Code Fellows

Class 07
Express

seattle-javascript-401n14

Lab 06 Review

Code Challenge 06 Review

Vocab Review!

What is a server?

What is a server?

A server is an application whose purpose is to provide data or expose APIs to another application, called the **client**. A single server can have multiple clients, and a single client can use multiple servers. We created a MongoDB server in Lab 05.

What is HTTP?

What is HTTP?

HTTP stands for **HyperText Transfer Protocol**, and it is how data is shared across the internet. HTTP is not just the beginning of most web URLs, it is also a format for making data requests. For example, every website is just a path from where our browser then loads data onto our screens.

What is REST?

What is REST?

Representational state transfer (REST) is a set of constraints to be used for creating **web servers**. REST also defines how clients can communicate with those **RESTful** servers.

What is an HTTP status code?

What is an HTTP status code?

When we make requests over HTTP, we need a standardized way to figure out what the status of our request is. HTTP status codes range from 1xx-5xx, and each status code has a set meaning which our code can interpret.

What is node-fetch?

What is node-fetch?

The package **node-fetch** let’s us easily write requests in our client-side code. We can send all the required REST fields, and, because **node-fetch** is promise based, we can asynchronously handle our responses.

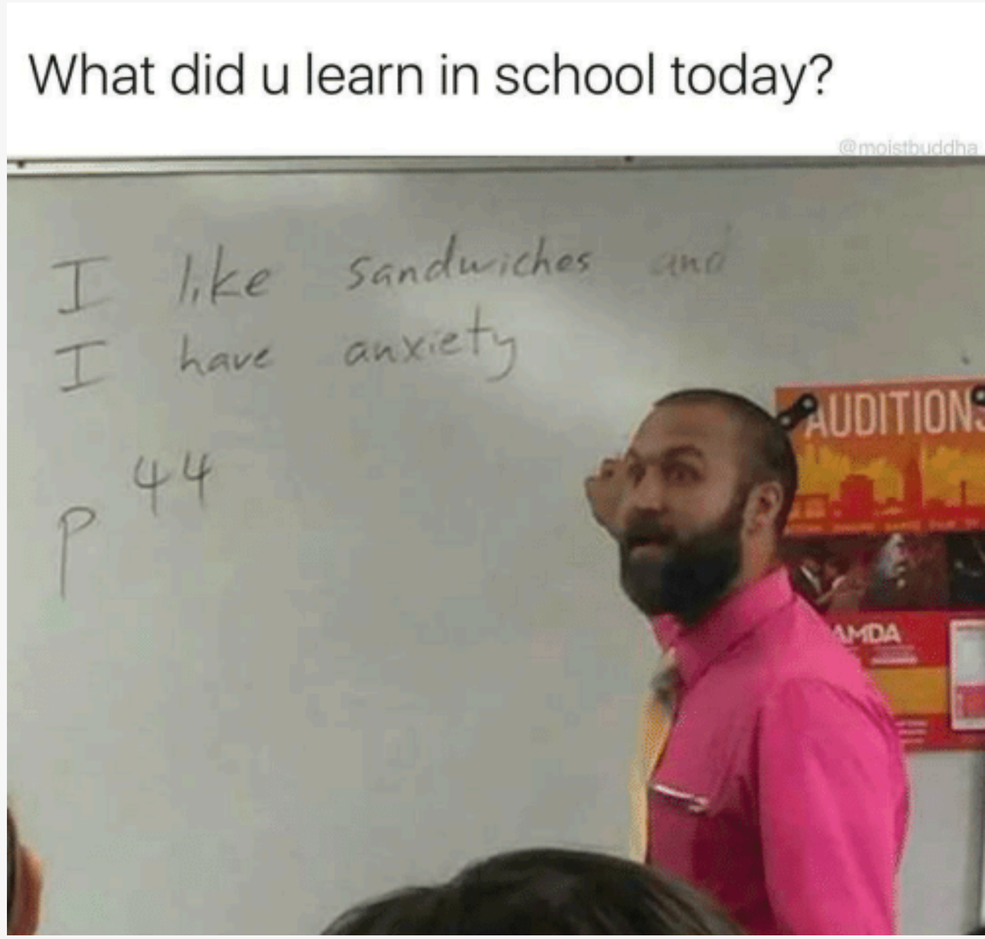
What is the web request response cycle?

What is the web request response cycle?

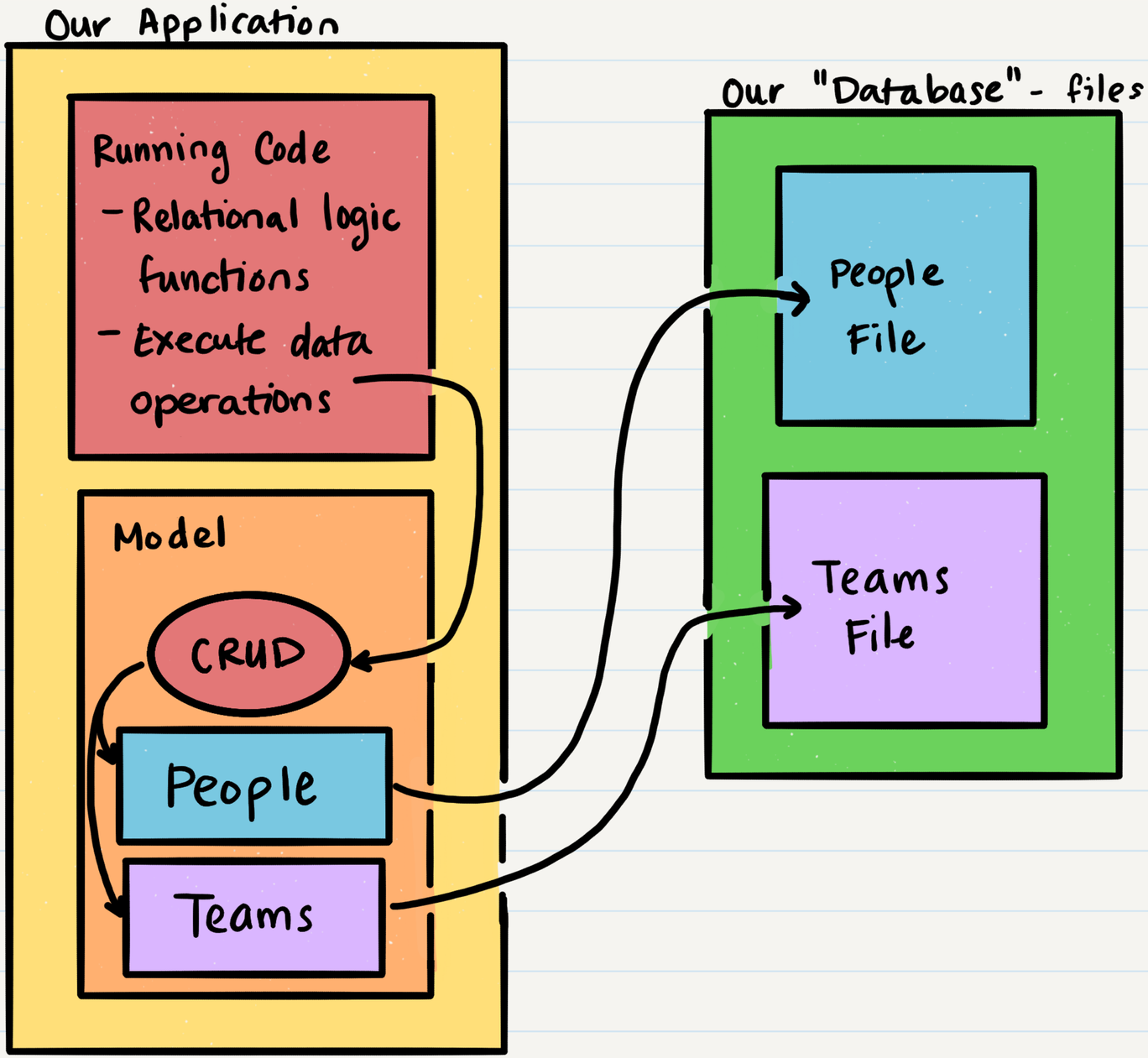
The web request response cycle (WRRC) is simply a way of understanding how the internet works. Everything you do on the internet is a series of requests and responses; the browser sends GET requests to URLs in order to load HTML content, and web applications use additional HTTP requests to get and manage data.

What Have We Done?

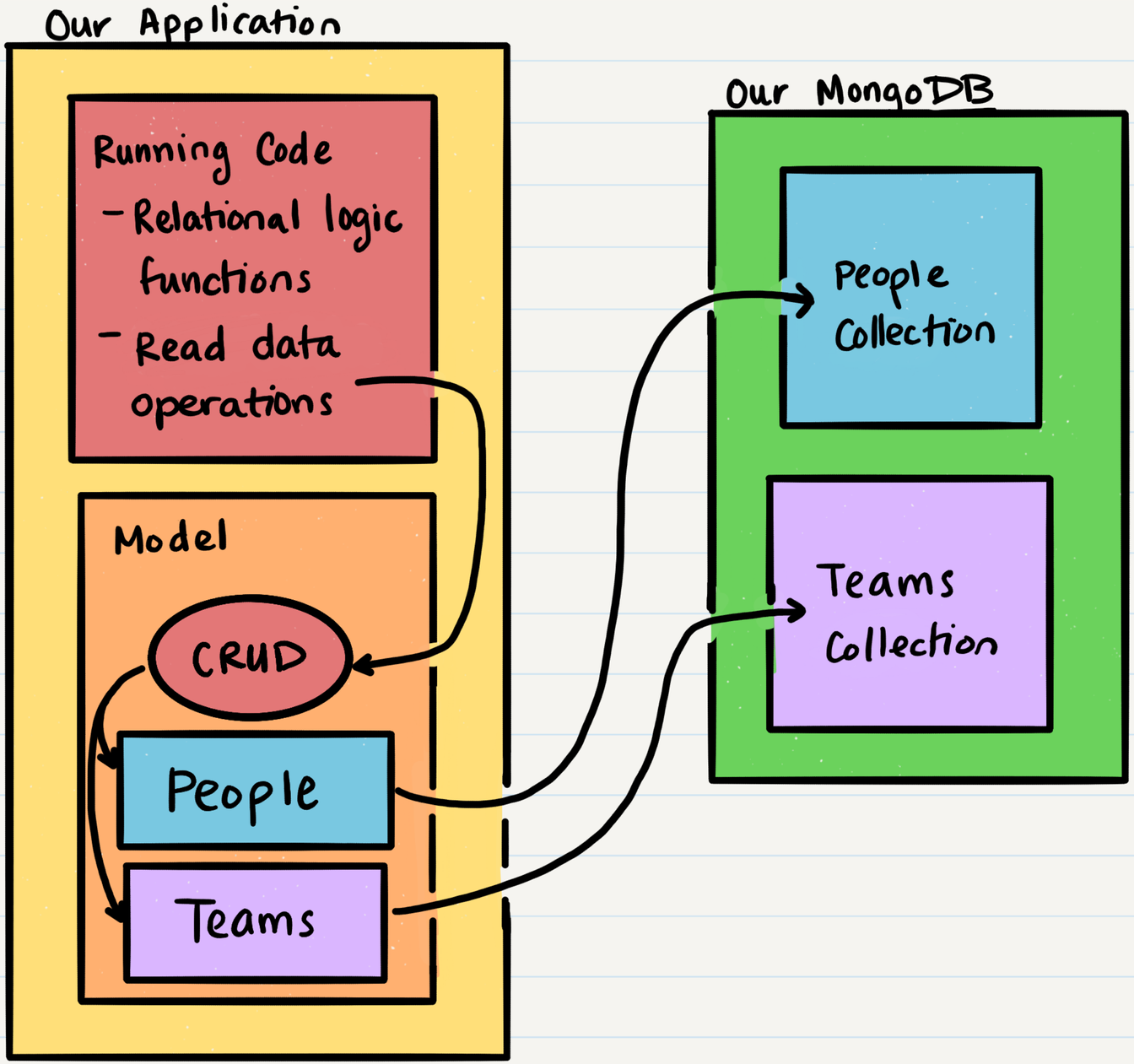
- Our past three classes have been building on each other
- Class 04 - How to model data, what CRUD is, async calls to save data, relational data
- Class 05 - NoSQL, MongoDB, validation via the middleman Mongoose, database connection
- Class 06 - HTTP requests, web servers, REST, requests and responses



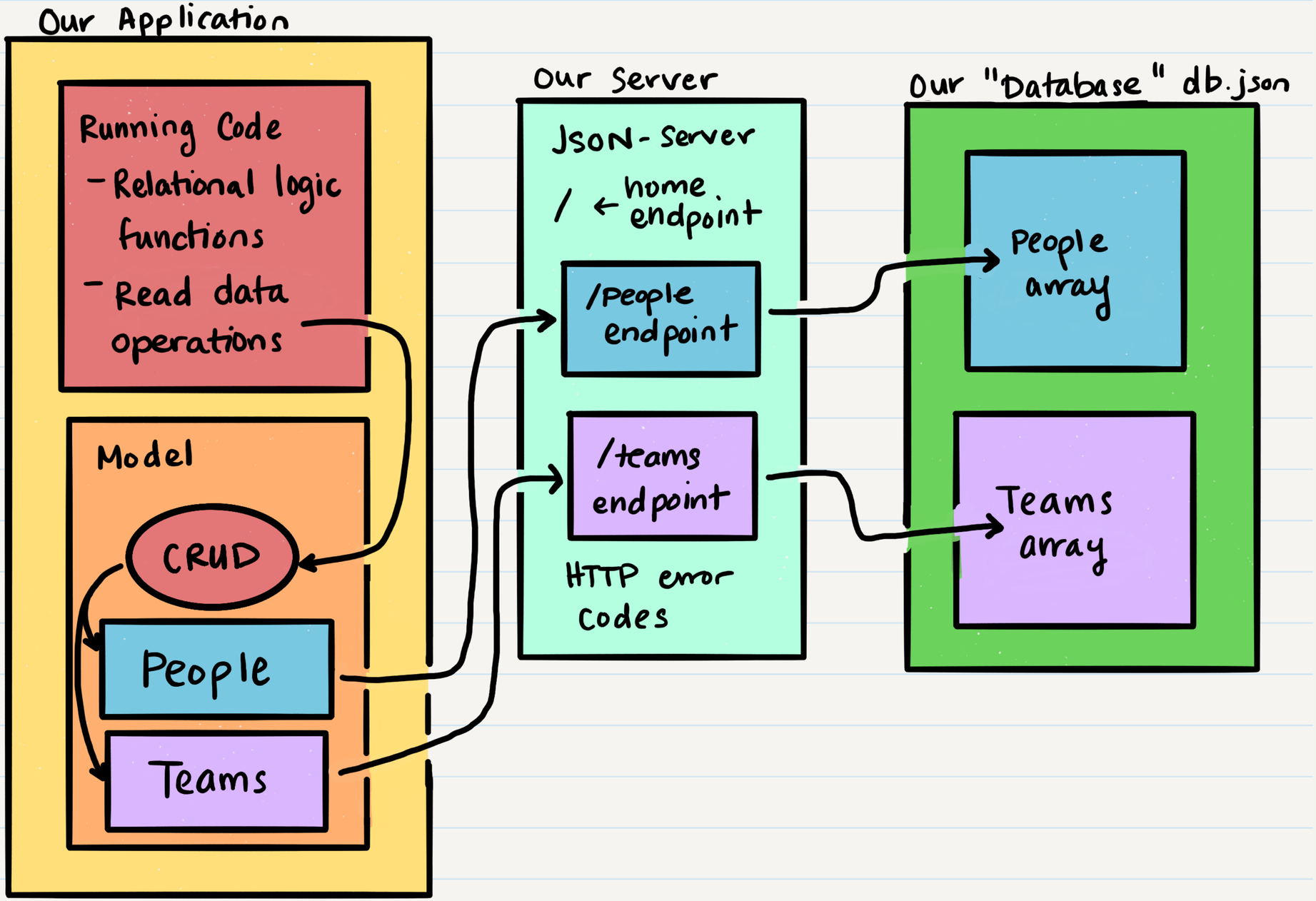
CLASS 04


Our Application
Running Code
- Relational logic functions
- Execute data operations
Model
CRUD
People
Teams
Our "Database" - files
People File
Teams File

CLASS 05

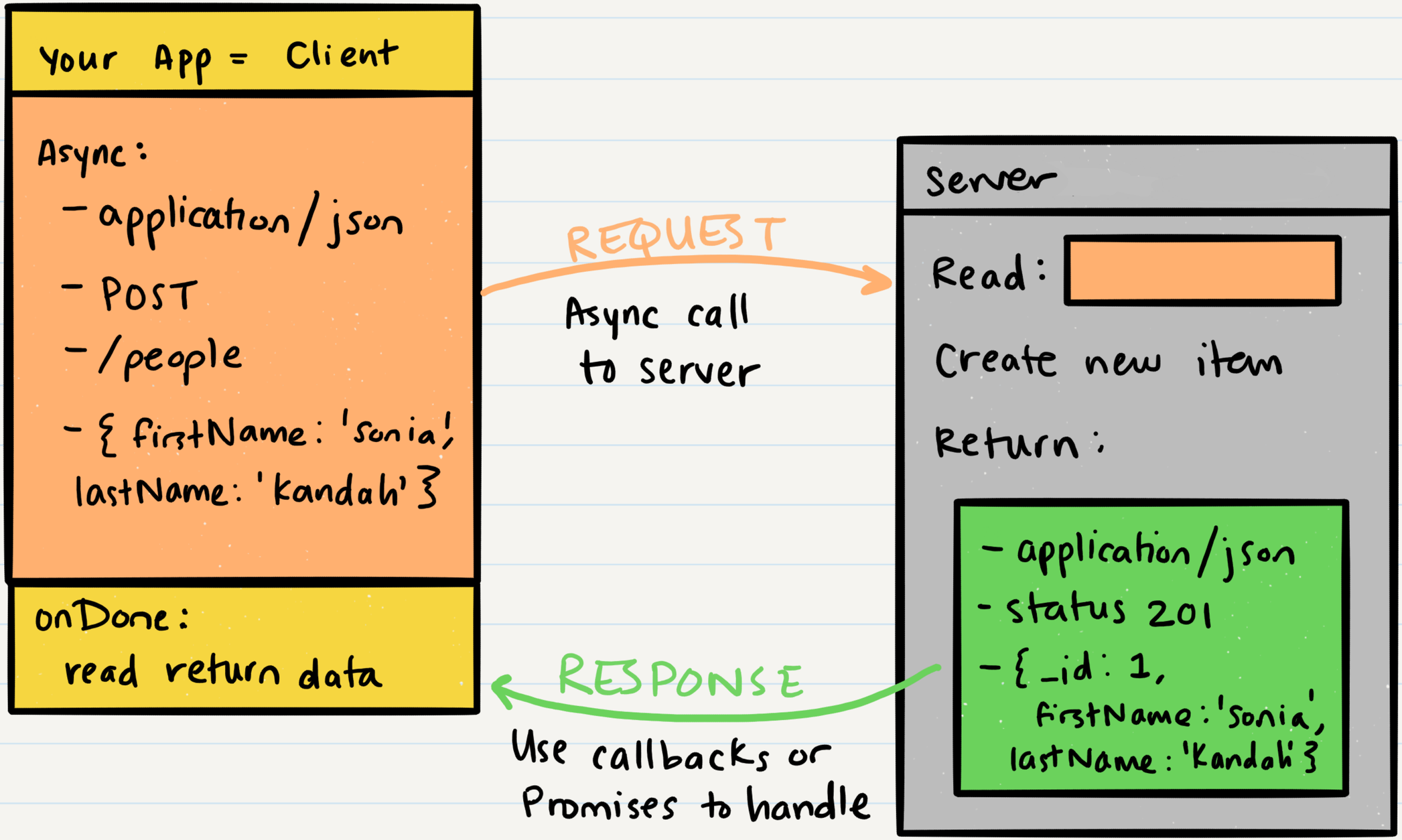

Our Application
Running Code
- Relational logic functions
- Read data operations
Model
CRUD
People
Teams
Our MongoDB
People Collection
Teams Collection

CLASS 06


Our Application
Running Code
- Relational logic functions
- Read data operations
Model
CRUD
People
Teams
Our Server
JSON-Server
/ ← home endpoint
/people endpoint
/teams endpoint
HTTP error codes
Our "Database" db.json
People array
Teams array

The Web Request Response Cycle

- The internet is just a series of **requests and responses** between clients and servers
- We want to request **resources** from servers: these resources can be HTML pages (if we request `facebook.com`) or they can be data (if we request `localhost:3000/people/1`)
- When the client receives the resource in a response, it can handle the response however it wants
- Your web browser (Chrome, Safari, Edge) is a client, with the servers being websites you navigate to
    - Browsers use `GET` requests to get a URL, and then attempt to display responses as an HTML page


Your App = Client
Async:
- application/json
- POST
- /people
- { firstName: 'sonia',
lastName: 'kandah' }
onDone:
read return data
REQUEST
Async call
to server
RESPONSE
Use callbacks or
Promises to handle
Server
Read:
Create new item
Return:
- application/json
- status 201
- { _id: 1,
firstName: 'sonia',
lastName: 'Kandah' }

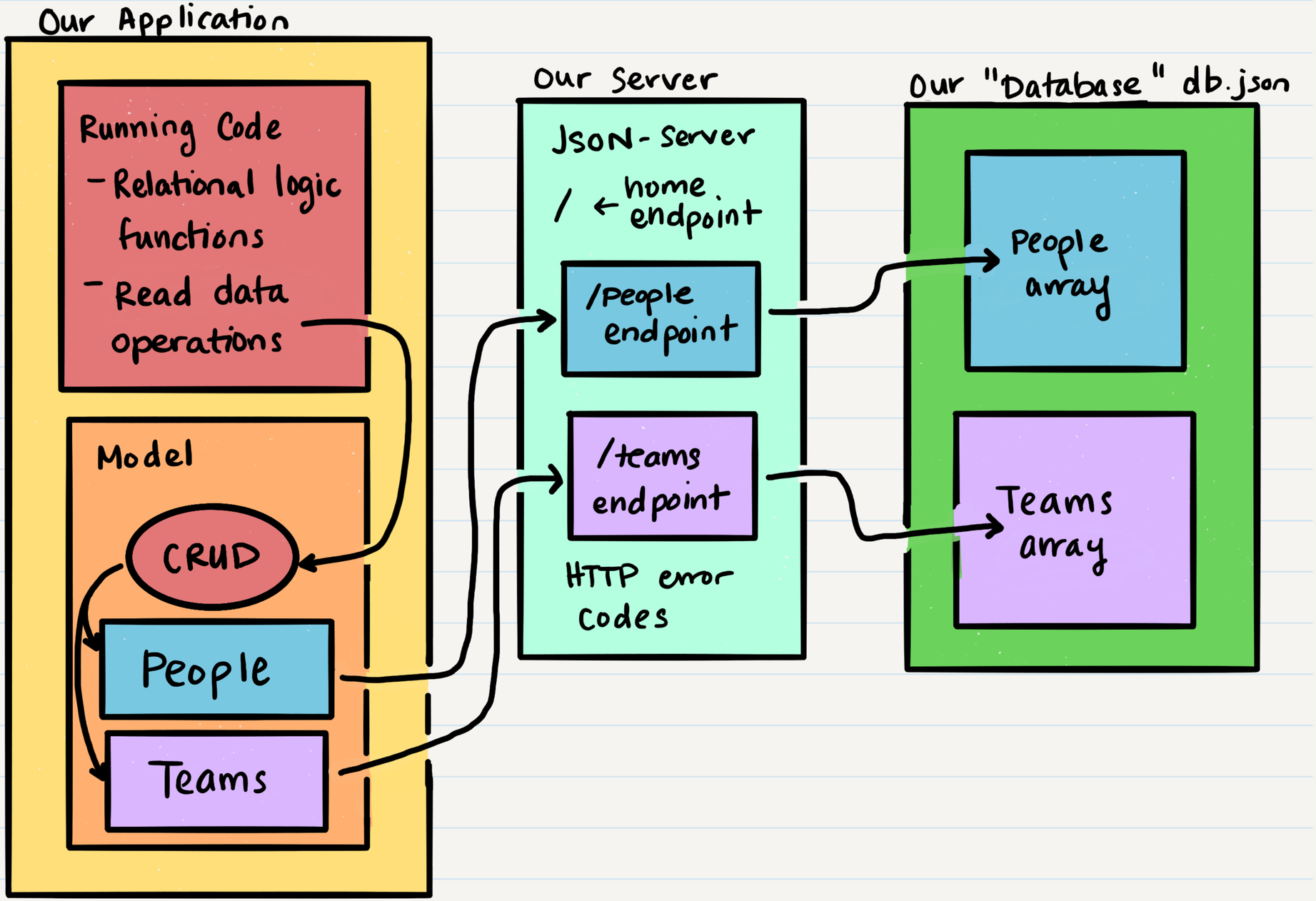
Web Applications

- When we think of web apps, we usually think of something more complex than a website with a few pages
- Facebook doesn’t have a collection of hardcoded HTML pages: the content of most Facebook pages is usually fully unique to each logged-in user
- How can we transform our code-complex applications into websites without HTML pages?

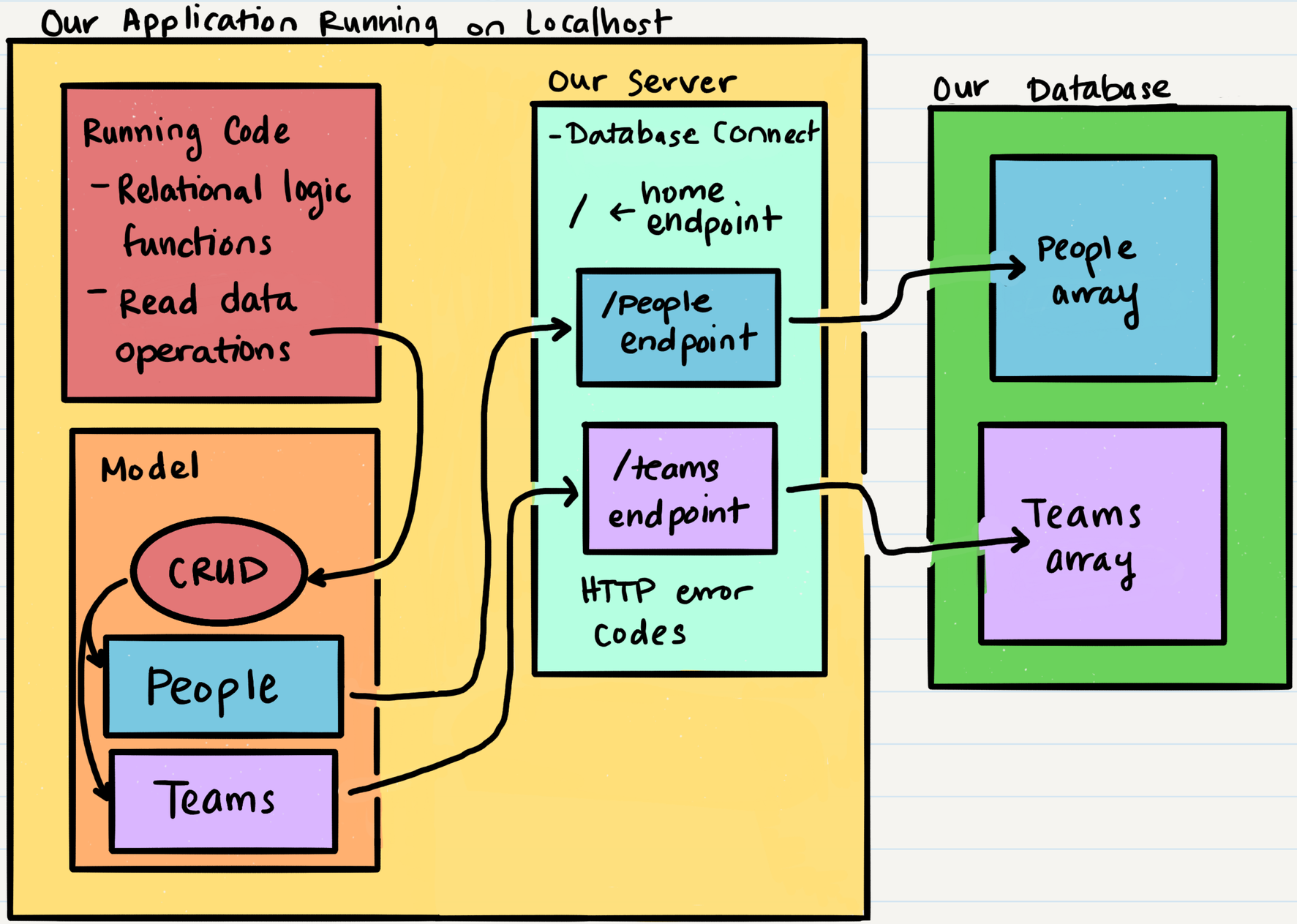
Remember JSON-Server?

- We ran it from the command line
- It created some HTML pages for us
- It hosted those HTML pages on our localhost
- It exposed URL paths (`/people` or `/teams`) that we could navigate to in our browser
- What if we had more control on this functionality in our application?

CLASS 06


Our Application
Running Code
- Relational logic functions
- Read data operations
Model
CRUD
People
Teams
Our Server
JSON-Server
/ ← home endpoint
/people endpoint
/teams endpoint
HTTP error codes
Our "Database" db.json
People array
Teams array

OUR GOAL


Our Application Running on Localhost
Running Code
- Relational logic functions
- Read data operations
Model
CRUD
People
Teams
Our Server
-Database Connect
/ ← home endpoint
/people endpoint
/teams endpoint
HTTP error codes
Our Database
People array
Teams array

Making Our Own Server

- If we can create a server within our own application, we'll have a lot more control over how our browser reads our application
- We can also break our application into front-end and back-end; **client-side** and **server-side**
- Using the package **express**, we can build our server a little easier

```
const start = port => {
  let PORT = port || process.env.PORT || 3000;
  app.listen(PORT, () => {
    console.log(`Listening on ${PORT}`);
  });
};

app.get('/', (req, res) => {
  res.send(
    "Congrats! You're running a web application with a client and a server!"
  );
});

app.get('/cats', (req, res) => {
  res.send('Did you know that I love cats?');
});

app.get('/dogs', (req, res) => {
  res.send('I happen to also love dogs!');
});

module.exports = {
  server: app,
  start: start
};
```

```
soniakandah … class-07 demo api-server master + 1 nodemon --exec node index.js
[nodemon] 1.19.4
[nodemon] to restart at any time, enter `rs`
[nodemon] watching dir(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
Listening on 3000
```

Demo

demo/api-server

The node package express can make it super simple to get basic server functionality quickly. This can immediately transform our applications from running on our terminal into being hosted on our localhost.

Routes and Middleware

- In our server, there are two fundamental things we want to accomplish:
  - Create endpoints or **routes** that either provide data or host content
  - Write some custom code that runs between requests our client-side makes and the response our server-side returns: we call this **middleware**
  - Middleware is software that acts as a bridge between a client and server

```
// It will run on all routes (and you'll see it's logs in the console)
app.use(logger);

let routeMW = str => {
  return (req, res, next) => {
    req.str = str;
    next();
  };
};

app.get('/cats', routeMW('Kitto'), (req, res, next) => {
  res.status(200);
  res.send(`My cat is named ${req.str}`);
});

// ==== ROUTE MIDDLEWARE ================================================

router.get('/', (req, res, next) => {
  res.status(200);
  res.send("It looks like I'm at /, but really I'm at /dogs");
});
```

```
soniakandah … class-07 demo middleware master + 2 nodemon --exec node index-server.js
[nodemon] 1.19.4
[nodemon] to restart at any time, enter `rs`
[nodemon] watching dir(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index-server.js`
Logging something
Logging something
Logging something
Logging something
Unknown Route
```

Demo

demo/ middleware

Middleware let's us do some action between the client's request and response, which can be very powerful.

There are many packages that create middleware for your application.

Types of Middleware

- Application Middleware
    - Usually effects either the whole application, or an entire endpoint
    - We run this middleware either with `app.use()` or `app.METHOD()`, where `METHOD` is one of: get, put, post, delete, etc.
- Route Middleware
    - Usually changes how we define routes
    - We run this middleware using `express.Router()`

Types of Middleware

- Error Handling
    - Takes a fourth argument at the beginning, `err`
    - Automatically called when we throw an error or when we pass a parameter into `next()`

```
let db = require('./db.js');

app.use(express.json());

// Default Route
app.get('/', (req, res, next) => {
  res.send('Homepage');
});

// Route to Get all People
app.get('/people', (req, res, next) => {
  let count = db.people.length;
  let results = db.people;
  res.json({ count, results });
});

// Route to Get a person
app.get('/people/:id', (req, res, next) => {
  let id = req.params.id;
  let record = db.people.filter(record => record.id === parseInt(id));
  res.json(record[0]);
});

// Route to Create a person
app.post('/people', (req, res, next) => {
  let record = req.body;
  record.id = Math.random();
  db.people.push(record);
  res.json(record);
});
```

Lab

lab/starter-code

Let's walk through getting set up with our starter code for Lab 07.

We'll create a simple express server, use a changeable JavaScript object as a database, and establish a few routes.

What's Next:

- After Lunch: **Lightning Talk - Liskov Substitution Principle**
- Due by Midnight: **Learning Journal 07**
- Due by Midnight Sunday: **Feedback Survey Week 4** and **2 Career Coaching assignments**
- Due by Midnight Monday: **Code Challenge 07**
- Due by 6:30pm Tuesday:
    - **Lab 07**
    - **Read: Class 08**
- Next Class:
    - **Class 08 - Express Routing and Connected API**

Questions?