devCodeCamp

Axios

Web Services

- A web service is a software application that's accessible over a network (the Internet).
- Web services communicate with each other using HTTP requests & responses.
- HTTP acts as a common language between these services, regardless of the programming language they're developed in.
- They operate on the client-server model, where a server provides access to resources that client apps consume by request.

Web Services: Servers

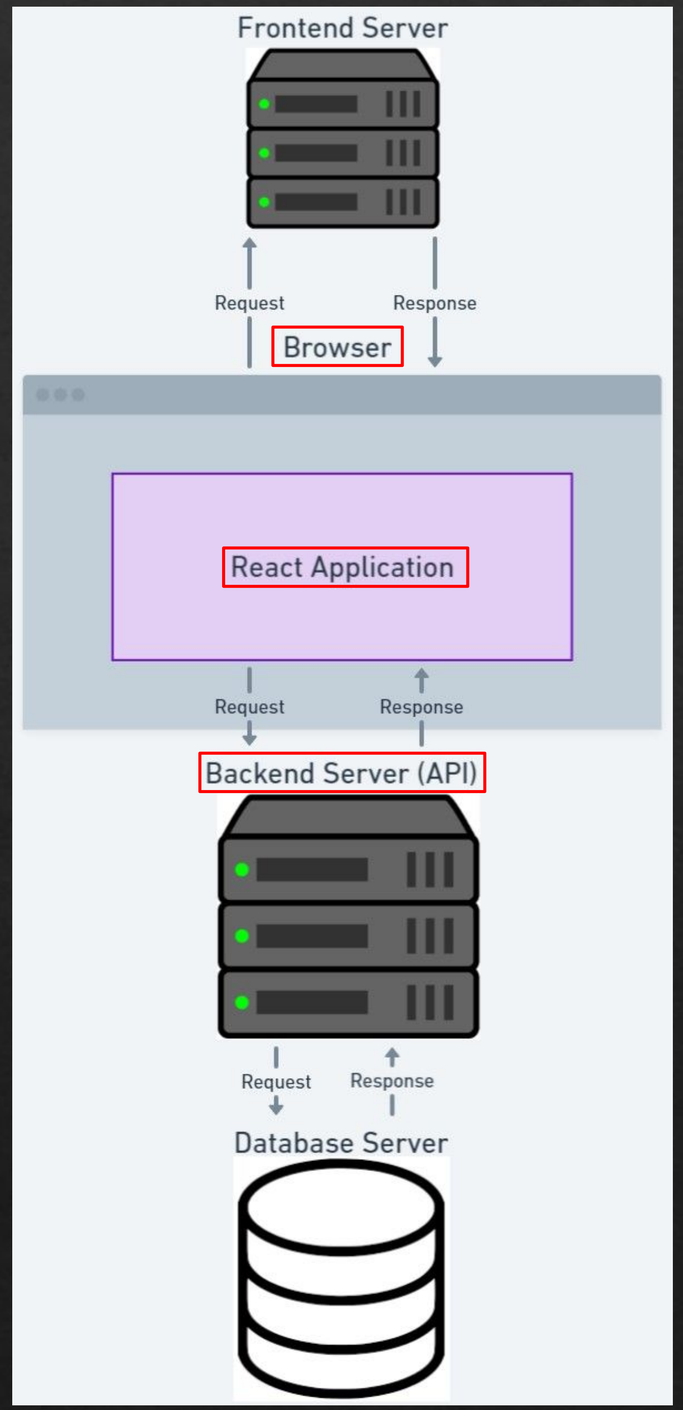
- A server is an application that exposes an interface through URLs (endpoints) where clients can send their requests.
- This interface is a set of functions that handle incoming requests and then respond with specific resources (JSON, HTML, etc).
- A REST API is a server that responds with raw JSON data, which client applications consume.
- A web server has endpoints that respond with webpages or complete web apps consumed by a browser.

Web Services: Clients

- A client is *any* application that sends requests to a server's endpoints, consuming the data returned in the server's response.
- Some examples of client applications include:
  - Postman
  - MySQL Workbench
  - Web browsers (Chrome, Safari, Firefox, Edge)
  - Mobile applications
- Even servers can double as clients, requesting data from another server while simultaneously responding to client requests.

Full Stack Applications

- Full stack applications typically consist of three servers:
  - The frontend server is a web server that serves a web app to a browser. This web app requests raw JSON data from a backend server.
  - The backend server is a REST API that serves JSON data to the web app by requesting data from a database server.
  - The database server serves JSON data to the backend directly from a database.
- How many clients are there in this full stack example?



React: Development Server

- React apps are web applications "served up" by a web server at localhost:3000.
- npm start activates this web server during React development (development server).
- A browser then makes a GET request to it, which responds with the entire React application that's displayed to the user.
- Using JS libraries like fetch or axios, React apps can send requests to servers using asynchronous functions, which require special syntax.

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

PS D:\workspace\devCodeCamp\CONTENT_CREATION\ReactDebugging> npm start

> debug-sandbox@0.1.0 start
> react-scripts start
Compiled successfully!

You can now view debug-sandbox in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.8.148:3000

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

webpack compiled successfully
```

Async Functions

- JavaScript code runs synchronously by default, meaning tasks are performed one at a time, in a specific order.
- However, sending network requests can take some time, causing an application to “freeze” while waiting for a response.
- Asynchronous functions can run in the background, so the app can run other tasks while waiting for the function to complete.
- Async functions are declared with the “async” keyword, which is shorthand for creating functions that return a Promise object.

```
async function fetchData(){
    // request logic
}

const fetchData = async () => {
    // request logic
}
```

Promises

- A Promise is a JavaScript object that holds the eventual completion or failure of an async operation, like sending a request to an API.
- Promises exist in one of three states during a request/response lifecycle:
  - Pending: The request is sent and the Promise is awaiting a response from a server; its initial state.
  - Fulfilled: The request succeeded and a response object is returned.
  - Rejected: The request failed and an error object is returned.
- Async functions return these Promises, which eventually return a response object from the function using the await keyword.

Await

- We must prepend async function calls with await which can only be used within other async functions.
- Without it, the Promise is instantly returned instead, before the response had time to return from the server.
- Under the hood, await suspends further execution of the function until the Promise has been fulfilled or rejected.
- Meanwhile the rest of the program continues its normal, synchronous execution of the code.

```
const fetchData = async () => {
    // Calling an async function
    const response = await axios.get('<endpoint_url>')
}
```

Axios

- Axios is a JavaScript library used to make HTTP requests from browser-based applications.
- It can easily be installed into any Node app with npm install axios and then imported into modules to access its methods.
- Each axios method is named after its respective HTTP method (get, post, put, patch & delete).
- These are all async methods that require the async/await keywords when used.

```
import axios from 'axios';

const App = () => {
    // Primary axios methods
    axios.get()
    axios.post()
    axios.put()
    axios.patch()
    axios.delete()
```

Axios: Method Parameters

1. The first param of an axios method is the endpoint URL where the request is to be sent.
2. The data param is how we send data in the request body, typically in the form of an object (only for POST, PUT & PATCH).
3. The config param is an optional object for further customizing the request (more to come).

```
response = await axios.get(url, config)

response = await axios.post(url, data, config)

response = await axios.put(url, data, config)

response = await axios.patch(url, data, config)

response = await axios.delete(url, config)
```

Axios: The Response Object

- All axios methods return a response object with properties detailing the server's response.
- response.data is the property you'll interact with most, containing the response body.
- response.status holds its status code, which is vital for debugging bad requests with error handling.
- Logging the response always helps to visualize what data it contains.

```
const fetchData = async () => {
    const response = await axios.get('<url>');
    console.log(response);
};
```

```
                                        App.jsx:35
{data: Array(2), status: 200, statusText:
'', headers: AxiosHeaders, config: {…}, …}
  i
  ▶ config: {transitional: {…}, adapter: 'xhr'
  ▶ data: (2) [{…}, {…}]
  ▶ headers: AxiosHeaders {content-type: 'appl
  ▶ request: XMLHttpRequest {onreadystatechang
    status: 200
    statusText: ""
  ▶ [[Prototype]]: Object
```

Axios: Error Handling

- We wrap axios methods in a try-catch block to debug requests in development.
- Responses with a status code in the 400's or 500's are rejected in the Promise, returning an error object.
- We can catch this error and log it to debug or display to the user if necessary.
- Logging the error or its properties can reveal valuable information about it when debugging.

```
const fetchData = async () => {
    try {
        const response = await axios.get('https://localhost:7185/api/books/4');
        console.log(response);
    } catch (error) {
        console.log('Error in fetchData: ', error.response.data);
    }
};
```

```
GET https://localhost:7185/a  xhr.js:251
pi/books/4 404

Error in fetchData:                App.jsx:23
  {type: 'https://tools.ietf.org/html/rfc7231
  #section-6.5.4', title: 'Not Found', statu
  s: 404, traceId: '00-50db95b41f249306567b10
  98c2b5875c-97f9705af163c0ca-00'}
    status: 404
    title: "Not Found"
    traceId: "00-50db95b41f249306567b1098c2b58
    type: "https://tools.ietf.org/html/rfc7231
  [[Prototype]]: Object
```

Side Quest: Status Codes

- Successful Responses
  - 200 Ok: The server returned the requested data.
  - 201 Created: The server created a new resource (POST).
  - 204 No Content: The server has no data to return.
- Client Errors (400's) mean there is a problem with the request.
  - 400 Bad Request: The server couldn't understand the request.
  - 404 Not Found: The endpoint or resource wasn't found on the server.
  - 405 Method Not Allowed: The endpoint doesn't accept the HTTP method.
- Server Errors (500's) indicate an error has been thrown server-side.
  - 500 Internal Server Error: A generic error message for server errors.

Axios: Fetching Data

- "Fetching" refers to retrieving API data with a GET request.
- We write an async function, which calls and awaits the axios.get() method within a try block.
- The call is returned to a response variable, and then response.data is assigned to state.
- This fetch function is called from a "mount effect" to update the state on a component's initial render.
- A console.log() can reveal the response data when the component updates.

```
const App = () => {
    const [books, setBooks] = useState([]);

    console.log(books); // Log books state on every render

    const fetchBooks = async () => {
        try {
            const response = await axios.get('https://localhost:7185/api/books');
            setBooks(response.data);
        } catch (error) {
            console.error('Error in fetchBooks:', error.response.data);
        }
    };

    // fetch books on mount
    useEffect(() => {
        fetchBooks();
    }, []);
```

```
▶ []                                                    App.jsx:16

                                                        App.jsx:16
▼ (2) [{…}, {…}]
  ▶ 0: {id: 1, title: 'The Stars', author: 'Ne
  ▶ 1: {id: 2, title: 'The Planets', author: '
    length: 2
  ▶ [[Prototype]]: Array(0)
```

Axios: Sending Data

- POST, PUT & PATCH requests require data to be sent along in their body.
- We create an async onSubmit handler for a form that bundles the user input into an object.
- This object is passed into the second argument (data) of axios.post().
- If the response status is 201 Created, the handler calls the fetch function lowered into the component's props to update the books state.

```jsx
// App.jsx
return (
    <div>
        <h1>Book Depot</h1>
        <NewBookForm onNewBook={fetchBooks} />
        <BookTable books={books} />
    </div>
);
```

```jsx
const NewBookForm = ({ onNewBook }) => {
    const [title, setTitle] = useState('');
    const [author, setAuthor] = useState('');
    const [datePublished, setDatePublished] = useState('');
    const [pages, setPages] = useState('');

    const handleSubmit = async (e) => {
        e.preventDefault();

        const newBookObj = {
            title,
            author,
            datePublished,
            pages,
        };

        try {
            const response = await axios.post('https://localhost:7185/api/books', newBookObj);
            if (response.status === 201) {
                onNewBook();
            }
        } catch (error) {
            console.error('Failed to add new book', error.response.data.errors);
        }
    };
```

Review

- What is the client-server model?
- What are the three servers in a full stack application?
- What does async do for a function?
- What does await do when calling an async function?
- How do you access a response body from a completed axios request?
- Why do we need a try-catch for axios methods?