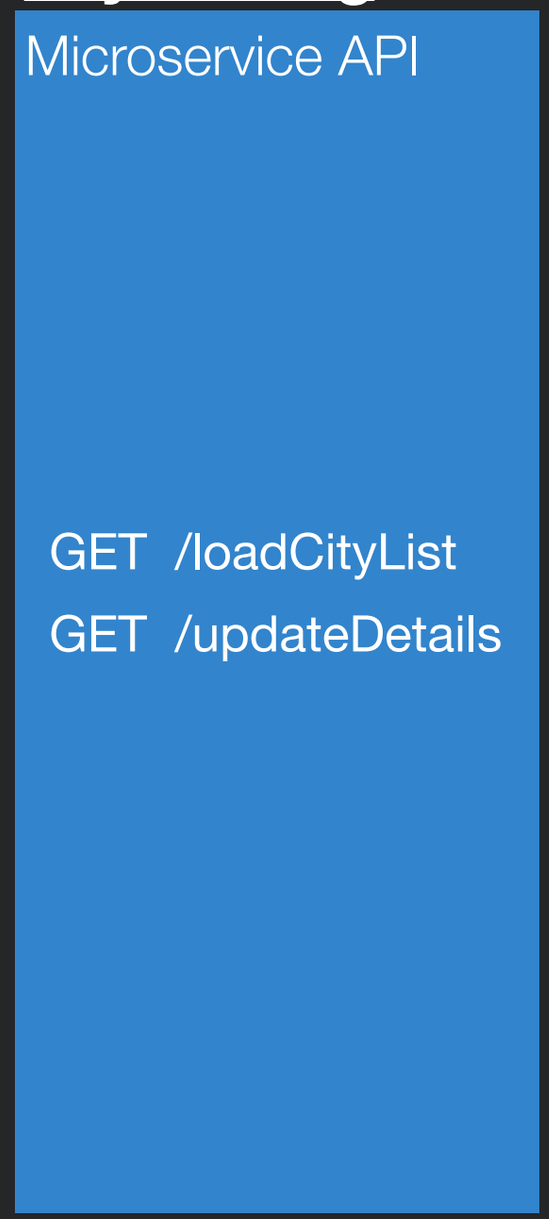
Support Scaling

- Yesterday, cityinfo.org had 10 daily active users. Today, it was featured on several news sites and has 10,000 daily active users.
- Yesterday, you were running on a single server. Today, you need more than a single server.
- Can you just add more servers?
  - What should you have done yesterday to make sure you can scale quickly today?



Support Change

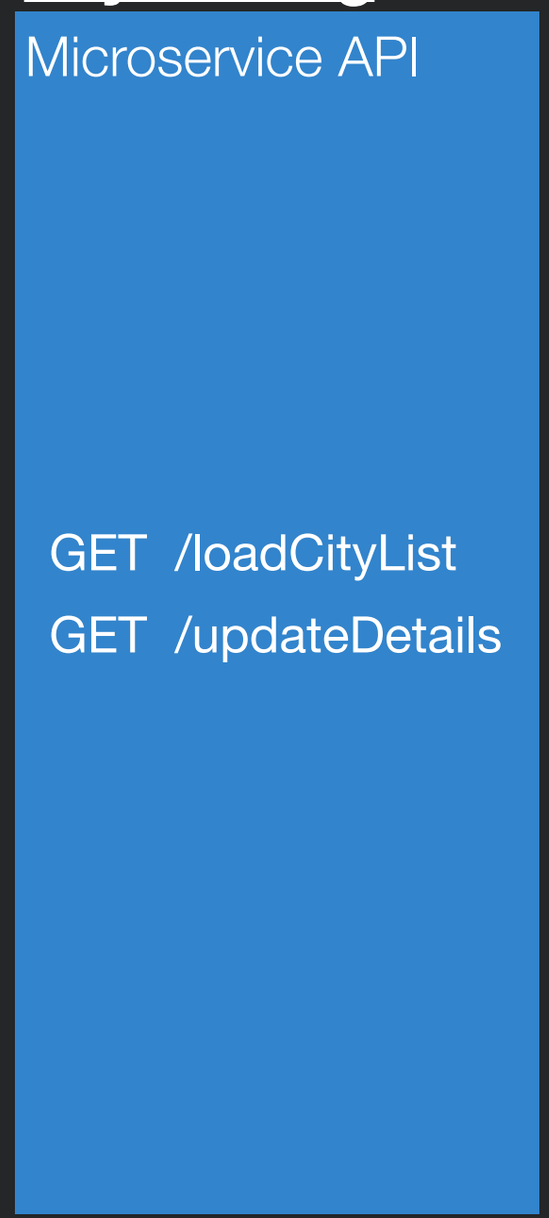
- Due to your popularity, your backend data provider just backed out of their contract and are now your competitor.
- The data you have is now in a different format.
- Also, you've decided to migrate your backend from PHP to node.js to enable better scaling.
- How do you update your backend without breaking all of your clients?



Support Reuse

- You have your own frontend for cityinfo.org. But everyone now wants to build their own sites on top of your city analytics.

- Can they do that?



Design Considerations for Microservice APIs

- API: What requests should be supported?
- Identifiers: How are requests described?
- Errors: What happens when a request fails?
- Heterogeneity: What happens when different clients make different requests?
- Caching: How can server requests be reduced by caching responses?
- Versioning: What happens when the supported requests change?

REST: REpresentational State Transfer

- Defined by Roy Fielding in his 2000 Ph.D. dissertation
  - Used by Fielding to design HTTP 1.1 that generalizes URLs to URIs
  - http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf
- *“Throughout the HTTP standardization process, I was called on to defend the design choices of the Web. That is an extremely difficult thing to do… I had comments from well over 500 developers, many of whom were distinguished engineers with decades of experience. That process honed my model down to a core set of principles, properties, and constraints that are now called REST.”*
- Interfaces that follow REST principles are called RESTful

Properties of REST

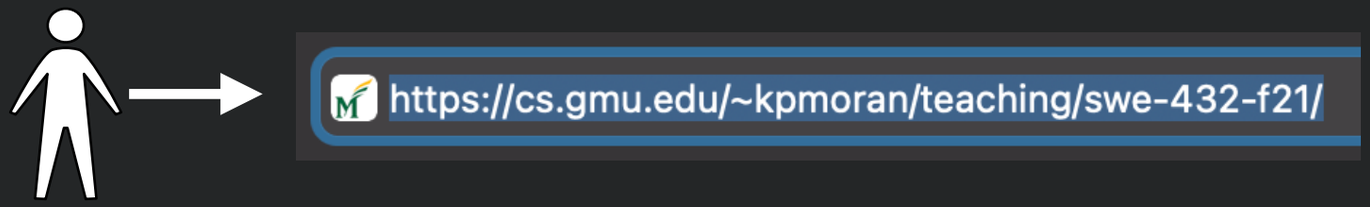
- Performance
- Scalability
- Simplicity of a Uniform Interface
- Modifiability of components (even at runtime)
- Visibility of communication between components by service agents
- Portability of components by moving program code with data
- Reliability

Principles of REST

- Client server: separation of concerns (reuse)
- Stateless: each client request contains all information necessary to service request (scaling)
- Cacheable: clients and intermediaries may cache responses. (scaling)
- Layered system: client cannot determine if it is connected to end server or intermediary along the way. (scaling)
- Uniform interface for resources: a single uniform interface (URIs) simplifies and decouples architecture (change & reuse)

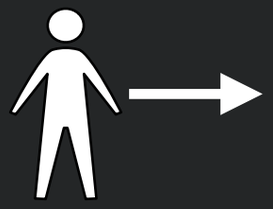
HTTP: HyperText Transfer Protocol

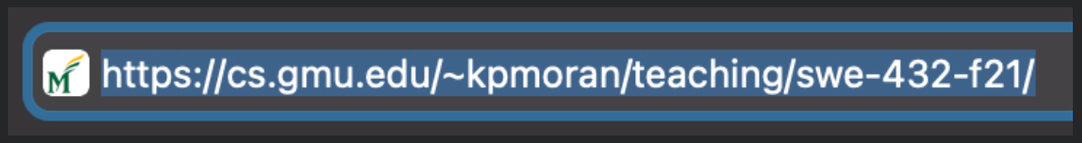
High-level protocol built on TCP/IP that defines how data is transferred on the web



HTTP: HyperText Transfer Protocol

High-level protocol built on TCP/IP that defines how data is transferred on the web



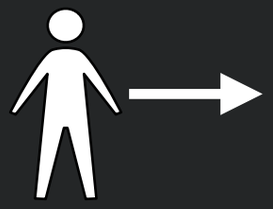


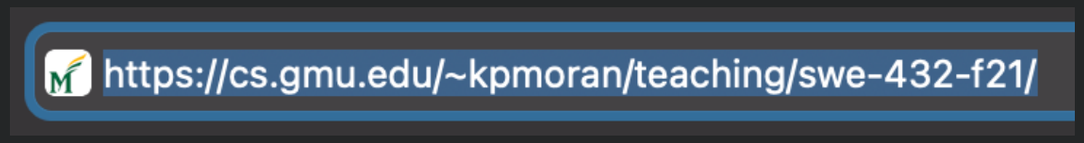
HTTP Request

```
GET /~kpmoran/swe-432-f21.html HTTP/1.1
Host: cs.gmu.edu
Accept: text/html
```

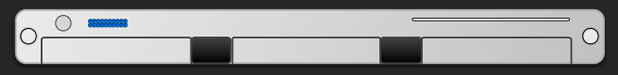
HTTP: HyperText Transfer Protocol

High-level protocol built on TCP/IP that defines how data is transferred on the web





web server

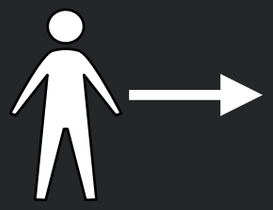


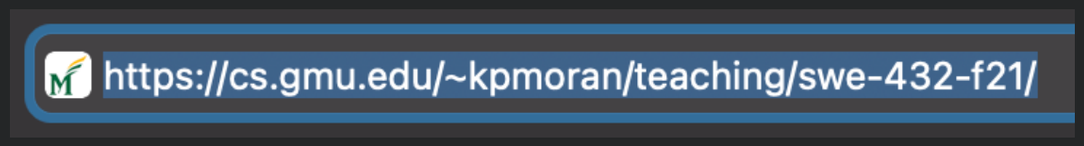
HTTP Request

```
GET /~kpmoran/swe-432-f21.html HTTP/1.1
Host: cs.gmu.edu
Accept: text/html
```

HTTP: HyperText Transfer Protocol

High-level protocol built on TCP/IP that defines how data is transferred on the web

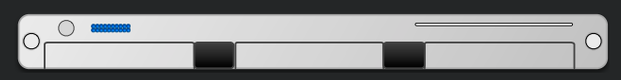




HTTP Request

```
GET /~kpmoran/swe-432-f21.html HTTP/1.1
Host: cs.gmu.edu
Accept: text/html
```

web server



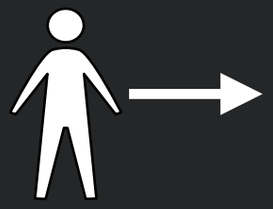
Reads file from disk

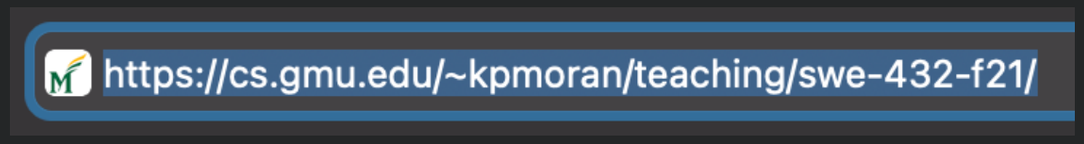
HTTP Response

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

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

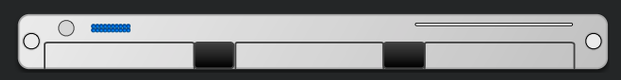
HTTP: HyperText Transfer Protocol

High-level protocol built on TCP/IP that defines how data is transferred on the web





web server



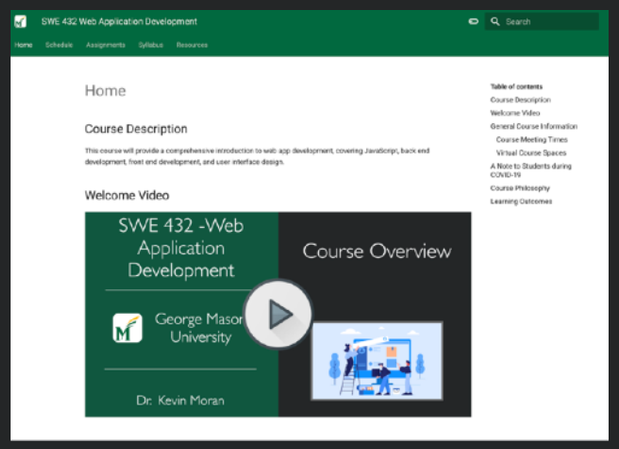
HTTP Request

```
GET /~kpmoran/swe-432-f21.html HTTP/1.1
Host: cs.gmu.edu
Accept: text/html
```

Reads file from disk

HTTP Response

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

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



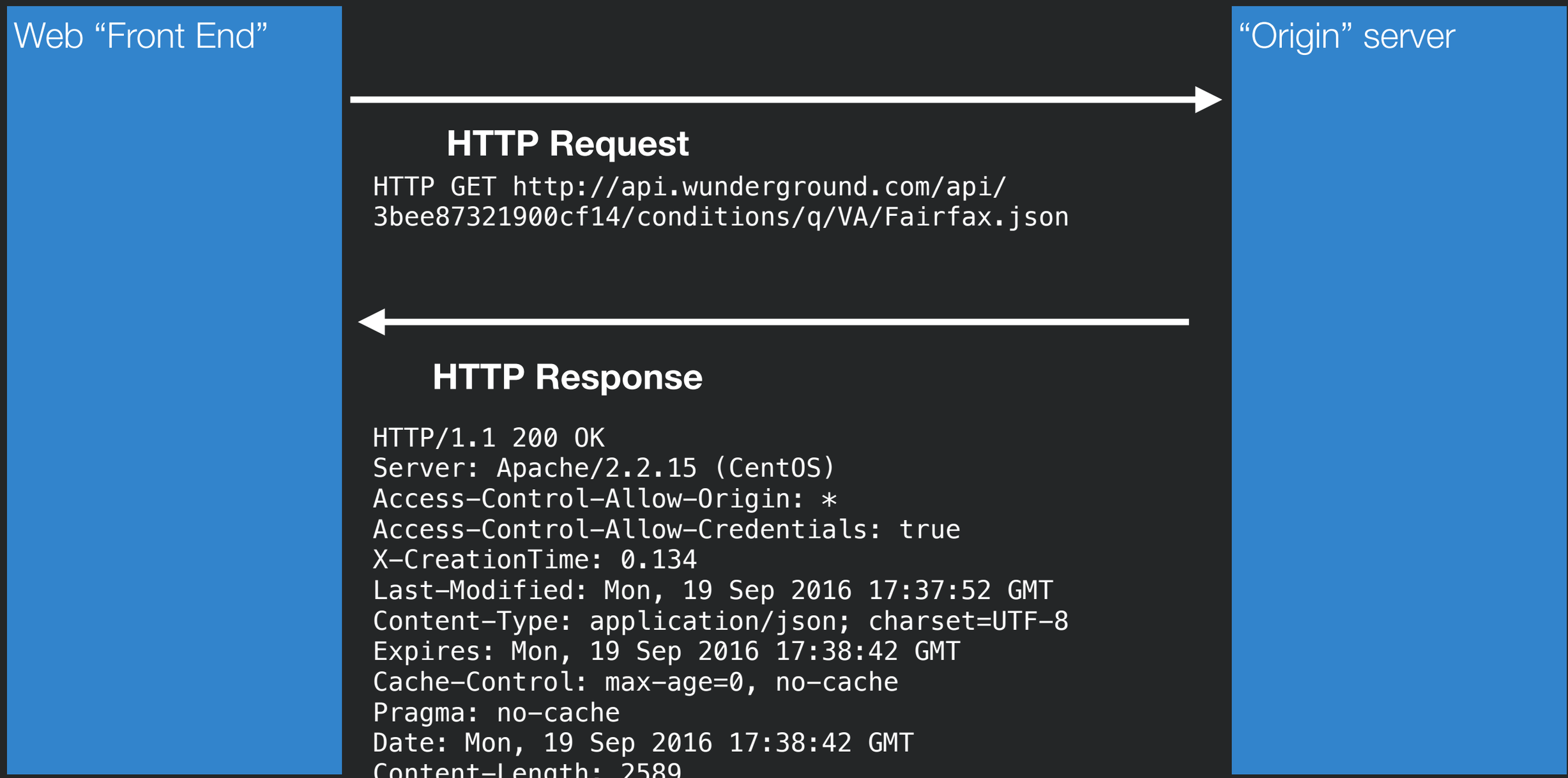
Uniform Interface for Resources

- Originally files on a web server
  - URL refers to directory path and file of a resource
- But… URIs might be used as an identity for any entity
  - A person, location, place, item, tweet, email, detail view, like
  - *Does not matter* if resource is a file, an entry in a database, retrieved from another server, or computed by the server on demand
  - Resources offer an *interface* to the server describing the resources with which clients can interact

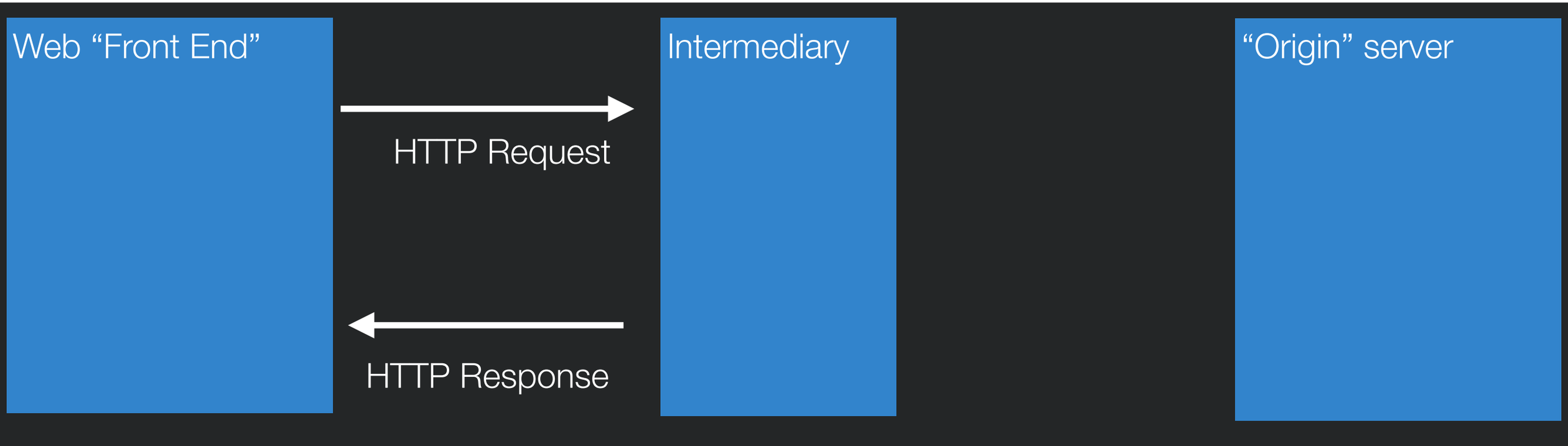
URI: Universal Resource Identifier

- Uniquely describes a resource
  - https://mail.google.com/mail/u/0/#inbox/157d5fb795159ac0
  - https://www.amazon.com/gp/yourstore/home/ref=nav_cs_ys
  - http://gotocon.com/dl/goto-amsterdam-2014/slides/StefanTilkov_RESTIDontThinkItMeansWhatYouThinkItDoes.pdf
  - Which is a file, external web service request, or stored in a database?
    - It does not matter
- As client, only matters what actions we can *do* with resource, not how resource is represented on server

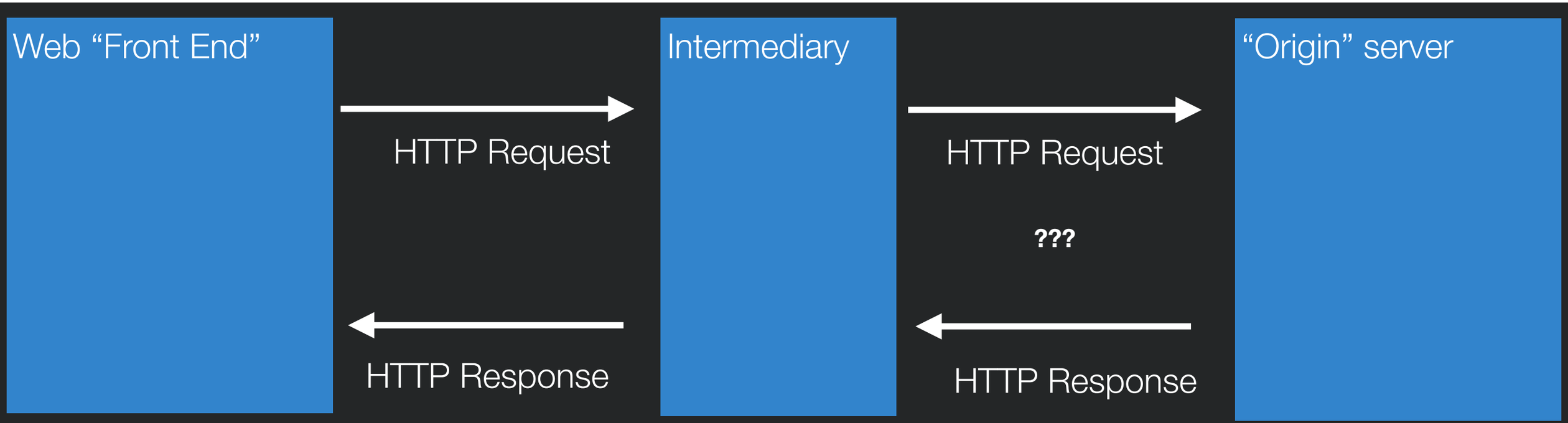
Intermediaries



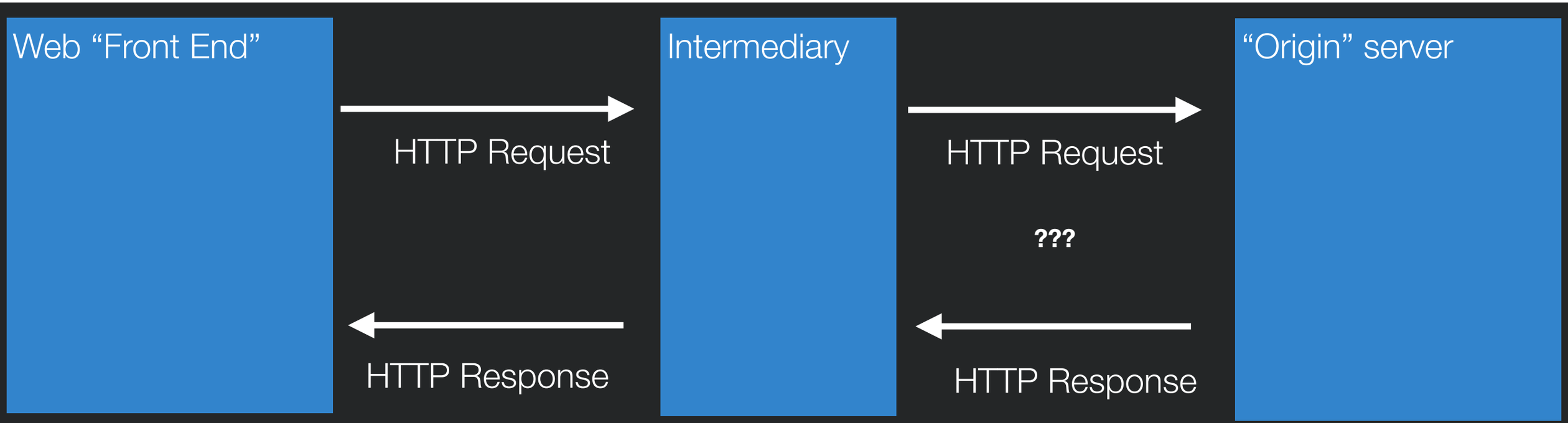
Intermediaries



Intermediaries



Intermediaries



- Client interacts with a resource identified by a URI
- But it never knows (or cares) whether it interacts with origin server or an unknown intermediary server
  - Might be randomly load balanced to one of many servers
  - Might be cache, so that large file can be stored locally
    - (e.g., GMU caching an OSX update)
  - Might be server checking security and rejecting requests

Challenges with intermediaries

- But can all requests really be intercepted in the same way?
  - Some requests might produce a change to a resource
    - Can’t just cache a response… would not get updated!
  - Some requests might create a change every time they execute
    - Must be careful retrying failed requests or could create extra copies of resources

HTTP Actions

- How do intermediaries know what they can and cannot do with a request?
- Solution: HTTP Actions
  - Describes what will be done with resource
  - GET: retrieve the current state of the resource
  - PUT: modify the state of a resource
  - DELETE: clear a resource
  - POST: initialize the state of a new resource

HTTP Actions

- GET: safe method with no side effects
  - Requests can be intercepted and replaced with cache response
- PUT, DELETE: idempotent method that can be repeated with same result
  - Requests that fail can be retried indefinitely till they succeed
- POST: creates new element
  - Retrying a failed request might create duplicate copies of new resource