CSE 331
Software Design & Implementation

Autumn 2022

HW9, JSON, and Fetch

Administrivia

- HW8 due yesterday (Wed. 11/30 @ 11:00pm)
  - No re-runs (no staff tests). It's your responsibility to check that your submission runs without any compilation errors!
    - Double-check you tagged the correct commit by heading over to GitLab, and locating Repository > Graph on the left sidebar!

- HW9 due next **Friday** (12/9 @ 11:00pm)
  - Extra credit available!
    - Get creative! Lots of cool opportunities.
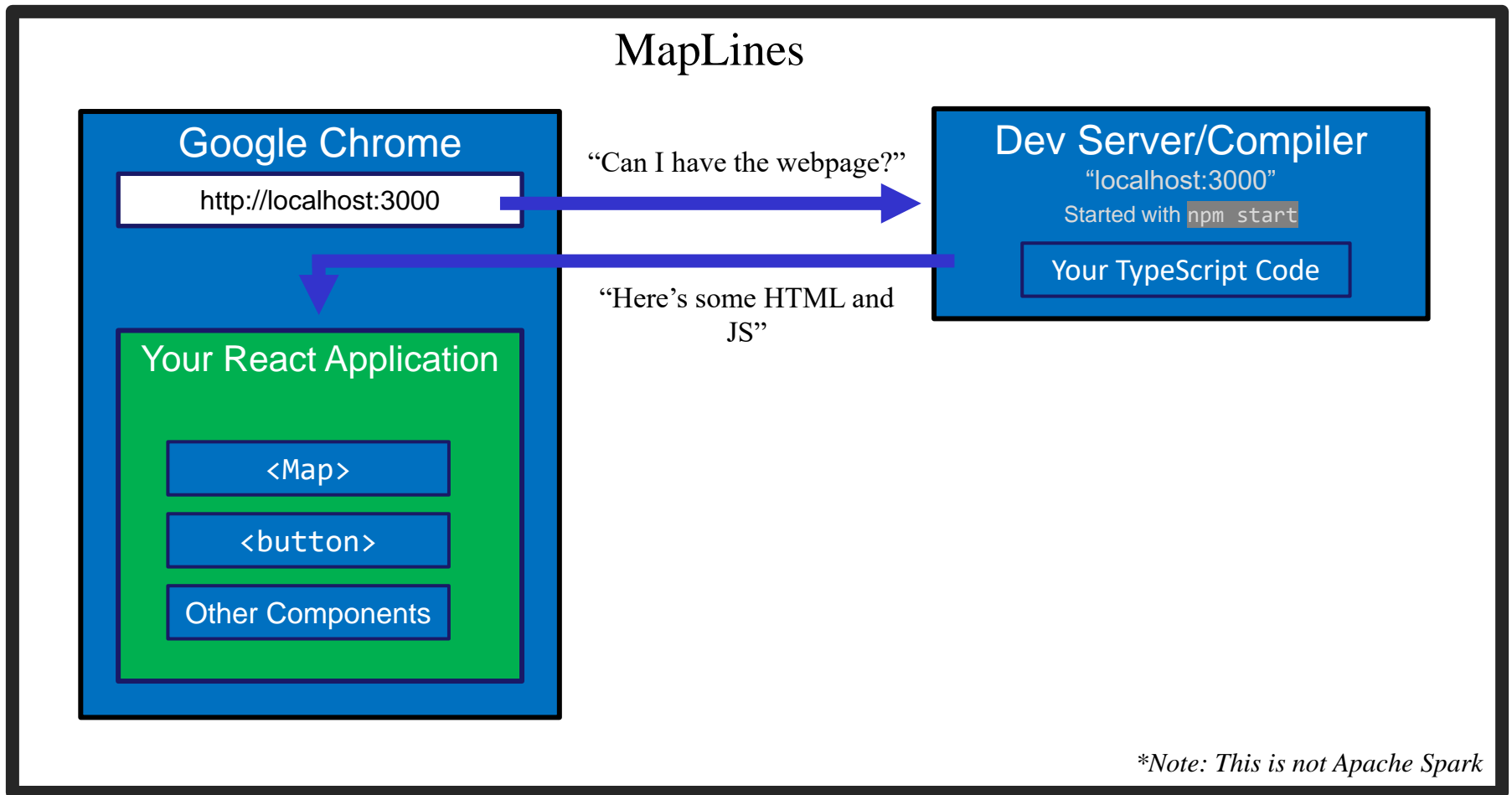 - No GitLab pipeline, **tag** needed still! No re-runs again.

- Any questions?

Agenda

- HW9 Overview

- JSON
  - Brief overview
  - Helps share data between Java and JS.

- Fetch
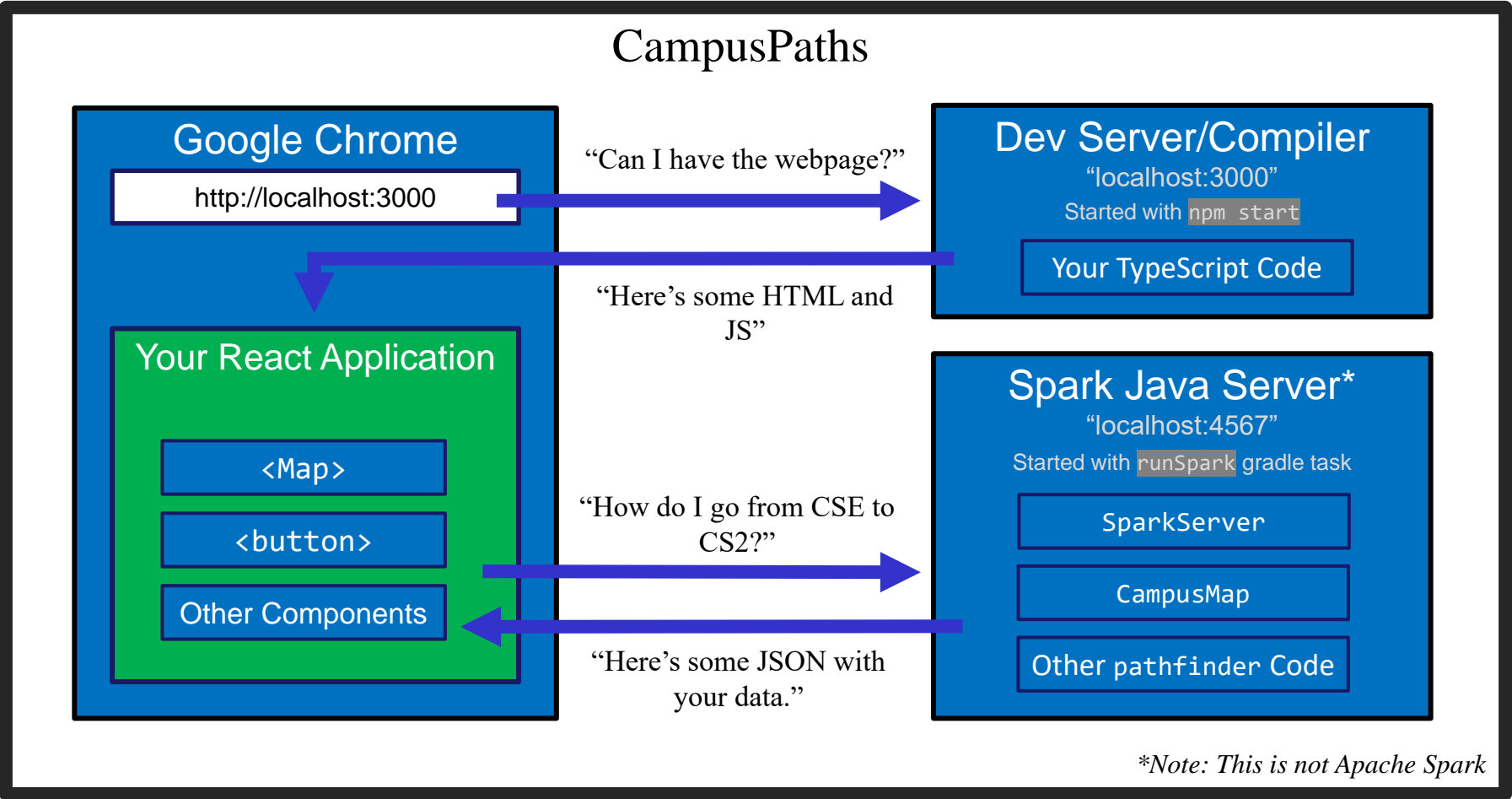 - How your JS sends requests to the Java server.

Homework 9 Overview

- Creating a new web GUI using React
  - Display a map and draw paths between two points on the map.
  - Similar to your React app in HW8 – but you may add more!
  - Send requests to your **Java server** (new) to request building and path info.

- Creating a **Java server** as part of your previous HW5-7 code
  - Receives **requests** from the React app to calculate paths/send data.
  - Not much code to write here thanks to **MVC**.
    - Reuse your `CampusMap` class from HW7.

The Map Lines Stack

MapLines

Google Chrome
http://localhost:3000

"Can I have the webpage?"

"Here's some HTML and JS"

Dev Server/Compiler
"localhost:3000"
Started with `npm start`

Your TypeScript Code

Your React Application

`<Map>`

`<button>`

Other Components

Note: This is not Apache Spark

The Campus Paths Stack

CampusPaths

Google Chrome
http://localhost:3000

"Can I have the webpage?"

"Here's some HTML and JS"

Your React Application
- `<Map>`
- `<button>`
- Other Components

"How do I go from CSE to CS2?"

"Here's some JSON with your data."

Dev Server/Compiler
"localhost:3000"
Started with `npm start`

Your TypeScript Code

Spark Java Server*
"localhost:4567"
Started with `runSpark` gradle task

SparkServer

CampusMap

Other `pathfinder` Code

Note: This is not Apache Spark

Any Questions?

- Done:
  - HW9 Basic Overview

- Up Next:
  - JSON
  - Fetch

JSON

- We have a whole application written in Java so far:
  - Reads CSV data, manages a Graph data structure with campus data, uses Dijkstra's algorithm to find paths.
- We're writing a whole application in JavaScript:
  - React web app to create an interactive GUI for your users

- Even if we get them to communicate (discussed later), we need to make sure they "speak the same language".
  - JavaScript and Java store data *very* differently.
- JSON = JavaScript Object Notation
  - Can convert JS Object → String, and String → JS Object
  - Bonus: Strings are easy to send inside server requests/responses.

JSON ↔ Java

Java Object

```java
public class SchoolInfo {

  String name = "U of Washington";
  String location = "Seattle";
  int founded = 1861;
  String mascot = "Dubs II";
  boolean isRainy = true;
  String website = "www.uw.edu";
  String[] colors = new String[]
                  {"Purple", "Gold"};

}
```

JSON String

`{"name":"U of Washington","location":"Seattle","founded":1861,"mascot":"Dubs II","isRainy":true,"website":"www.uw.edu","colors":["Purple","Gold"]}`

- Use Gson (a library from Google) to convert between them.
  - Tricky (but possible) to go from JSON String to Java Object, but we don't need that for this assignment.

```java
Gson gson = new Gson();
SchoolInfo sInfo = new SchoolInfo();
String json = gson.toJson(sInfo);
```

JSON ↔ JS

Javascript Object

```
let schoolInfo = {

  name: "U of Washington",
  location: "Seattle",
  founded: 1861,
  mascot: "Dubs II",
  isRainy: true,
  website: "www.uw.edu",
  colors: ["Purple","Gold"]

}
```

JSON String

```
{"name":"U of
Washington","location":"Seattle","foun
ded":1861,"mascot":"Dubs
II","isRainy":true,"website":"www.uw.e
du","colors":["Purple","Gold"]}
```

- Can convert between the two easily (we'll see how later)
- This means: if the server sent back a JSON String, it'd be easy to use the data inside of it – just turn it into a JS Object and read the fields out of the object.

JSON – Key Ideas

- Use Gson to turn Java objects containing the data into JSON before we send it back.

  – The Java objects don't have to be simple, like in the example, Gson can handle complicated structures.

- We can then turn the JSON string into a Javascript object so we can use the data (fetch can help us with that).

Any Questions?

- Done:
  - HW9 Basic Overview
  - JSON

- Up Next:
  - Fetch

What is a Request?

- Recall from lecture:
  - When you type a URL into your browser, it makes a GET request to that URL, the response to that request is the website itself (HTML, JS, etc..).
    - A GET request says "Hey server, can I get some info about _____?"
  - We're going to make a request from inside Javascript to ask for data about paths on campus.
  - There are other kinds of requests, but we're just using GET. (It's the default for `fetch`).
- Each "place" that a request can be sent is called an "endpoint."
  - Your Java server will provide multiple endpoints – one for each kind of request that your React app might want to make.
    - Find a path, get building info, etc...

Forming a Request
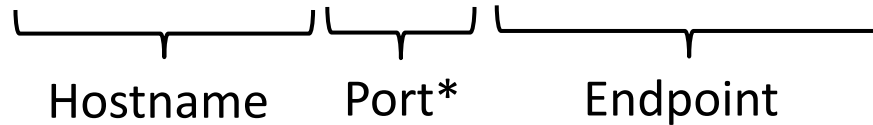
- Basic request with no extra data: `"http://localhost:4567/getSomeData"`
    - A request to the `"/getSomeData"` endpoint in the server at `"localhost:4567"`
    - `"localhost"` just means "on this same computer"
    - `":4567"` specifies a port number – every computer has multiple ports so multiple things can be running at a given time.
- Sending extra information in a request is done with a query string:
    - Add a `"?"`, then a list of `"key=value"` pairs. Each pair is separated by `"&"`.
    - Query string might look like: `"?start=CSE&end=KNE"`
- Complete request looks like:

    `http://localhost:4567/findPath?start=CSE&end=KNE`

- Sends a "/findPath" request to the server at "localhost:4567", and includes two pieces of extra information, named "start" and "end".
- You don't need to name your endpoints or query string parameters anything specific, the above is just an example.

Forming a Request
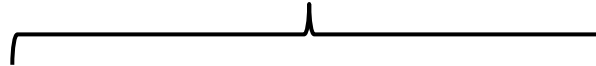
`http://washington.edu/about`

`http://localhost:4567/getSomeData`

Hostname   Port*   Endpoint

Query Params*

`http://localhost:4567/findPath?start=CSE&end=KNE`

*Port and query params are technically optional

Servicing Requests

- Recall from lecture:
  - We need some way to respond to these requests
  - This is what we use our **SparkServer** for!
  - For each "endpoint" we want, we need to define a route:

```
Spark.get("/hello-world", new Route() {
    @Override
    public Object handle(Request request, Response response)
            throws Exception {
        // we need to return our response
        return "Hello, Spark!";
    }
});
```

Requests and Spark Server Demo

Running the Section Demo

- Like last time, download and unzip the files from the website.
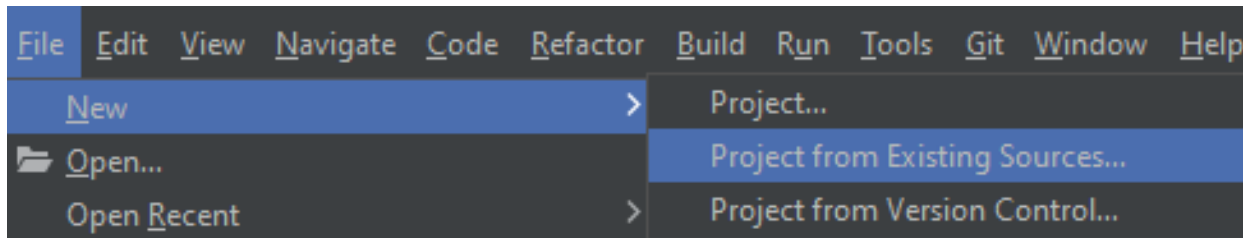
- New > Project from Existing Sources…
  - Choose the **build.gradle** file inside of the **sec09-demo** directory.
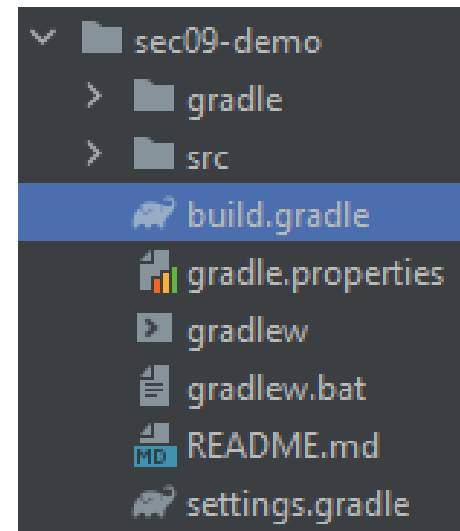
Running the Section Demo

- Get the installation out of the way since it takes a while (have this install in the background while you check out the Spark demo!)

- In the IntelliJ terminal:
  - **`cd src/main/react`**
  - **`npm install`**

- Success! (Again, these warnings are **expected** and **normal**.)

```
added 1914 packages from 751 contributors and audited 1920 packages in 284.332s

127 packages are looking for funding
  run `npm fund` for details

found 128 vulnerabilities (2 low, 65 moderate, 46 high, 15 critical)
  run `npm audit fix` to fix them, or `npm audit` for details
```
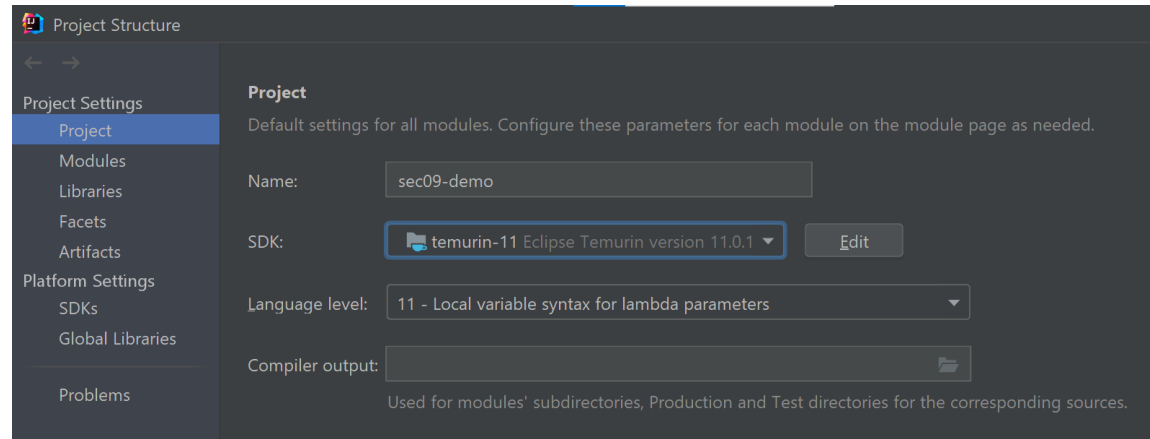
Starting up the Spark Server

- Start up the Spark Server by running the **runSpark** Gradle task**.**

- Alternatively, run the **main** method of
  src/main/java/sparkDemo/SparkServer.java
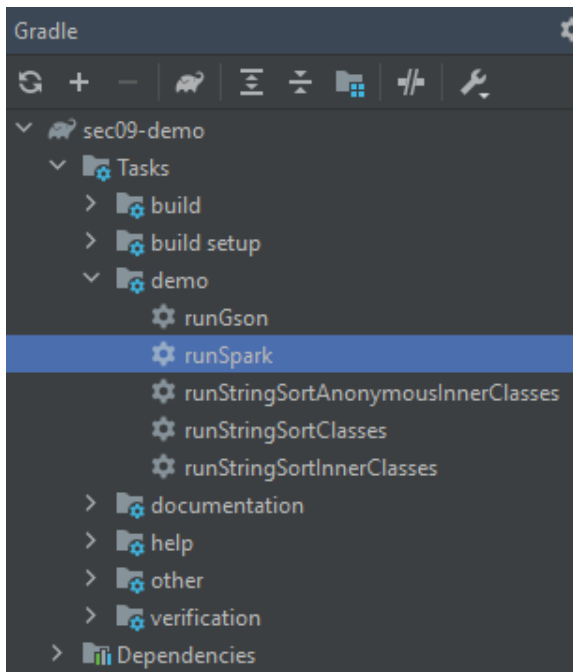
Compile error? Make sure you're using Java 11!
File > Project Structure > Project
Check that the SDK is correct!

Starting up the Spark Server

- Your server is now running on **http://localhost:4567**

```
[main] INFO Spark Demo Server - Listening on: http://localhost:4567
[Thread-0] INFO org.eclipse.jetty.util.log - Logging initialized @299ms to org.eclipse.jetty.util.log.Slf4jLog
[Thread-0] WARN org.eclipse.jetty.server.AbstractConnector - Ignoring deprecated socket close linger time
[Thread-0] INFO spark.embeddedserver.jetty.EmbeddedJettyServer - == Spark has ignited ...
[Thread-0] INFO spark.embeddedserver.jetty.EmbeddedJettyServer - >> Listening on 0.0.0.0:4567
[Thread-0] INFO org.eclipse.jetty.server.Server - jetty-9.4.12.v20180830; built: 2018-08-30T13:59:14.071Z; git: 27208684755d94a9218
[Thread-0] INFO org.eclipse.jetty.server.session - DefaultSessionIdManager workerName=node0
[Thread-0] INFO org.eclipse.jetty.server.session - No SessionScavenger set, using defaults
[Thread-0] INFO org.eclipse.jetty.server.session - node0 Scavenging every 600000ms
[Thread-0] INFO org.eclipse.jetty.server.AbstractConnector - Started ServerConnector@30124862{HTTP/1.1,[http/1.1]}{0.0.0.0:4567}
[Thread-0] INFO org.eclipse.jetty.server.Server - Started @896ms
```

- These are **not** errors – the server just outputs info in red text.

- Let's try sending a request to the server…
  – Visit **http://localhost:4567** in a browser

Starting up the Spark Server

- We got a 404 Not Found Page. Why is this?
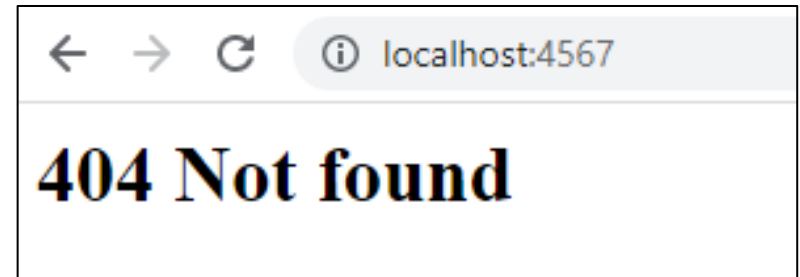
- INFO spark.http.matching.MatcherFilter - The requested route [/] has not been mapped in Spark for Accept

- Our server doesn't have an endpoint called "/"

- But our server does have other endpoints. Let's examine the code…
  - Open up `src/main/java/sparkDemo/SparkServer.java`

Hello, World

```
Spark.get("/hello-world", new Route() {
    @Override
    public Object handle(Request request,
                    Response response) throws Exception {
        // As a first example, let's just return
        // a static string.
        return "Hello, Spark!";
    }
});
```
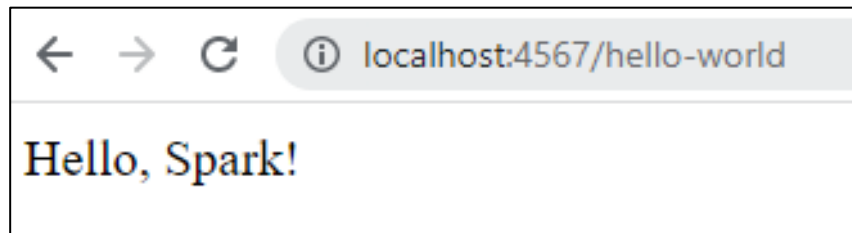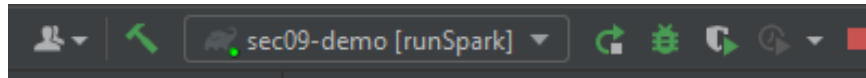
Create Your Own Route!

- Create your own endpoint!

```
Spark.get("/your-endpoint-here", new Route() {
    @Override
    public Object handle(Request request,
                         Response response) throws Exception {
        return "Your message here!";
    }
});
```
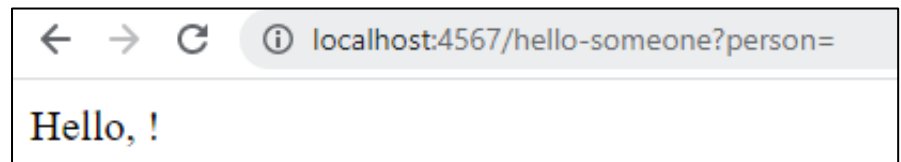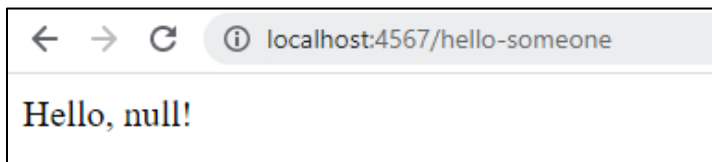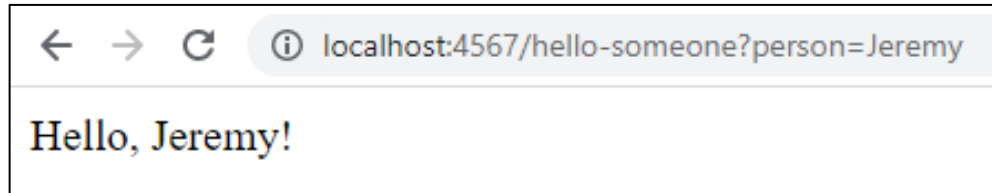

- When you're done, you'll need to restart the server. Use the **stop** button and re-run the **runSpark** Gradle task.
  - Visit your newly-created endpoint!

Query Parameters

```
Spark.get("/hello-someone", new Route() {

    @Override

    public Object handle(Request request,

                    Response response) throws Exception {

        String personName = request.queryParams("person");

        return "Hello, " + personName + "!";

    }

});
```
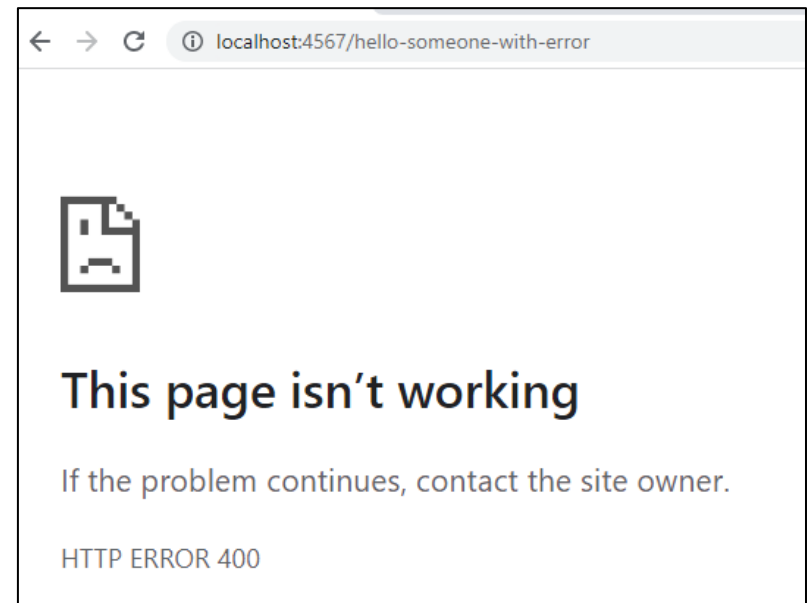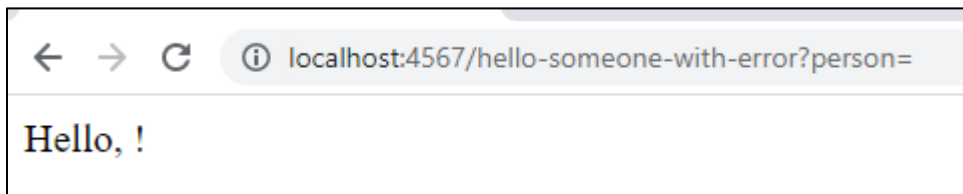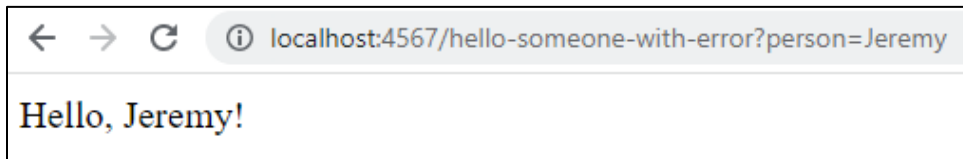
# Parameter Error Handling

```
Spark.get("/hello-someone-with-error", new Route() {

        ...

        String personName = request.queryParams("person");

        if (personName == null) { Spark.halt(400); }

        return "Hello, " + personName + "!";

    }

});
```


Hello, Jeremy!

Hello, !

This page isn't working

If the problem continues, contact the site owner.

HTTP ERROR 400

Sending Back a Simple Java Object

```java
Spark.get("/range", new Route() {

        ...

        List<Integer> range = new ArrayList<>();
        for (int i = start; i <= end; i++) {
            range.add(i);
        }
        Gson gson = new Gson();
        String jsonResponse = gson.toJson(range);
        return jsonResponse;
    }
});
```



```
← → C  ⓘ localhost:4567/range?start=1&end=10

[1,2,3,4,5,6,7,8,9,10]
```
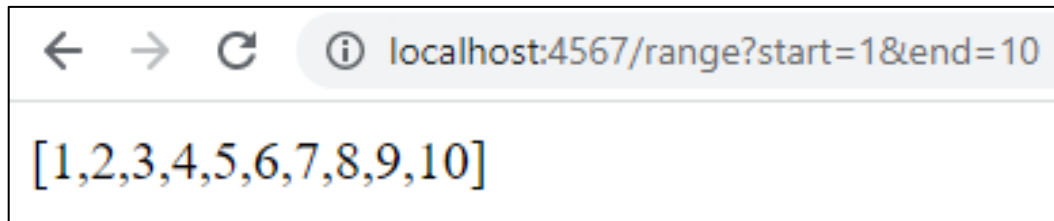
Sending Back a Simple Java Object
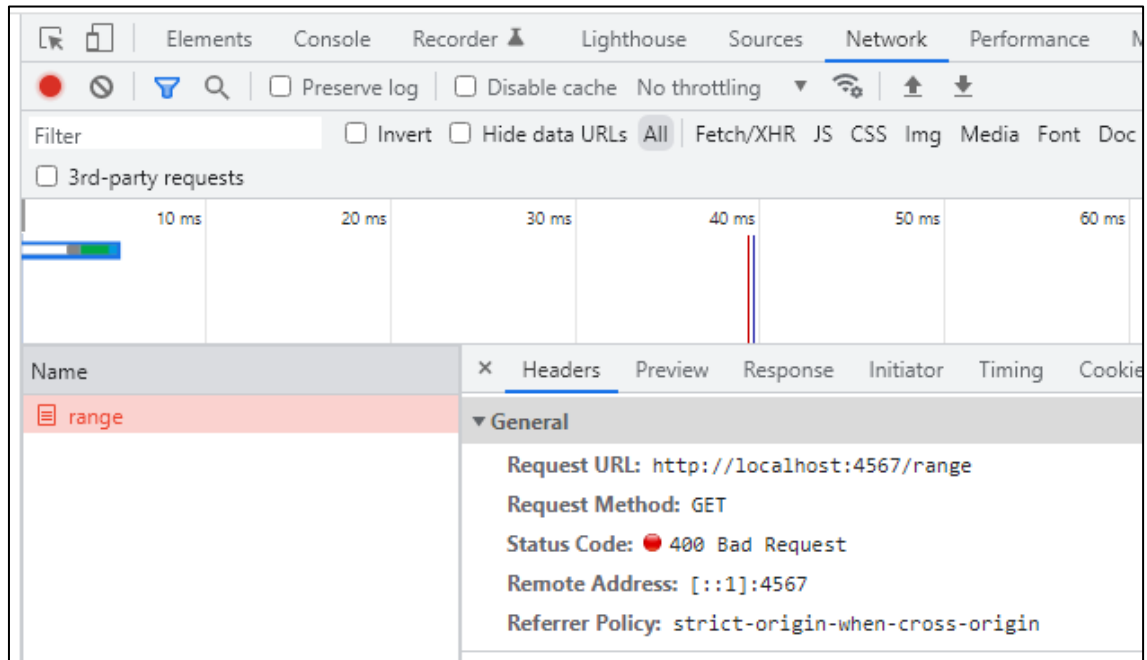
- Tip: Use the network tab to view requests and responses!
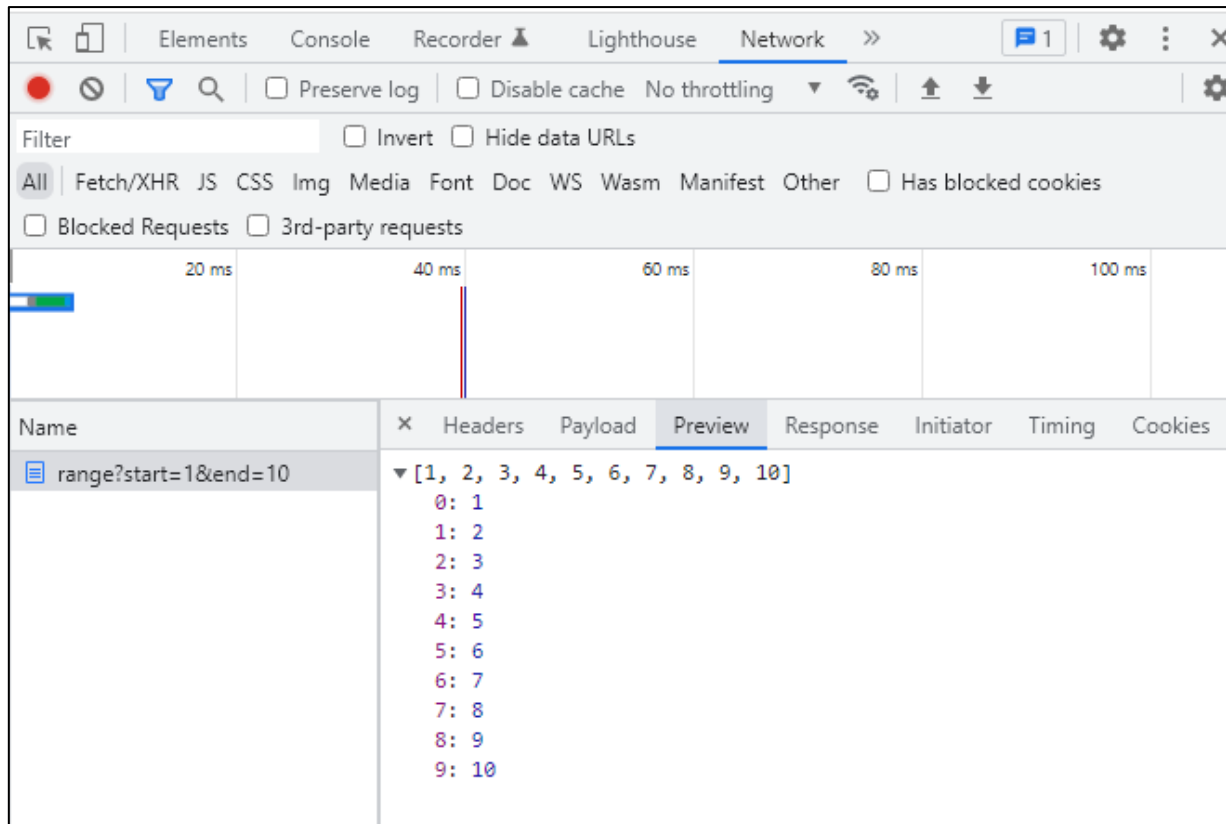
Sending Back a Simple Java Object

- Use descriptive and informative error messages!
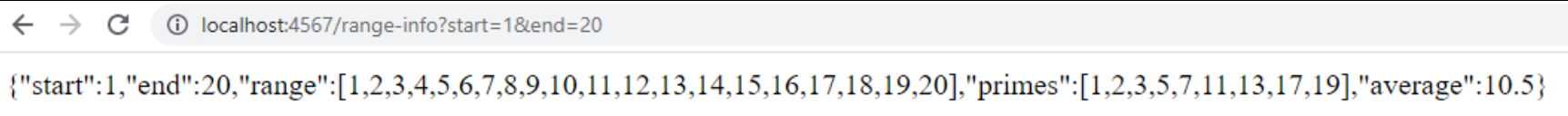
```
Spark.halt(400, "must have start and end");
```

- Limited freedom to pick a status #!
  - See the docs
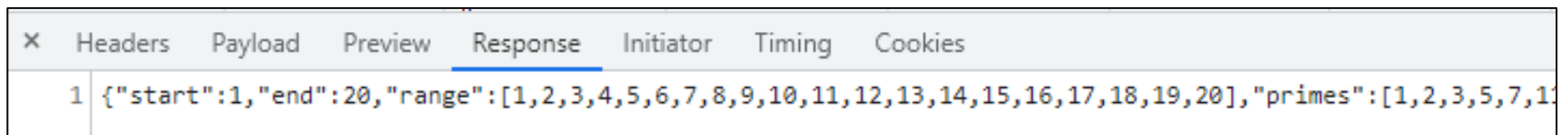
Sending Back a Complex Java Object

```java
Spark.get("/range-info", new Route() {

        ...

        // RangeInfo is a class with fields:

        // start, end, range, primes, average

        RangeInfo rangeInfo = new RangeInfo(start, end);

        Gson gson = new Gson();

        return gson.toJson(rangeInfo);

    }

});
```

← → C   ⓘ localhost:4567/range-info?start=1&end=20

{"start":1,"end":20,"range":[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20],"primes":[1,2,3,5,7,11,13,17,19],"average":10.5}

- The network tab also shows this!

✕   Headers   Payload   Preview   Response   Initiator   Timing   Cookies

1 {"start":1,"end":20,"range":[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20],"primes":[1,2,3,5,7,1

Fetch

- Used by JS to send requests to servers to ask for info.
  - alternative to `XmlHttpRequest`

- Uses Promises:
  - Promises capture the idea of "it'll be finished later."
  - Asking a server for a response can be *slow*, so Promises allow the browser to keep working instead of stopping to wait.
  - Getting the data out is a little more complicated.
  - Java has Promises too – called `CompletableFuture`

- Can use `async`/`await` syntax to deal with promises.

Sending the Request in React

```
let responsePromise = fetch("http://localhost:4567/findPath?start=CSE&end=KNE");
```

- The URL you pass to `fetch()` can include a query string if you need to send extra data.

- `responsePromise` is a Promise object
  - Once the Promise "resolves," it'll hold whatever is sent back from the server.

- How do we get the data out of the Promise?
  - We can `await` the promise's resolution.
  - `await` tells the browser that it can pause the currently-executing function and go do other things. Once the promise resolves, it'll resume where we left off.
  - Prevents the browser from freezing while the request is happening (which can take some time to complete)

Getting Useful Data

"This function is pause-able"

Will eventually resolve to an actual JS object based on the JSON string.

Once we have the data, store it in a useful place.

```
async sendRequest() {
    let responsePromise = fetch("...");
    let response = await responsePromise;
    let parsingPromise = response.json();
    let parsedObject = await parsingPromise;
    this.setState({
        importantData: parsedObject
    });
}
```

Error Checking

Every response has a 'status code' (404 = Not Found). This checks for 200-299 = OK

On a complete failure (e.g. server isn't running) an error is thrown.

Make sure you create **informative** and **helpful** error messages!

```javascript
async sendRequest() {
    try {
        let response = await fetch("...");
        if (!response.ok) {
            alert("Error message!");
            return;
        }
        let parsed = await response.json();
        this.setState({
            importantData: parsed
        });
    } catch (e) {
        alert("Error message!");
    }
}
```

Fetch Demo

Running the Fetch Demo

- Make sure your Spark Server is running (**runSpark** Gradle task)

- In the IntelliJ terminal:
  - Make sure you're in `src/main/react`
  - `npm start`

```
Compiled successfully!

You can now view sec09-demo in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.1.9:3000

Note that the development build is not optimized.
To create a production build, use npm run build.
```

- A browser window should open up automatically
  - Issues: have you run `npm install` yet?
  - If so, run `npm audit fix --force` then run `npm start`
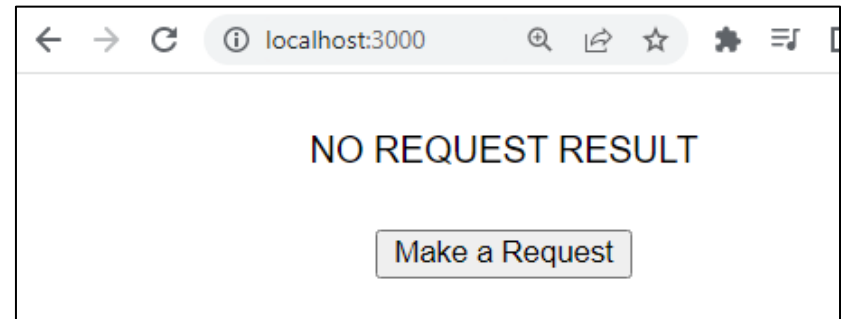
Fetch

App.tsx:

```
constructor(props: {}) {
  super(props);
  this.state = { requestResult: "NO REQUEST RESULT" };

}



render() {
  return (
    <div className="App">

      <p>{this.state.requestResult}</p>
      <button onClick={this.makeRequestLong}>

        Make a Request

      </button>

    </div>

  );

}
```

Fetch

```
makeRequestLong = async () => {
  try {
    let responsePromise = fetch("http://localhost:4567/
                                hello-someone?person=React");

    let response = await responsePromise;
    if (!response.ok) {
      alert("Error! Expected: 200, Was: " + response.status);
      return;
    }
    let textPromise = response.text();
    let text = await textPromise;
    this.setState({ requestResult: text });
  } catch (e) {
    alert("There was an error contacting the server.");
    console.log(e);
  }
};
```

Fetch

```
makeRequestLong = async () => {
  try {
    let responsePromise = fetch("http://localhost:4567/
                              hello-someone?person=React");
```

The type of this is
Promise<Response>

Do **NOT** use **https**

```
    let response = await responsePromise;
    …
};
```

await "resolves" a promise
(waits for the promise to be fulfilled)

The type of this is
Response

Fetch

```
makeRequestLong = async () => {

    ...

    if (!response.ok) {

      alert("Error! Expected: 200, Was: " + response.status);

      return;

    }

    ...

};
```

Stop the execution of this function if the response is bad.
Response objects have other fields too, such as:
- **.headers**
- **.statusText**
- **.url**

Check out the [docs](#) for more info on **Response** objects!

Fetch

```
makeRequestLong = async () => {

    ...

    let textPromise = response.text();
```

This endpoint returns a string (text). If your endpoint returns a JSON string, use `response.json()` instead.

Since we used `.text()`, the type of this is `Promise<string>`

```
    let text = await textPromise;

    ...
};
```

`Promise<string>` resolves into `string`. `text` is of type `string`.

Fetch

```
makeRequestLong = async () => {

    ...

    let text = await textPromise;

    this.setState({ requestResult: text });
```

We update the **state** with the response from the server!

```
  } catch (e) {

    alert("There was an error contacting the server.");

    console.log(e);

  }
};
```

Handle errors gracefully and inform the user of an error. Most common sources of errors:
- Fetch URL is wrong
- Server is offline
- Using `.json()` if the response doesn't contain valid JSON

Fetch

Recap:

- When we click the button, its `onClick` listener will call the callback function we passed in: `this.makeRequestLong`

- `this.makeRequestLong` sends a `fetch` request to our **Spark Server**: **http://localhost:4567/hello-someone?person=React**

- `this.makeRequestLong` receives a response from the server and updates `App`'s **state** 👀 Queue a re-render!

- React notices the **state** update and queues a re-render
- The `<p>` element is re-rendered with the updated **state**!

localhost:3000

Hello, React!

Make a Request
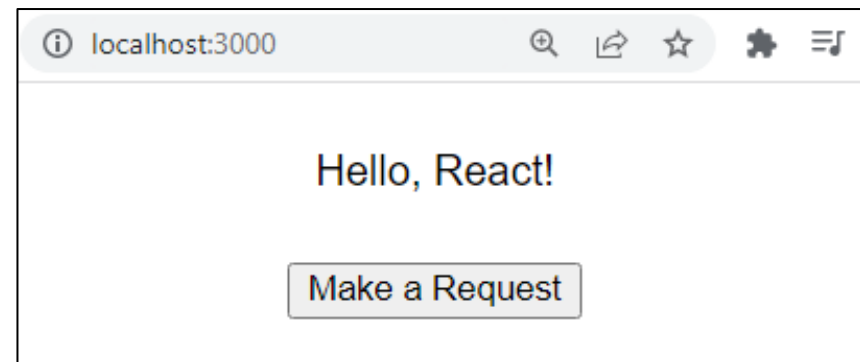
Fetch, but more compact

```
makeRequest = async () => {
  try {
    let response = await fetch("...");
    if (!response.ok) {
      alert("...");
      return;
    }
    let text = await response.text();
    this.setState({ requestResult: text });
  } catch (e) {
    alert("There was an error contacting the server.");
    console.log(e);
  }
};
```

Reduced the number of temporary variables!

Fetching objects

```
printRangeToConsole = async () => {
  try {
    let response = await fetch("...");
    if (!response.ok) {
      alert("...");
      return;
    }
    let text = (await response.json()) as number[];
    ...
  } catch (e) {
    alert("There was an error contacting the server.");
    console.log(e);
  }
};
```

Can use `.json()` and cast to some type

Things to Know

- Can only use the **await** keyword inside a function declared with the **async** keyword.

  - **async** keyword means that a function can be "paused" while **await**-ing

- **async** functions automatically return a Promise that (will eventually) contain(s) their return value.

  - This means that if you need a return value from the function you declared as **async**, you'll need to **await** the function call.

  - But that means that the caller also needs to be **async**.

  - Therefore: generally best to **not** have useful return values from **async** functions (in 331, there are lots of use cases outside of this course, but can get complicated fast).

  - Instead of returning, consider calling **setState** to store the result and trigger an update.

More Things to Know

- Error checking is **important**.

  - If you forget, the error most likely will disappear without actually causing your program to explode.

  - This is BAD! Silent errors can cause tricky bugs.

  - Happens because errors don't bubble outside of promises, and the `async` function you're inside is effectively "inside" a promise.

  - Means that if you don't catch an exception, it'll just disappear as soon as your function ends.

More More Things to Know

- The return value of `await response.json()` will be `any`
  - As we know, this is dangerous! (No TypeScript checks)
- To solve, we create an interface describing what the server will respond with (e.g. a `Path`) and *cast* the value to that type:

  ```
  interface Path { ... }

  const parsed: Path = await response.json() as Path;
  ```

- Note: This does not check that the value *actually has* this type
  - If the server sends back something different, could crash later
  - A true solution would check the object before casting
    - Can get pretty complicated – **not required** for HW9
    - If you're curious – libraries like `io-ts` can help with this

Any Questions?

- Done:
    - HW9 Overview
    - JSON
    - Fetch

Wrap-Up

- Don't forget:
    - HW9 due next week (Fri. 12/9 @ 11:00pm)

- Use your resources!
    - Office Hours
    - Links from HW specs
    - React Tips & Tricks Handout (See "Resources" page on the course website)
    - Other students (remember academic honesty policies: **can't share/show/copy code**, but discussion is great!)
    - Google (carefully, always fully understand code you use)