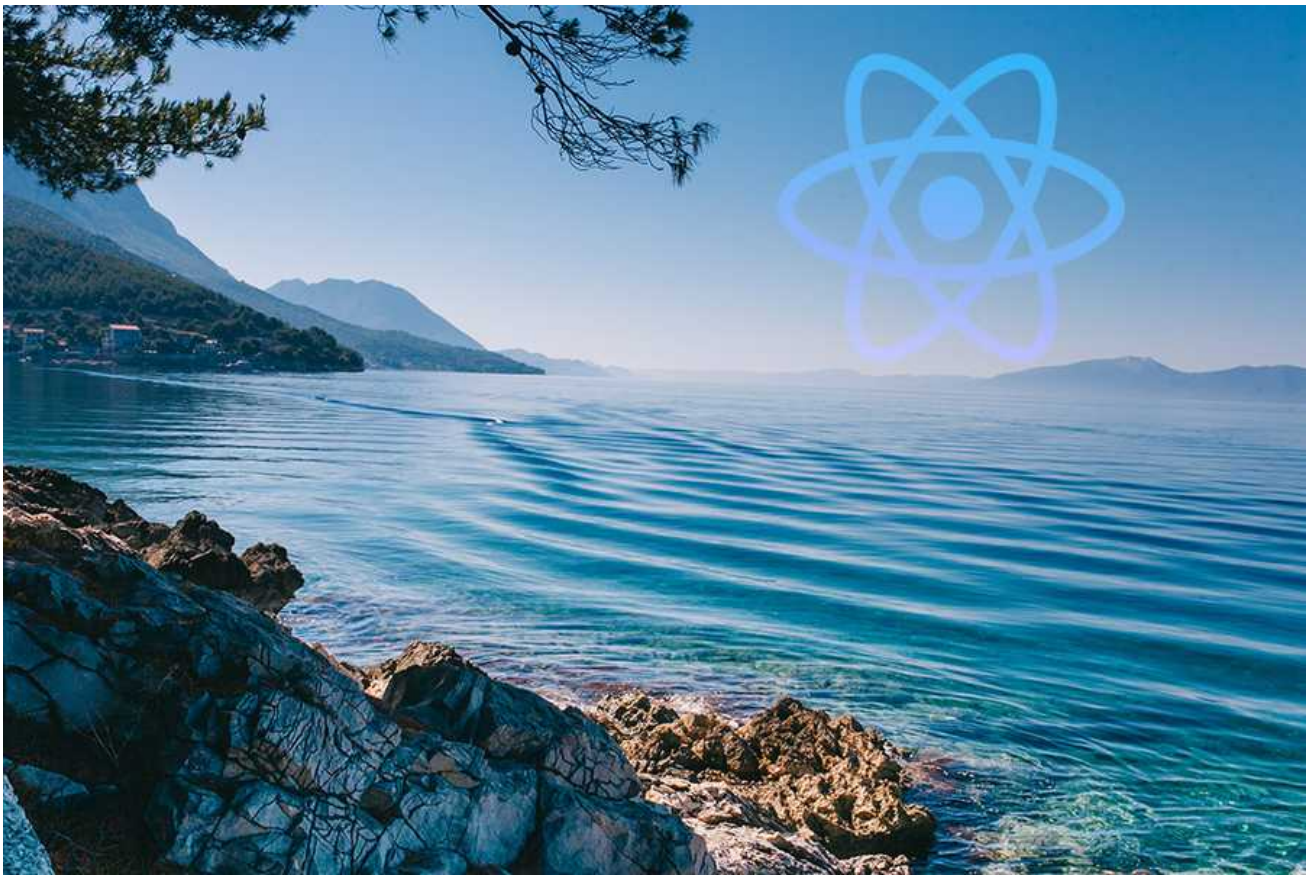
How to mock data in React with a fake API

SEPTEMBER 18, 2020 BY ROBIN WIERUCH - EDIT THIS POST



This tutorial is part 2 of 2 in this series.

Part 1: JavaScript Fake API with Mock Data

In this tutorial we will implement use JavaScript fake API with mock data from a pseudo backend to create our frontend application with React. Often this helps whenever there is no backend yet and you need to implement your React frontend against some kind of realistic data. You can find the finished project on GitHub.

REACT WITH MOCK DATA FROM A FAKE API

In a previous tutorial, we implemented the fake API with JavaScript's Promises for having it asynchronous and JavaScript's `setTimeout` function for having an artificial delay. Now we want to use this fake API with its mock data as replacement for a backend in our React application. We will start with a clean slate of a React application:

```
import React from 'react';

import { getUsers, createUser, updateUser, deleteUser } from './api';

const App = () => {
  return <div>Hello React</div>;
};

export default App;
```

For our App component, we import all the functions from our previously implemented fake API. The `getUser` function isn't needed, hence we don't have to import it.

Next, we are going to fetch the mock data from the fake API. Therefore we are using React's useEffect Hook and store the mock data with React's useState Hook in the component's state:

```
import React from 'react';

import { getUsers, createUser, updateUser, deleteUser } from './api';

const App = () => {
  const [users, setUsers] = React.useState(null);

  React.useEffect(() => {
    const doGetUsers = async () => {
      const result = await getUsers();
      setUsers(result);
    };

    doGetUsers();
  }, []);

  return <div>Hello React</div>;
};

export default App;
```

If there is no mock data when the component renders for the first time, we don't render anything by returning null (see conditional rendering). If the data arrives eventually, we render it as a list:

```
const getDeveloperText = (isDeveloper) =>
  `is ${isDeveloper ? 'a' : 'not a'} developer`;

const App = () => {
  ...

  if (!users) {
    return null;
  }

  return (
    <div>
      <ul>
        {users.map((user) => {
          const developerText = getDeveloperText(user.isDeveloper);

          return (
            <li key={user.id}>
              {user.firstName} {user.lastName} {isDeveloperText}
            </li>
          );
        })}
      </ul>
    </div>
  );
};
```

Optionally we could also add error handling for the fetching of the mock data. However, in this tutorial we won't implement any real error handling (or loading indicators). Please check my how to fetch data with React tutorial if you are interested in this topic.

```
const App = () => {
  const [users, setUsers] = React.useState(null);

  React.useEffect(() => {
    const doGetUsers = async () => {
      try {
        const result = await getUsers();
        setUsers(result);
      } catch (error) {
        console.log(error);
      }
    };

    doGetUsers();
  }, []);

  ...
};
```

That's it for fetching mock data and rendering it. You could replace this API function always with a function which calls a real backend's API.

REACT FOR CREATING MOCK DATA WITH A FAKE API

Let's continue with creating more mock data. First, we will need a form to input all the information for the new data:

```
const App = () => {
  ...

  return (
    <div>
      <ul>
        ...
      </ul>

      <hr />

      <span>Create User:</span>

      <form onSubmit={handleCreate}>
        <label>
          First Name:
          <input type="input" onChange={handleChangeFirstName} />
        </label>

        <label>
          Last Name:
          <input type="input" onChange={handleChangeLastName} />
        </label>

        <button type="submit">Create</button>
      </form>
    </div>
  );
};
```

Second, the implementation details for the two event handlers which update the form state:

```
const App = () => {
  const [users, setUsers] = React.useState(null);

  React.useEffect(() => {
    ...
```

```
  }, []);

  const [firstName, setFirstName] = React.useState('');
  const [lastName, setLastName] = React.useState('');

  const handleChangeFirstName = (event) => {
    setFirstName(event.target.value);
  };

  const handleChangeLastName = (event) => {
    setLastName(event.target.value);
  };

  if (!users) {
    return null;
  }

  return (
    ...
  );
};
```

Third, the handler for the actual creation when the form gets submitted; which prevents the default to avoid a browser refresh. Again, there could be handling for error state and loading state too, but we cover this in another tutorial.

```
const App = () => {
  ...

  const handleCreate = async (event) => {
    event.preventDefault();

    try {
      await createUser({ firstName, lastName, isDeveloper: false });
    } catch (error) {
      console.log(error);
    }
  };

  ...
};
```

The actual creation of the additional mock data should work, however, you will not see the result reflected in the React UI. That's because we are not updating the mock data in the UI. There are two ways to keep the UI in sync after a request which modifies data on the backend:

- After the request finishes, we know about the mock data which we just created, so we could update the React's state with it (.e.g. updating the `users` state with the new user).

- After the request finishes, we could refetch all the mock data from the backend. That's another network roundtrip to the backend (which is our fake API here), but it keeps our data in sync with the rendered UI as well.

We will do the latter way to keep the mocked data in sync. However, feel free to follow the other way. In order to implement the refetching, we need to extract the actual logic which is used to fetch the mock data in the first place:

```
const App = () => {
  const [users, setUsers] = React.useState(null);

  const doGetUsers = React.useCallback(async () => {
    try {
      const result = await getUsers();
      setUsers(result);
    } catch (error) {
      console.log(error);
    }
  }, []);

  React.useEffect(() => {
    doGetUsers();
  }, [doGetUsers]);

  ...
};
```

We have to use React's useCallback Hook here, because it memoizes the function for us which mean that it doesn't change and therefore React's useEffect Hook isn't called in an endless loop. Next, we can reuse this extracted function to refetch the mocked data after creating new data:

```
const App = () => {
  const [users, setUsers] = React.useState(null);

  const doGetUsers = React.useCallback(async () => {
    try {
      const result = await getUsers();
      setUsers(result);
    } catch (error) {
      console.log(error);
    }
  }, []);

  React.useEffect(() => {
    doGetUsers();
  }, [doGetUsers]);

  const refetchUsers = async () => {
```

```
    await doGetUsers();
  };

  ...

  const handleCreate = async (event) => {
    event.preventDefault();

    try {
      await createUser({ firstName, lastName, isDeveloper: false });
      await refetchUsers();
    } catch (error) {
      console.log(error);
    }
  };

  ...
};
```

That's it. After creating new data for our fake backend, we refetch all the mock data and let the component re-render with it. Just refetching everything keeps the data from (pseudo) backend and frontend in sync.

UPDATING AND DELETING MOCK DATA IN REACT

Next we will implement the process of updating data in our pseudo database. First, we will introduce a button which will be used to flip the boolean for one property of our mock data:

```
const App = () => {
  ...

  return (
    <div>
      <ul>
        {users.map((user) => {
          const developerText = getDeveloperText(user.isDeveloper);

          return (
            <li key={user.id}>
              {user.firstName} {user.lastName} {developerText}
              <button
                type="button"
                onClick={() => handleEdit(user.id)}
              >
                Toggle Developer (Update)
              </button>
            </li>
```

```
          );
        })}
      </ul>

      <hr />

      ...
    </div>
  );
};
```

What's missing is the handler which updates the mock data via our fake API and which refetches all mock data afterward for keeping the data in sync:

```
const App = () => {
  ...

  const handleEdit = async (id) => {
    const user = users.find((user) => user.id === id);
    const isDeveloper = !user.isDeveloper;

    try {
      await updateUser(id, { isDeveloper });
      await refetchUsers();
    } catch (error) {
      console.log(error);
    }
  };

  ...
};
```

That's it for updating the mock data. We just flipped a boolean here, but you can imagine how you could use this with input fields as well to change the other properties of the mocked entity. Last but not least, we will implement a button to remove mock data and a handler which does the actual deletion of it:

```
const App = () => {
  ...

  const handleRemove = async (id) => {
    try {
      await deleteUser(id);
      await refetchUsers();
    } catch (error) {
      console.log(error);
    }
  };
```

```
  ...

  return (
    <div>
      <ul>
        {users.map((user) => {
          const developerText = getDeveloperText(user.isDeveloper);

          return (
            <li key={user.id}>
              {user.firstName} {user.lastName} {developerText}
              <button
                type="button"
                onClick={() => handleEdit(user.id)}
              >
                Toggle Developer (Update)
              </button>
              <button
                type="button"
                onClick={() => handleRemove(user.id)}
              >
                Remove User (Delete)
              </button>
            </li>
          );
        })}
      </ul>

      <hr />

      ...
    </div>
  );
};
```

That's it. Updating and deleting items in our mock data from our pseudo database which is accessible via our fake API is possible now.

. . .

After all, we use all four CRUD operations from our fake API which connects to a pseudo database in our React application. We could easily exchange the functions for the fake API with functions which hit real API endpoints now. The frontend wouldn't need any changes. With all this knowledge, frontend developers are empowered to create their own mock data API until the backend gives you a real API to work with.

Show Comments

KEEP READING ABOUT JAVASCRIPT›

HOW TO CREATE A REST API WITH OAK IN DENO

An Oak application is most often used as a backend application in a client-server architecture whereas the client could be written in React.js or another popular frontend solution and the server could…

JAVASCRIPT FAKE API WITH MOCK DATA

In this tutorial we will implement a JavaScript fake API. Often this helps whenever there is no backend yet and you need to implement your frontend against some kind of realistic data. Fake it till…

PORTFOLIO

Online Courses

Open Source

Tutorials

ABOUT

About me

What I use

How to work with me

How to support me

Contact Me Privacy & Terms