APIs and Events

Topics covered:

- What are APIs?
    - Protocols of APIs
    - API requests
- How to add APIs in react?
    - Using Fetch
    - Using Axios
- Difference between Axios and fetch

1. **APIs**:

   The API defines commands, functions, protocols, and objects that programmers can use to build software or interact with external systems.
   - APIs enable various unrelated software products to integrate and communicate with other applications and data.
   - APIs also enable developers to add features and functionality to software by using the APIs of other developers.
   - A wide variety of APIs is used in today's workplace, mobile, and web software.
   - However, not all APIs are created equal. Developers can work with a variety of API types, protocols, and architectures to meet the specific demands of various applications and enterprises.

   a. **Protocols of APIs:**

      APIs transmit instructions and data, which requires defined protocols and architectures—the guidelines, frameworks, and limitations that control an API's operation. REST, RPC, and SOAP are the three major categories of API protocols or architectures used today. These may all be referred to as "formats," each with distinctive properties and trade-offs that are used for various objectives.
      - **REST:** The most popular technique for developing APIs is the *REpresentational State Transfer* (REST) architecture.
        REST is based on a client/server model that separates both the front and back ends of the API and allows for a great deal of freedom in development and implementation.

Because REST is stateless, the API retains no data or status between requests. For slow or non-time-sensitive APIs, REST enables caching, which caches answers.

REST APIs, also known as **RESTful APIs**, can communicate directly or via intermediary systems such as API gateways and load balancers.

- **RPC:** The *Remote Procedure Call* (RPC) protocol is a straightforward method for sending and receiving several

  The remote procedure call (RPC) protocol is a straightforward method for sending and receiving several. RPC APIs perform executable operations or processes, whereas REST APIs primarily exchange data or resources like documents.

  RPC can use two alternative languages for coding: JSON and XML; these APIs are known as JSON-RPC and XML-RPC, respectively.
- **SOAP:** The World Wide Web Consortium created the *simple object access protocol (SOAP)*, a messaging standard that is widely used to establish web APIs, typically with XML.

  Numerous internet-based communication protocols, including HTTP, SMTP, and TCP/IP, are supported by SOAP.

  Developers can easily add features and functionality to SOAP APIs since SOAP is flexible and independent of writing styles.

  The SOAP method specifies the features and modules that go into a SOAP message, the supported communication protocol(s), how SOAP messages are constructed, and how the SOAP message is handled.

b. API request:

The HTTP request types GET, POST, PUT, PATCH, and DELETE are the most frequently used ones. In other words, these operations are CRUD (create, read, update, and delete).

- **GET**: A GET request reads/retrieves data from a web server. If the data is successfully retrieved from the server, GET returns an HTTP status code of 200 (OK).
- **POST**: A POST request is used to transmit data to the server (file, form data, etc.). It returns an HTTP status code of 201 upon successful creation.

- **PUT**: To update data on the server, submit a PUT request. It passes data that is contained in the body payload in place of the complete content at a specific position. It will create one if there are no resources that match the request.
- **PATCH**: A PATCH request alters a portion of the data, whereas a PUT request modifies the entire request. Only the content you want to update will be replaced.
- **DELETE**: A DELETE request is used to remove data from a specific location on a server.

2. **Add APIs in React:**
    a. **Using Fetch:**
        - A built-in JavaScript method for obtaining resources from a server or API endpoint is fetch(). It is comparable to XMLHttpRequest, except the fetch API is more powerful and flexible.
        - It introduces topics like CORS and the HTTP Origin header semantics, which it replaces with distinct definitions elsewhere.
        - The fetch() API method always requires a mandatory argument, which is the path or URL of the resource to be retrieved. Whether the request is successful or not, it delivers a promise pointing to the response. As the second argument, you can optionally send in an init options object.

    Parameters for fetch():

        - **resource**

          This is the route to the resource you wish to retrieve; it may be a request object or a straight link to the resource path.
        - **init**

          This is an object that contains any special parameters or login information you want to include with your fetch() call. Some of the choices that could be included in the init object include the following:
            - **method:** This is where the HTTP request method, such as GET, POST, etc., is specified.
            - **headers:** You can use this to include any headers to your request that are typically contained in an object or an object literal.

- **body:** You can add a Blob, BufferSource, FormData, URLSearchParams, USVString, or ReadableStream object as the body of your request by providing it here.
- **mode:** You can use this to indicate the request mode, such as cors, no-cors, or same-origin.
- **credentials:** This option must be provided if you think about sending cookies automatically for the current domain. It allows you to define the request credentials you want to use for the request.

SYNTAX PRINCIPLES FOR USING THE FETCH() API

Look at the code below to see how easy it is to build a basic fetch request:

```

fetch("https://api.github.com/users/hacktivist123/repos")
  .then((response) => response.json())
  .then((data) => console.log(data));
```

The above code retrieves data from a URL that delivers it as JSON, prints it to the console, and does so. The path to the resource you wish to fetch is typically the only argument required for the simplest version of using fetch(), which then returns a promise with the result of the fetch request. An object, that is, is the reaction.

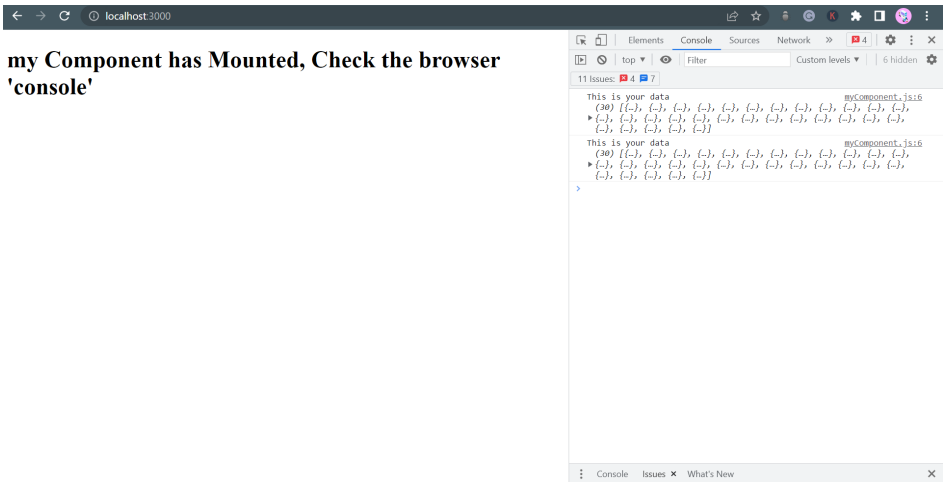
Instead of the actual JSON, the response is merely a standard HTTP response. We would need to use the response's json() function to convert the response to actual JSON in order to extract the response's JSON body content.

```
import React from "react";

class ApiComponent extends React.Component {
  componentDidMount() {
    const apiUrl = "https://api.github.com/users/hacktivist123/repos";
    fetch(apiUrl)
      .then((response) => response.json())
      .then((data) => console.log("This is your data", data));
  }
  render() {
    return <h1>my Component has Mounted, Check the browser 'console' </h1>;
  }
}
export default ApiComponent;
```

The code above creates a very straightforward class component that, once the React component has done mounting, sends a fetch request to the API URL and logs the results into the browser console.

The resource's path, which is stored in a variable called apiUrl, is sent to the retrieve() method. Following the successful completion of the retrieve request, a promise containing a response object is returned. Then, we use the json() method to extract the response's JSON body content, and finally, we log the promise's final value into the console.



A Class Component to a Function Component:

- useState and useEffect are hooks that are used to keep local states in function components.
- useEffect is a function that is used to execute functions after a component has been rendered (to "perform side effects"). useEffect can be restricted to circumstances in which a specific collection of values changes. These values are known as 'dependencies'.
- useEffects combines the functions of componentDidMount, componentDidUpdate, and componentWillUpdate.
- These two hooks essentially provide all of the conveniences formerly provided by class states and lifecycle methods.
- So, instead of a class component, let's refactor the App to a function component.

useEffect and useState

- useState is a hook that is used to keep local states in function components.
- useEffect is a function that is used to execute functions after a component has been rendered (to "perform side effects"). useEffect can be restricted to circumstances in which a specific collection of values changes. These values are known as 'dependencies'.
- useEffects combines the functions of componentDidMount, componentDidUpdate, and componentWillUpdate.
- These two hooks essentially provide all of the conveniences formerly provided by class states and lifecycle methods.
- So, instead of a class component, let's refactor the App to a function component.
- A state-based local array is managed by the useState.
- On component render, the useEffect will send a network request. When that fetch resolves, the setState function will be used to set the server response to the local state. The component will then render as a result, updating the DOM with the new data.

```jsx
import React from "react";
import { useEffect, useState } from "react";

export default function ApiComponent() {
  const [data, setData] = useState([]);
  const myStyle = {
    listStyle: "none",
    marginRight: "auto",
    marginLeft: "auto",
    border: " 0.1px  solid black",
    width: 400,
    textAlign: "justify",
  };

  useEffect(() => {
    const apiUrl = "https://api.github.com/users/hacktivist123/repos";
    fetch(apiUrl)
      .then((response) => response.json())
      .then((data) => {
        setData(data);
        console.log(data);
      });
  }, []);
  return (
    <div style={{ textAlign: "center" }}>
      <h1>Available Repositories</h1>
      <ul style={myStyle}>
        {data.map((d) => (
          <li style={{ padding: 5 }}>
            {d.name} {d.id}
          </li>
        ))}
      </ul>
    </div>
  );
}

```

Available Repositories

Akintayoshedrack.me 111079785
Algorithm-Challenges 126191710
Angular-Blog 137686365
Angular-Features 137689252
awesome-app-building-tutorials 96143889
Awesome-Cloud-Foundry 355944223
awesome-cloud-native 355944822
awesome-code-review 188363277
awesome-documentation-tools 96142505
awesome-meanstack 96143920
awesome-nextjs 96140554
awesome-opensource-documents 303984891
base16-iterm2 235361951
berkshire-deck-demo 264793845
black-speakers-in-tech 160467559
Blunt 266178711
clean-code-javascript 262692557
cloudfoundry-nodejs-app 297975380
cloudfoundry-nodejs-tutorial-pt-3 304160778

b. Using Axios to Access APIs:

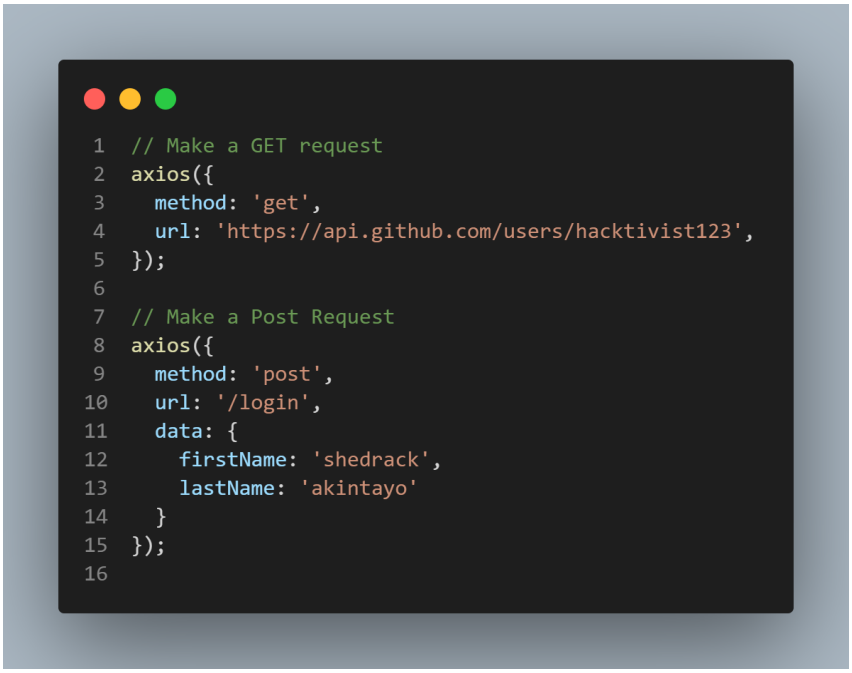
Axios is a basic promise-based HTTP client for sites, browser and nodes.js. Because Axios is promise-based, we can use async and await to write more understandable and asynchronous code. Axios provides the ability to intercept and cancel requests, as well as a built-in functionality that protects against cross-site request forgery.

AXIOS FEATURES

- Interception of requests and responses.
- streamlined handling of errors.
- defense against XSRF.
- assistance with upload progress.
- Response delay.
- being able to refuse requests.
- support for legacy browsers
- automatic transformation of JSON data.

REQUEST MAKING WITH AXIOS #

Making HTTP requests in Axios is simple. The code below demonstrates how to send an HTTP request.

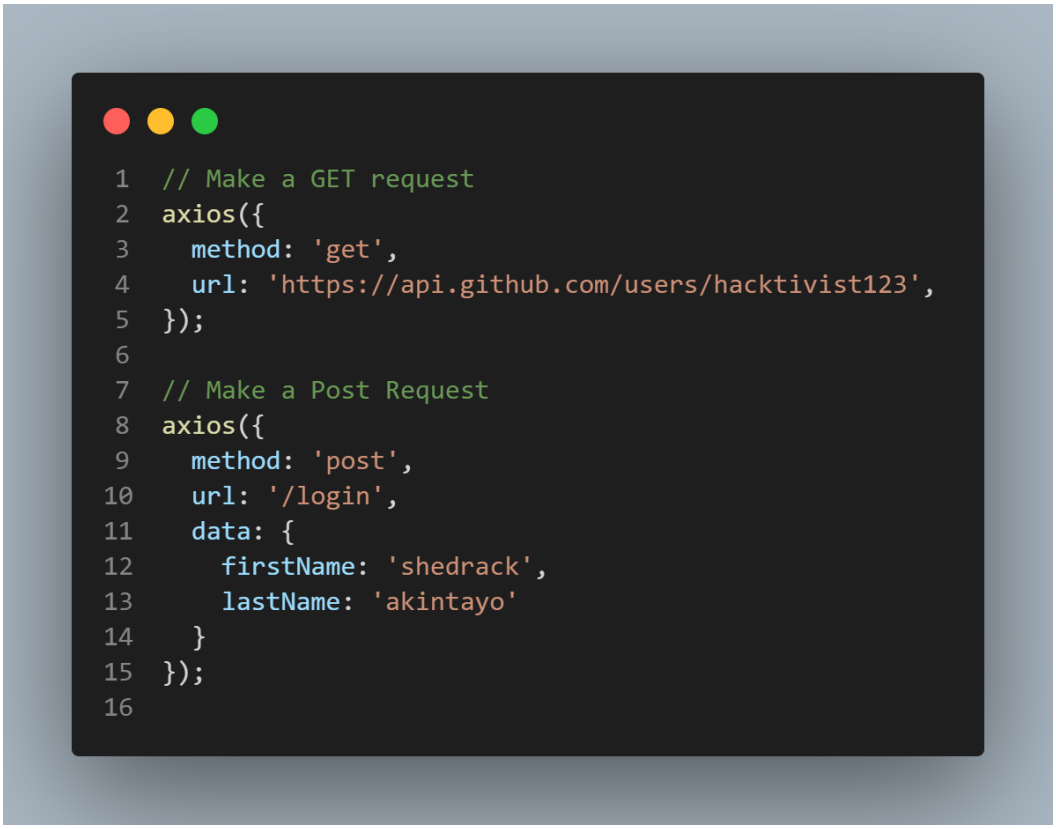


```
// Make a GET request
axios({
  method: 'get',
  url: 'https://api.github.com/users/hacktivist123',
});

// Make a Post Request
axios({
  method: 'post',
  url: '/login',
  data: {
    firstName: 'shedrack',
    lastName: 'akintayo'
  }
});
```

- The code above demonstrates the fundamentals of making a GET and POST HTTP request with Axios.
- Axios also has a collection of shorthand methods for performing certain HTTP requests. The procedures are as follows:
    - axios.request(config)
    - axios.get(url[, config])
    - axios.delete(url[, config])
    - axios.head(url[, config])
    - axios.options(url[, config])
    - axios.post(url[, data[, config]])
    - axios.put(url[, data[, config]])
    - axios.patch(url[, data[, config]])

  For example, if we wish to make a request similar to the example code above but using the abbreviated methods, we can do so as follows:



```
// Make a GET request
axios({
  method: 'get',
  url: 'https://api.github.com/users/hacktivist123',
});

// Make a Post Request
axios({
  method: 'post',
  url: '/login',
  data: {
    firstName: 'shedrack',
    lastName: 'akintayo'
  }
});

```

In the code above, we make the same request as before, but this time using the abbreviated method. Axios adds flexibility and makes HTTP requests more readable.

Let's Use Axios Client to Consume A REST API

In this part, we will simply replace the fetch() method in our existing React application with Axios. All that remains is to install Axios and then use it in our App.js file to make the HTTP call to the GitHub API.

Let's now install Axios in our React project by doing one of the following:

Using NPM:

```
PS C:\Users\hp\Desktop\newproject> npm install axios
```

Using yarn:

```
PS C:\Users\hp\Desktop\newproject> yarn  axios
```

- After the installation is finished, we must import axios into our App.js. To the top of our App.js file, we'll add the following line:

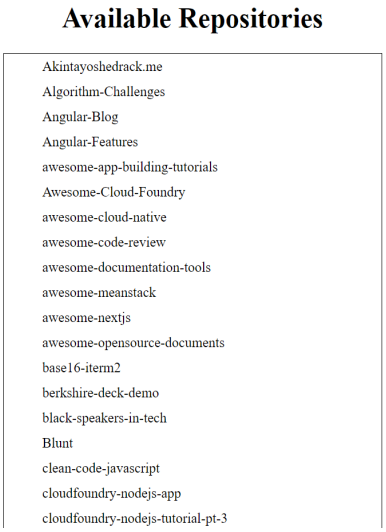
```
1  import axios from "axios";
```

- After adding the piece of code to our App.js, we only need to insert the following code within our useEffect():

```
useEffect(() => {
  const apiUrl = "https://api.github.com/users/hacktivist123/repos";
  axios.get(apiUrl).then((repos) => {
    const allRepos = repos.data;
    setData(allRepos);
  });
}, [setData]);
```

```jsx
import React from "react";
import { useEffect, useState } from "react";
import axios from "axios";
export default function ApiComponent() {
  const [data, setData] = useState([]);
  const myStyle = {
    listStyle: "none",
    marginRight: "auto",
    marginLeft: "auto",
    border: " 0.1px  solid black",
    width: 400,
    textAlign: "justify",
  };
  useEffect(() => {
    const apiUrl = "https://api.github.com/users/hacktivist123/repos";
    axios.get(apiUrl).then((repos) => {
      const allRepos = repos.data;
      setData(allRepos);
    });
  }, [setData]);
  return (
    <div style={{ textAlign: "center" }}>
      <h1>Available Repositories</h1>
      <ul style={myStyle}>
        {data.map((d) => (
          <li key={d.id} style={{ padding: 5 }}>
            {d.name}
          </li>
        ))}
      </ul>
    </div>
  );
}
```

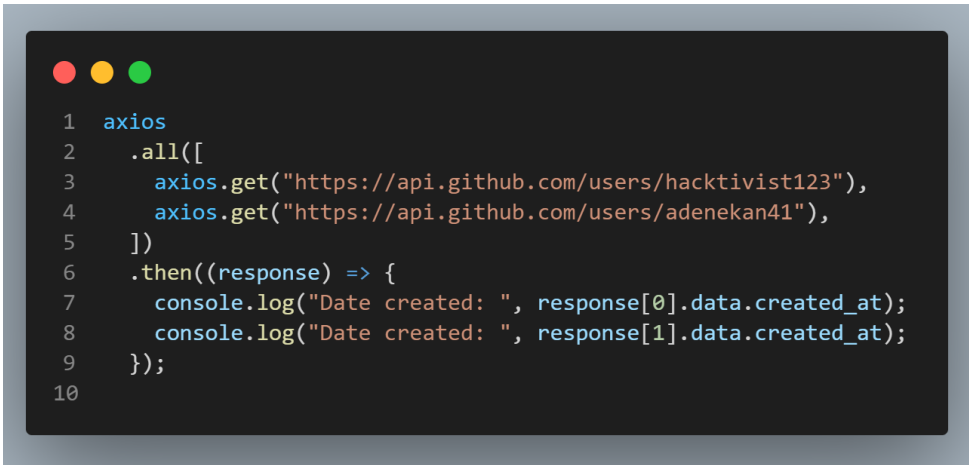
You may have noticed that the retrieve API has been replaced by the Axios shortcut method axios. To make a get request to the API, use get.
If we performed everything right, our app should still seem the same:



3. **Difference between Axios and fetch:**
    - **Syntax Fundamental concepts:** Both Fetch and Axios have relatively basic request syntaxes. But Axios has an advantage because it automatically transforms a response to JSON, therefore when we use Axios, we bypass the step of converting the response to JSON, whereas Fetch() requires us to do so. Finally, Axios shorthand methods allow us to simplify certain HTTP Requests.
    - **Compatibility with Web Browsers:** One of the numerous reasons why developers prefer Axios to Fetch is that Axios is supported by all major browsers and versions, whereas Fetch is only supported by Chrome 42+, Firefox 39+, Edge 14+, and Safari 10.1+.
    - **Response Timeout Management:** Setting a timeout for responses is simple with Axios by using the timeout option within the request object. However, this is not so simple in Fetch. Fetch has a comparable capability that uses the AbortController() interface, but it takes longer to implement and can be confusing.
    - **HTTP Request Interception:** Axios gives developers the ability to intercept HTTP requests. When we need to alter HTTP requests from our application to the server, we need HTTP interceptors. Interceptors enable us to do so without writing additional code.

- **Making Several Requests Simultaneously:**Axios allows us to send numerous HTTP requests using the axios all() methods. With the promise, fetch() gives the same functionality. We can perform numerous fetch() queries within the all() method.

  For example, we can use the axios.all() method to make several queries to the GitHub API, as shown below:



```
axios
  .all([
    axios.get("https://api.github.com/users/hacktivist123"),
    axios.get("https://api.github.com/users/adenekan41"),
  ])
  .then((response) => {
    console.log("Date created: ", response[0].data.created_at);
    console.log("Date created: ", response[1].data.created_at);
  });

```

  The code above executes parallel queries to an array of arguments and returns the response data; in our example, it will report the created at object from each API response to the console.