

Fullstack Web Development

Network Call & Form Validation

Outline

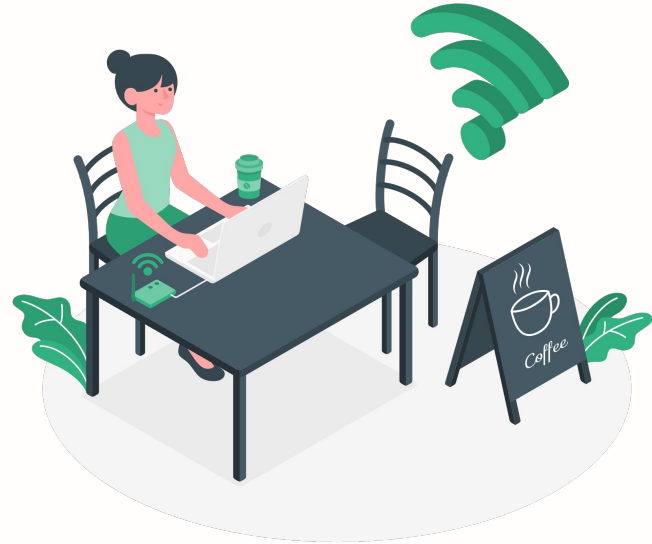
- Network call using Fetch and Axios
- Form submission & validation
- Create local server using json-server package

Network Call using Fetch and Axios

Network call can be done by two ways using:

1. Fetch
2. Axios

Let's see how they both work!



Fetch API

Fetch api is a built in promise-based api. Let's look at an example, here we took [json placeholder](#) API which is generally used for testing.

The beside code explains the basic syntax of fetching data from an api

```
// fetches api data from json placeholder
fetch("https://jsonplaceholder.typicode.com/todos/1")
  // gets response in json format
  .then((response) => response.json())
  // prints the response in console
  .then((json) => console.log(json))
  // catches error if any network error occurs
  .catch((err) => console.log(err));
```

Axios is a promise-based HTTP Client for node.js and the browser. It is isomorphic (it can run in the browser and nodejs with the same codebase).

Unlike Fetch api, axios is not a build-in api. So, we need to install it.

```
npm install axios
```

Then, import the axios in your file where you are going to fetch data.

```
axios.get("https://jsonplaceholder.typicode.com/posts")
  // gets response in json format
  .then((response) =>
    this.setState({
      post: response,
      isLoading: false,
    })
  )
  // catches error if any network error occurs
  .catch((err) =>
    this.setState({
      error: err,
      isLoading: false,
    })
  );
```

Axios or Fetch?

Axios provides an easy-to-use API in a compact package for most of your HTTP communication needs. However, if you prefer to stick with native APIs, nothing stops you from implementing Axios features.

It's perfectly possible to reproduce the key features of the Axios library using the `fetch()` method provided by web browsers. Ultimately, whether it's worth loading a client HTTP API depends on whether you're comfortable working with built-in APIs.



Basic HTTP Methods

HTTP defines a set of **request methods** to indicate the desired action to be performed for a given resource. Although they can also be nouns, these request methods are sometimes referred to as *HTTP verbs*.



-
- **GET**, The GET method requests a representation of the specified resource. Requests using GET should only retrieve data.
 - **POST**, The POST method submits an entity to the specified resource, often causing a change in state or side effects on the server.
 - **PUT**, The PUT method replaces all current representations of the target resource with the request payload.
 - **PATCH**, The PATCH method applies partial modifications to a resource.
 - **DELETE**, The DELETE method deletes the specified resource.

HTTP Method	CRUD operation	Entire Collection (e.g. /users)	Specific Item (e.g. /users/{id})
GET	Read	200 (OK), list of entities. Use pagination, sorting and filtering to navigate big lists.	200 (OK), single entity. 404 (Not Found), if ID not found or invalid.
POST	Create	201 (Created), Response contains response similar to GET /user/{id} containing new ID.	not applicable
PATCH	Update	Batch API	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
DELETE	Delete	204 (No Content). 400(Bad Request) if no filter is specified.	204 (No Content). 404 (Not Found), if ID not found or invalid.
PUT	Update/Replace	not implemented	not implemented

Form Submission and Validation

Now we will make a simple form submission with formik and yup as a validator.


First install formik and yup into your react project

```
npm i formik yup
```



Form Submission and Validation

First, import all things that we need to make a form and validation.



```
import React from "react";  
import { Formik, Form, Field, ErrorMessage } from "formik";  
import * as Yup from "yup"
```

- Formik needed as a wrap for our form
- Form needed as a wrap for input field
- Field needed to make an input for user
- ErrorMessage needed to show error message from validation
- Yup needed to make a validation schema

Form Submission and Validation

Make our validation schema using Yup. What we are doing in code below is making validation for email and password with some rules. For email, must be in email format and required (can't be empty). For password, must be 3 characters at minimum and required (can't be empty). For more information, you can go directly to the documentation from yup <https://www.npmjs.com/package/yup>

```
const LoginSchema = Yup.object().shape({
  email: Yup.string()
    .email("Invalid email address format")
    .required("Email is required"),
  password: Yup.string()
    .min(3, "Password must be 3 characters at minimum")
    .required("Password is required"),
});
```

Form Submission and Validation

Now make our component like code beside. Wrap our form inside `<Formik>`, with some props for Formik, like `initialValues`, `validationSchema`, and `onSubmit`.

```
class App extends React.Component {
  render() {
    return (
      <div>
        <Formik
          initialValues={{ email: "", password: "" }}
          validationSchema={LoginSchema}
          onSubmit={(values) => {
            console.log(values);
          }}
        >
          {(props) => {
            console.log(props)
            return <Form></Form>;
          }}
        </Formik>
      </div>
    );
  }
}

export default App;
```

-
- `initialValues`, is for making our field have default value, in our case we make our field, email and password, have default value an empty string.
 - `validationSchema` is a schema for validation.
 - `onSubmit`, is an event handler for what we want to do when we submit our form. In our case for make it simple, we just want to `console.log()` the value from email and password field.

Form Submission and Validation

```
class App extends React.Component {
  render() {
    return (
      <div>
        <Formik
          initialValues={{ email: "", password: "" }}
          validationSchema={LoginSchema}
          onSubmit={values => {
            console.log(values);
          }}
        >
          {(props) => {
            console.log(props)
            return <Form></Form>;
          }}
        </Formik>
      </div>
    );
  }
}

export default App;
```

Take a look at red square.

So inside `<Formik>` we will fill with a function that will return our form, This function has a params (we name it props) which contains an object that has many properties that we can use. You can `console.log(props)` for further information about it.

Form Submission and Validation

This is the content of the function, where we will fill in our form component. The picture on the left is Field for email, and on the right side is Field for password.

```
<Form>
  <div>
    <label htmlFor="email">Email</label>
    <Field
      type="text"
      name="email"
      placeholder="Enter email"
      autoComplete="off"
    />
    <ErrorMessage
      component="div"
      name="email"
      style={{ color: "red" }}
    />
  </div>
```

```
<div>
  <label htmlFor="password">Password</label>
  <Field
    type="password"
    name="password"
    placeholder="Enter password"
  />
  <ErrorMessage
    component="div"
    name="password"
    style={{ color: "red" }}
  />
</div>
<button type="submit">Login</button>
```


Form Submission and Validation

And that is our simple form submission and validation using formik dan yup. Just run our application by typing npm start in terminal.

When we fill in the email or password form with a format that does not match the schema we created earlier, an error message will appear.

Email

Invalid email address format

Password

Password must be 3 characters at minimum

Login

The json-server is a JavaScript library to create testing REST API.

For installation you can install json-server globally.

```
npm install -g json-server
```

Now we can create server locally with json-server or the other name is fake API because we don't have to create API. To create API, we can learn that in module Back-end, but now we can combine axios with json-server to practice network call

```
{
  "users": [
    {
      "id": 1,
      "first_name": "Robert",
      "last_name": "Schwartz",
      "email": "rob23@gmail.com"
    },
    {
      "id": 2,
      "first_name": "Lucy",
      "last_name": "Ballmer",
      "email": "lucy56@gmail.com"
    },
    {
      "id": 2,
      "first_name": "Anna",
      "last_name": "Smith",
      "email": "annasmith23@gmail.com"
    }
  ]
}
```

Now you have to create a file with ".json" extension, for example you can create a file with the name users.json which contains data like in the side

```
{
  "users": [
    {
      "id": 1,
      "first_name": "Robert",
      "last_name": "Schwartz",
      "email": "rob23@gmail.com"
    },
    {
      "id": 2,
      "first_name": "Lucy",
      "last_name": "Ballmer",
      "email": "lucy56@gmail.com"
    },
    {
      "id": 2,
      "first_name": "Anna",
      "last_name": "Smith",
      "email": "annasmith23@gmail.com"
    }
  ]
}
```

To run our json-server with its data, we have to access its directory in terminal, then we can use this command:

```
json-server --watch jsonFileName => json-server --watch users.json
```

This command will run our server in port default which is 3000. But if you want to run it on different port, you can use -p like this

```
json-server -pPortYou want jsonFileName => json-server -p2000 users.json
```

JSON Server GET request

GET request is used when we want to get data from server




```
const axios = require('axios');

axios.get('http://localhost:3000/users')
  .then(resp => {
    console.log(resp.data)
  })
  .catch(error => {
    console.log(error);
  });
```

JSON Server GET request

Every single data in json-server has an unique id. When we want to retrieve data with specific id, we can get by id like this:



```
const axios = require('axios');

axios.get('http://localhost:3000/users/2')
  .then(resp => {
    console.log(resp.data)
  })
  .catch(error => {
    console.log(error);
  });
```

JSON Server POST request

With a POST request, we create a new user.

```
const axios = require('axios');

axios.post('http://localhost:3000/users', {
  id: 6,
  first_name: 'Fred',
  last_name: 'Blair',
  email: 'freddyb34@gmail.com'
}).then(resp => {
  console.log(resp.data);
}).catch(error => {
  console.log(error);
});
```

JSON Server PUT request


In the following example we modify data with a PUT request to modify the user's email address with id 6.

```
const axios = require('axios');

axios.put('http://localhost:3000/users/6/', {
  first_name: 'Fred',
  last_name: 'Blair',
  email: 'freddyb34@yahoo.com'
}).then(resp => {
  console.log(resp.data);
}).catch(error => {
  console.log(error);
});
```


JSON Server PUT request

In the following example, we show how to delete a user with id 1 with a DELETE request.



```
const axios = require('axios');

axios.delete('http://localhost:3000/users/1/')
  .then(resp => {
    console.log(resp.data)
  }).catch(error => {
    console.log(error);
  });
```

Other Operation Json-server

That's actually some basic and most used feature in json-server, but there are few other features that you can use like sorting, operators, full text search etc. For further explanation you can read it at it's documentation or at

<https://zetcode.com/javascript/jsonserver/>



Thank You!

