COURSEWARE

Professional Skills

Agile Fundamentals

Jira

Git

Databases Introduction

Java Beginner

Maven

Testing (Foundation)

Java Intermediate

HTML

CSS

Javascript

Spring Boot

Selenium

Sonarqube

Advanced Testing (Theory)

Cucumber

MongoDB

Express

NodeJS

React

- Introduction
- JSX
- Babel
- Component Hierarchy
- Components
- Props
- Lifecycle
- State
- Lifting State
- Hooks
- React Routing

Data Requests

Contents

- Overview
  - Making External Requests
  - HTTP Request and Response Structure
  - GET
  - POST
  - Status Codes
  - Fetch
  - Axios
- Tutorial
- Exercises

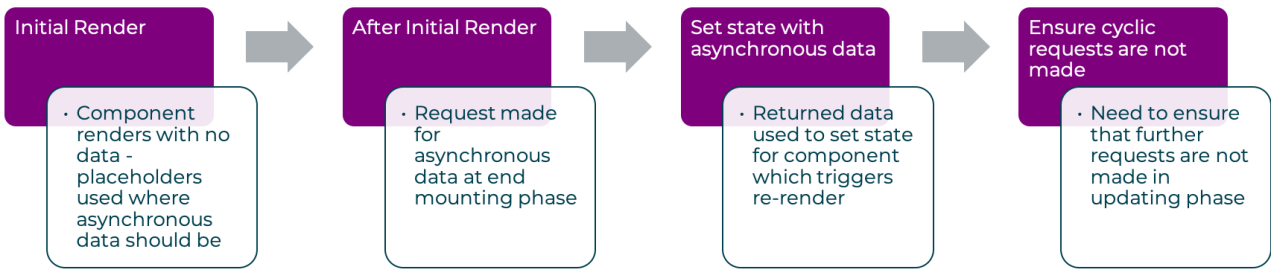
Overview

In this module, we will look at sending requests through React.

Making External Requests

When making external requests for data, we follow the below points:

- what happens in the initial render
- what happens after the initial render
- how are we setting the state
- how are we stopping requests when needed



HTTP requests and responses work very similarly to the above too!

- Data Requests
- Static Data
- State Management

Express-Testing

Networking

Security

Cloud Fundamentals

AWS Foundations

AWS Intermediate

Linux

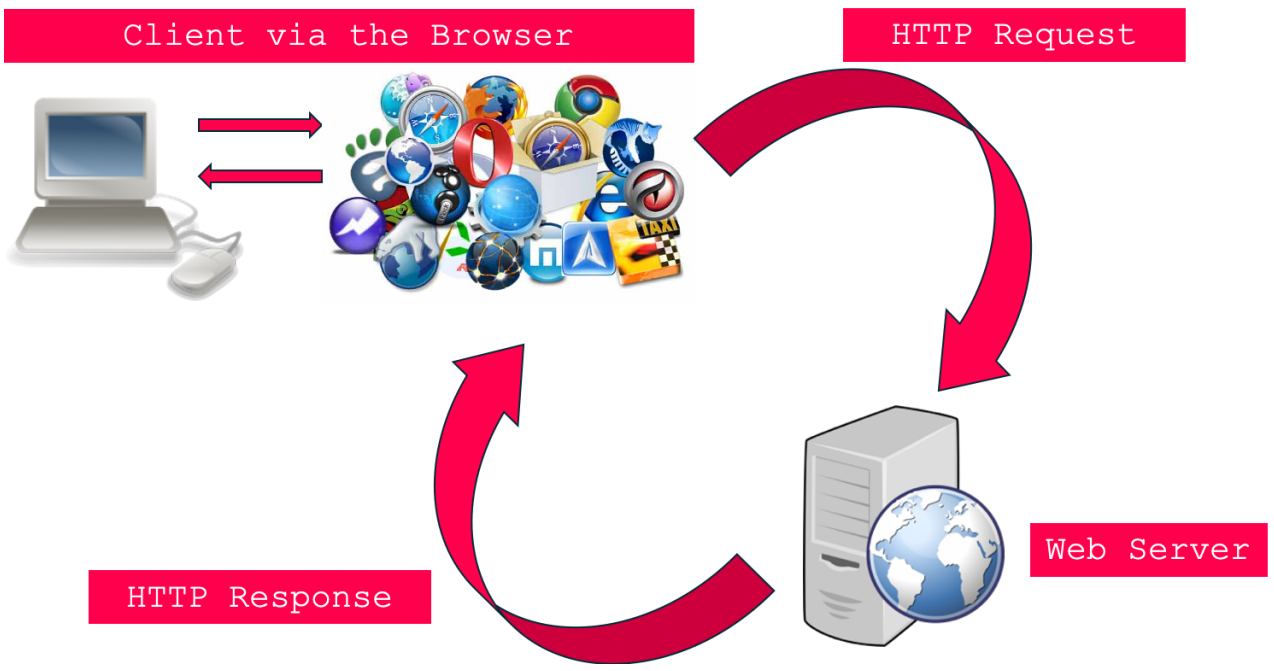
DevOps

Jenkins Introduction

Jenkins Pipeline

Markdown

IDE Cheatsheet



HTTP Request and Response Structure

HTTP requests consist of the following:

1. Request Line
2. Zero or more header fields followed by new line
3. An empty line indicating the end of the header fields
4. An optional message body (ie., data / content)

```
POST /php/myapplication.php HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5; Windows NT)
Host: www.example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: length
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive

key=value&username=example&email=me@example.com
```

HTTP responses have responses structured with the following:

1. A status line
2. Zero or more header fields followed by a new line
3. An empty line
4. An optional message body (eg. data, content)

```
HTTP/1.1 200 OK
Date: Mon, 21 Mar 2016 09:15:56 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Mon, 21 Mar 2016 09:14:01 GMT
Content-Length: 88
Content-Type: text/html
Connection: Closed
<html>
<body>
<h1>Hello, World!</h1>
</body>
</html>
```

GET

GET is a HTTP method that allows us to retrieve information for a given URI. GET requests have no request body.

The URI specifies most of the data the server requires:

- The protocol, eg. HTTP
- The host, eg. example.com
- The resource location, eg. /index.php
- The query string, eg. ?username=eg&email=foo@example.com
- Other information can be supplied with headers, either standard ones or custom headers typically preceded by "X-"

```
http://example.com/index.php?username=eg&email=foo@eg.com
```

POST

The POST HTTP method builds on a GET request by including a message body.

They can use the query string system and the message body to send data. Larger or more sensitive information is usually sent via POST.

The message body can have a practically unlimited size (usually limited by what the server will accept, eg. 20MB).

The browser sends the POST body "behind the scenes" whereas the URL is visible in the address bar.

POST messages are no more secure than GET messages, with both being sent via "plain text" over the network.

Status Codes

1. Informational responses, such as processing still going on (e.g. 102)
2. Successful responses (e.g. 200)
3. Redirection - resource has moved to a different location (e.g. 301)
4. Client error - problem with the requesting client (e.g. 404)
5. Internal Server Error - problem with the responding server (eg. 500)

Fetch

Fetch API provides an interface for fetching resources (including across the network).
It will seem familiar to anyone who has used XMLHttpRequests, but the new API provides a more powerful and flexible feature set.

The `fetch()` takes one mandatory argument, the path to the resource you want to fetch.
It returns a `Promise` that resolves to the `Response` to that request, whether it is successful or not.

Once a response is retrieved, there are a number of methods available to define what the body content is and how it should be handled (see Body).

Fetch is a two step process when dealing with JSON data; after making an initial request you'll then need to call the `.json()` method on the response in order to receive the actual data object.
If I were to have a fetch request to get back an object of relevant drivers for my carpool sourcing app, I might have something like the following:

```
const baseURL = 'http://localhost:3000/api/v1'
fetch(`${baseURL}/drivers`).then(res => res);
```

This would return a pending promising and response object letting me know my request was successful, but where's my data?!
We have to remember to pass our response to the `.json()` method aswell

```
const baseURL = 'http://localhost:3000/api/v1'
fetch(`${baseURL}/drivers`).then(res => res.json());
```

Axios

▶ Installation
Axios is another method to retrieve data.
In a nutshell, Axios is a JavaScript library used to make HTTP requests from node.js or XMLHttpRequests from the browser.

Axios aims to improve the process of `.fetch()`

To achieve the same fetch request we performed earlier, we can invoke `axios.get()` in place of our fetch method like so:

```
const baseURL = 'http://localhost:3000/api/v1'
axios.get(`${baseURL}/drivers`).then(res => res);
```

Presto! By using axios we remove the need to pass the result of the HTTP request to the `.json()` method.
Axios simply returns the data object you expect straight away.
Additionally, any kind of error with HTTP request will successfully execute the `.catch()` block right out of the box.

In summation, Axios is a quality of life improvement more than anything else. Making many small, incremental quality of life adjustments to one's workflow can drastically improve development quality and speed.

More on Axios can be found here

Tutorial

In this tutorial, we are going to be importing employee information from an external source.

1. Create a const called `EmployeeInfo` which takes no parameters as arguments:

```
const EmployeeInfo = () => {}
```

2. Define the following states with the respective default values:
    1. error = null
    2. isLoaded = false
    3. items = empty array

```
const [error, setError] = useState(null);
const [isLoaded, setIsLoaded] = useState(false);
const [items, setItems] = useState([]);
```

3. Call `useEffect ()`, create an anonymous function and within make a `axios.get()` call to "http://dummy.restapiexample.com/api/v1/employees":

```
useEffect(() => {
    axios
    .get("http://dummy.restapiexample.com/api/v1/employees")
```

4. Chain a `.then()` method which takes `res` as parameter and outputs `res`:

```
 .then(res => res)
```

5. Then, chain another `.then()` method with a `result` parameter as argument - within this function:
    1. Set the `isLoaded` state to true
    2. Set the `setItems` array to the data from result

```
.then((result) => {
 setIsLoaded(true);
 setItems(result.data.data);
},
```

6. In the same method, handle any errors; if there are any errors set the `error` state to `true`:

```
.then((result) => {
    setIsLoaded(true);
    setItems(result.data.data);
},
    // Note: it's important to handle errors here
    // instead of a catch() block so that we don't swallow
    // exceptions from actual bugs in components.
    (error) => {
      setIsLoaded(true);
      setError(error);
    }
  );
```

7. Specify an empty array at the end of the `useEffect()` method so it runs once, similar to `componentDidMount()`:

```
...
}, []);
```

8. Check the property of `error`; if it is true, return `'Error: ${error.message}'` inside a `<div>`:

```
if (error) {
    return <div>Error: {error.message}</div>;
}
```

9. Create an `else if` property to check if `isLoaded` is true:

```
else if (!isLoaded) {
    return <div>Loading...</div>;
}
```

10. `Else`, return the `items` state property in a list and print the employeeNames:

```
else {
    return (
    <ul>
        {items.map((item) => (
        <li key={item.id}>
            {item.employee_name}
        </li>
        ))}
    </ul>
    );
}
```

11. Export as default, import into `app.js` and run:

▶ EmployeeInfo.jsx

Exercises

- In this exercise, you are tasked with creating a movie search application.

    1. Navigate to OMDBAPI
    2. Get a free API key from the API Key tab
    3. Create a `FilmRequest` Component that lets the user enter a Movie title and axios fetches the correct movie from OMDBAPI.

       Try and use Parent/Child Components to present your data

       ▶ Basic Solution