

---

# CSE 331

# Software Design & Implementation

Andrew Gies  
Fall 2020  
HW9, JSON, Fetch

# Administrivia

---

- HW8 due today (Thur. 12/3 @ 11:00pm)
- HW9 due a week later (Thur. 12/10 @ 11:00pm)
  - Spec released soon. 😊
  - Plan ahead - this assignment can take a little longer than others.
  - Get creative! Lots of cool opportunities.
  - Demo opportunity next week
- Any questions?

# Agenda

---

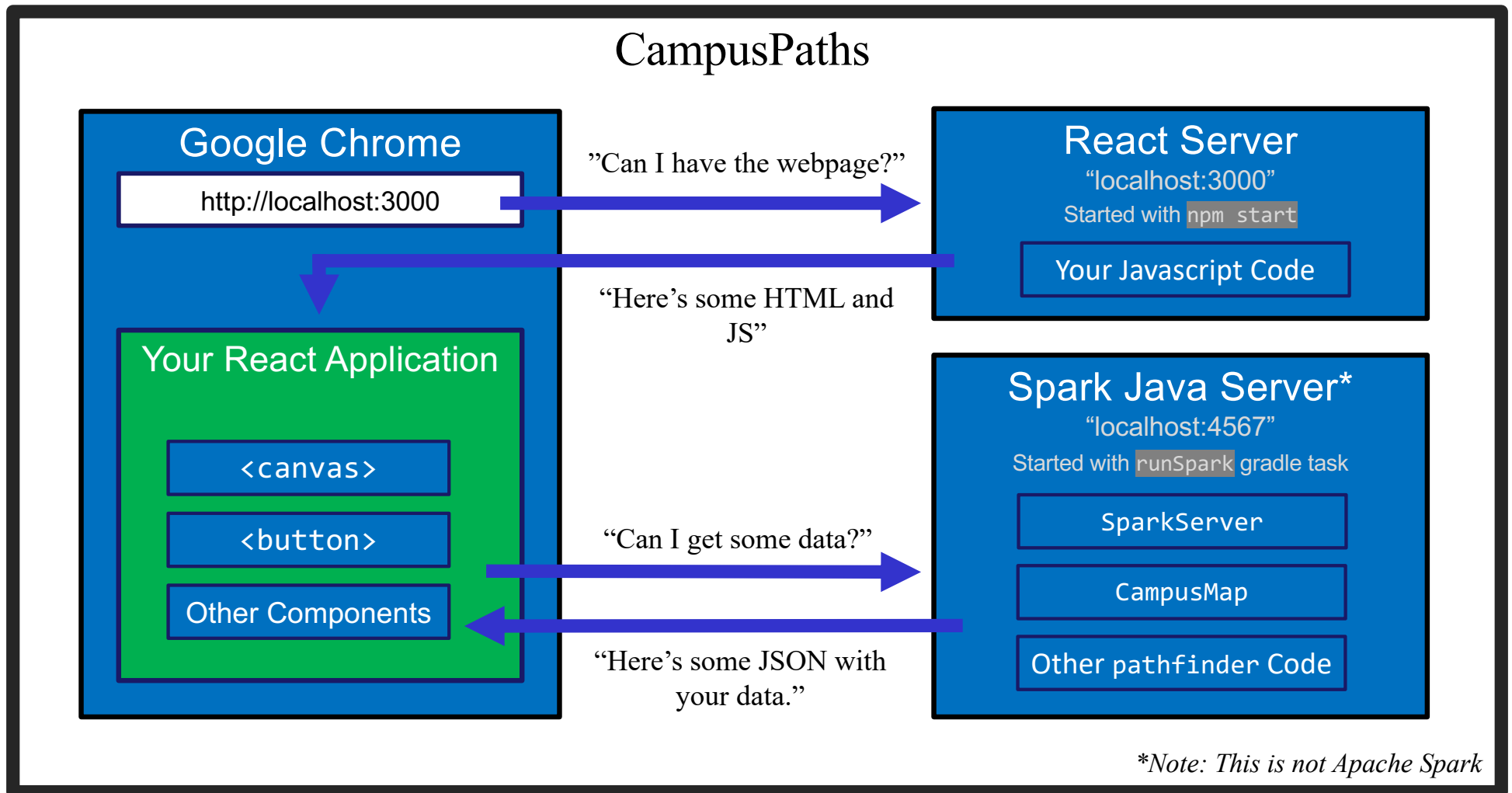
- HW9 Overview
- JSON
  - Brief overview
  - Helps share data between Java and JS.
- Fetch
  - How your JS sends requests to the Java server.

# Homework 9 Overview

---

- Creating a new web GUI using React
  - Display a map and draw paths between two points on the map.
  - Works just like your React app in HW8 – but you get to design it!
  - 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 Campus Paths Stack



# Any Questions?

---

- Done:
  - HW9 Basic Overview
- Up Next:
  - JSON
  - Fetch

# JSON

---

- We have a whole application written in Java so far:
  - Reads TSV data, manages a Graph data structure, manages building information, does Dijkstra’s algorithm.
- We’re writing a whole application in Javascript:
  - React web app to create a GUI for your users to interact with.
- 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 ↔ 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 ↔ JS

---

## Java Object

```
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","foun  
ded":1861,"mascot":"Dubs  
II","isRainy":true,"website":"www.uw.e  
du","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.

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

# 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.
- Easy to turn a JSON string into a Javascript object so we can use the data (node-fetch can help us with that).

# Any Questions?

---

- Done:
  - HW9 Basic Overview
  - JSON
- Up Next:
  - Fetch

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

# Creating 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...

# Creating a Request

Server Address: `http://localhost:4567`

- 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.

# Sending the Request

---

```
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

# 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 = OK

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

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

# 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.

# Any Questions?

---

- Done:
  - HW9 Basic Overview
  - JSON
  - Fetch

# Wrap-Up

---

- Don't forget:
  - HW8 Due This Week (Thurs 12/3 @ 11:00pm)
  - HW9 Due Next Week (Thurs 12/10 @ 11:00pm)
- Use your resources!
  - Office Hours
  - Links from HW specs
  - React Tips & Tricks Handout (See “Resources” page on web)
  - Other students (remember academic honesty policies: can't share/show/copy code, but discussion is great!)
  - Google (carefully, always fully understand code you use)