Fetching Data

You will get to learn how to fetch data from an API using one the lifecycle methods that we discussed in the previous lesson.

Now we're prepared to fetch data from the Hacker News API. There was one lifecycle method mentioned that can be used to fetch data: `componentDidMount()`. Before we use it, let's set up the URL constants and default parameters to break the URL endpoint for the API request into smaller pieces.

```
import React, { Component } from 'react';
import './App.css';

const DEFAULT_QUERY = 'redux';

const PATH_BASE = 'https://hn.algolia.com/api/v1';
const PATH_SEARCH = '/search';
const PARAM_SEARCH = 'query=';
```

In JavaScript ES6, you can use template literals for string concatenation or interpolation. You will use it to concatenate your URL for the API endpoint.

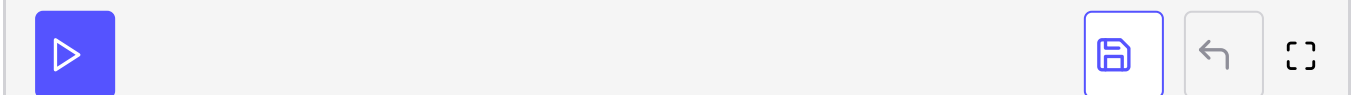
```
const DEFAULT_QUERY = 'redux';
const PATH_BASE = 'https://hn.algolia.com/api/v1';
const PATH_SEARCH = '/search';
const PARAM_SEARCH = 'query=';

// comment either ES5 or ES6

// ES6
const url1 = `${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${DEFAULT_QUERY}`;

// ES5
const url2 = PATH_BASE + PATH_SEARCH + '?' + PARAM_SEARCH + DEFAULT_QUERY;

console.log(url1);
// output: https://hn.algolia.com/api/v1/search?query=redux
// output
console.log(url2);
```



This will keep your URL composition flexible in the future. Below, the entire data fetch process will be presented, and each step will be explained afterward.

```
import React, { Component } from 'react'
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      result: null,
      searchTerm: DEFAULT_QUERY,
    };
    this.setSearchTopStories = this.setSearchTopStories.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  setSearchTopStories(result) {
    this.setState({ result });
  }

  componentDidMount() {
    const { searchTerm } = this.state;

    fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}`)
      .then(response => response.json())
      .then(result => this.setSearchTopStories(result))
      .catch(error => error);
  }
//  ...
}
```

First, we remove the sample list of items, because we will return a real list from the Hacker News API, so the sample data is no longer used. The initial state of your component has an empty result and default search term now. The same default search term is used in the input field of the Search component, and in your first request.

Second, you use the `componentDidMount()` lifecycle method to fetch the data after the component mounted. The first fetch uses default search term from the local state. It will fetch “redux” related stories, because that is the default parameter.

Third, the native fetch API is used. The JavaScript ES6 template strings allow it to compose the URL with the `searchTerm`. The URL is the argument for the native fetch API function. The response is transformed to a JSON data structure, a mandatory step in a native fetch with JSON data structures, after which it can be set as result in the local component state. If an error occurs during the request, the function will run into the catch block instead of the then block.

Last, remember to bind your new component method in the constructor.

Now you can use the fetched data instead of the sample list. Note that the result is not only a list of data, but a complex object with meta information and a list of hits that are news stories. You can output the lcoal state with `console.log(this.state);` in your `render()` method to visualize it.

In the next step, we use the result to render it. But we will prevent it from rendering anything, so we will return null, when there is no result in the first place. Once the request to the API has succeeded, the result is saved to the state and the App component will re-render with the updated state.

```
class App extends Component {

  ...

  render() {
    const { searchTerm, result } = this.state;

    if (!result) { return null; }

    return (
      <div className="page">
        ...
        <Table
          list={result.hits}
          pattern={searchTerm}
          onDismiss={this.onDismiss}
        />
      </div>
    );
  }
}
```

Now, let's recap what happens during the component lifecycle. Your component is initialized by the constructor, after which it renders for the first time. We prevented it from displaying anything, because the result in the local state is null. It is allowed to return null for a component to display nothing.

Then the `componentDidMount()` lifecycle method fetches the data from the Hacker News API asynchronously. Once the data arrives, it changes your local component state in `setSearchTopStories()`. The update lifecycle activates because the local state was updated. The component runs the `render()` method again, but this time with populated result in your local component state. The component and the Table component will be rendered with its content.

We used the native fetch API supported by most browsers to perform an asynchronous request to an API. The *create-react-app* configuration makes sure it is supported by all browsers. There are also third-party node packages that you can use to substitute the native fetch API: axios. You will use axios later in this course.

In this course, we build on JavaScript's shorthand notation for truthfulness checks. In the previous example, `if (!result)` was used in favor of `if (result === null)`. The same applies for other cases as well. For instance, `if (!list.length)` is used in favor of `if (list.length === 0)` or `if (someString)` is used in favor of `if (someString !== '')`.

The list of hits should now be visible in our application; however, two regression bugs have appeared. First, the "Dismiss" button is broken, because it doesn't know about the complex result object, but it still operates on the plain list from the sample data when dismissing an item. Second, when the list is displayed and you try to search for something else, it gets filtered on the client-side, though the initial search was made by searching for stories on the server-side. The perfect behavior would be to fetch another result object from the API when using the Search component. Both regression bugs will be fixed in the following chapters.

```
import React, { Component } from 'react';
require('./App.css');

const DEFAULT_QUERY = 'redux';

const PATH_BASE = 'https://hn.algolia.com/api/v1';
const PATH_SEARCH = '/search';
const PARAM_SEARCH = 'query=';

const isSearched = (searchTerm) => (item) =>
  item.title.toLowerCase().includes(searchTerm.toLowerCase());

class App extends Component {
```

```
constructor(props) {
  super(props);

  this.state = {
    result: null,
    searchTerm: DEFAULT_QUERY,
  };

  this.setSearchTopstories = this.setSearchTopstories.bind(this);
  this.fetchSearchTopstories = this.fetchSearchTopstories.bind(this);
  this.onSearchChange = this.onSearchChange.bind(this);
  this.onDismiss = this.onDismiss.bind(this);
}

setSearchTopstories(result) {
  this.setState({ result });
}

fetchSearchTopstories(searchTerm) {
  fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}`)
    .then(response => response.json())
    .then(result => this.setSearchTopstories(result))
    .catch(e => e);
}

componentDidMount() {
  const { searchTerm } = this.state;
  this.fetchSearchTopstories(searchTerm);
}

onSearchChange(event) {
  this.setState({ searchTerm: event.target.value });
}

onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedList = this.state.list.filter(isNotId);
  this.setState({ list: updatedList });
}

render() {
  const { searchTerm, result } = this.state;

  if (!result) { return null; }

  return (
    <div className="page">
      <div className="interactions">
        <Search
          value={searchTerm}
          onChange={this.onSearchChange}
        >
          Search
        </Search>
      </div>
      <Table
        list={result.hits}
        pattern={searchTerm}
        onDismiss={this.onDismiss}
      />
    </div>
  );
```

```
  }
}


const Search = ({ value, onChange, children }) =>
  <form>
    {children} <input
      type="text"
      value={value}
      onChange={onChange}
    />
  </form>

const Table = ({ list, pattern, onDismiss }) =>
  <div className="table">
    { list.filter(isSearched(pattern)).map(item =>
      <div key={item.objectID} className="table-row">
        <span style={{ width: '40%' }}>
          <a href={item.url}>{item.title}</a>
        </span>
        <span style={{ width: '30%' }}>
          {item.author}
        </span>
        <span style={{ width: '10%' }}>
          {item.num_comments}
        </span>
        <span style={{ width: '10%' }}>
          {item.points}
        </span>
        <span style={{ width: '10%' }}>
          <Button
            onClick={() => onDismiss(item.objectID)}
            className="button-inline"
          >
            Dismiss
          </Button>
        </span>
      </div>
    )}
  </div>

const Button = ({ onClick, className = '', children }) =>
  <button
    onClick={onClick}
    className={className}
    type="button"
  >
    {children}
  </button>

export default App;
```

Further Reading:

- Read about ES6 template literals
- Read about the native fetch API
- Read about data fetching in React