Using REST APIs

Course: Java

S1

Asynchronous HTTP in the browser changed everything.

Asynchronous HTTP

Before it was available, browsers fully reloaded their view each time a data interaction was required.

- Many UI events required a GET or POST that delivered a completely new HTML payload.
- The browser's JavaScript couldn't take advantage of its asynchronous nature.
- The HTML in a new view might just slightly differ from the original document.
- It was a waste of bandwidth.

Worse, it could be slow.

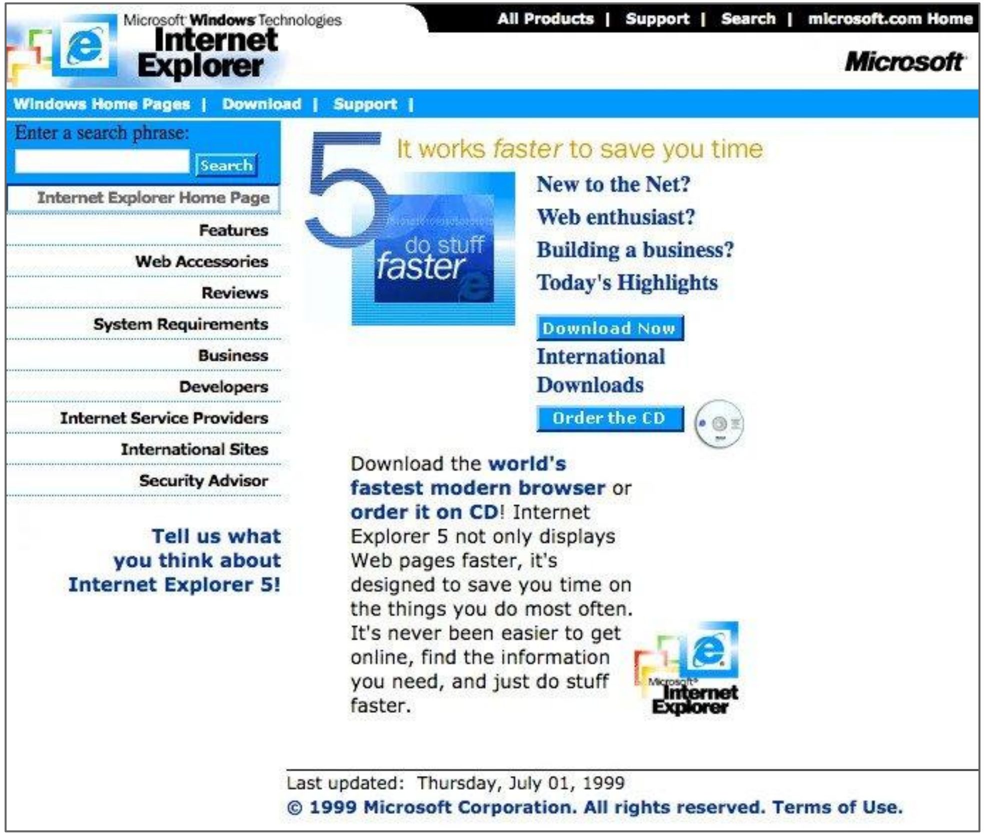
Depending on the network and HTTP server load, editing a single field could make the browser lock up and just...wait.

Asynchronous HTTP (continued)

Microsoft introduced asynchronous HTTP in 1999 with XMLHttpRequest.

The industry recognized a good thing.

But, it took another full decade before asynchronous responses were common and reliable.



Microsoft Windows Technologies Internet Explorer
All Products | Support | Search | microsoft.com Home
Microsoft
Windows Home Pages | Download | Support |
Enter a search phrase:
Search
Internet Explorer Home Page
Features
Web Accessories
Reviews
System Requirements
Business
Developers
Internet Service Providers
International Sites
Security Advisor
Tell us what you think about Internet Explorer 5!
5
do stuff faster
It works faster to save you time
New to the Net?
Web enthusiast?
Building a business?
Today's Highlights
Download Now
International Downloads
Order the CD

Download the world's **fastest modern browser** or **order it on CD**! Internet Explorer 5 not only displays Web pages faster, it's designed to save you time on the things you do most often. It's never been easier to get online, find the information you need, and just do stuff faster.

Last updated: Thursday, July 01, 1999

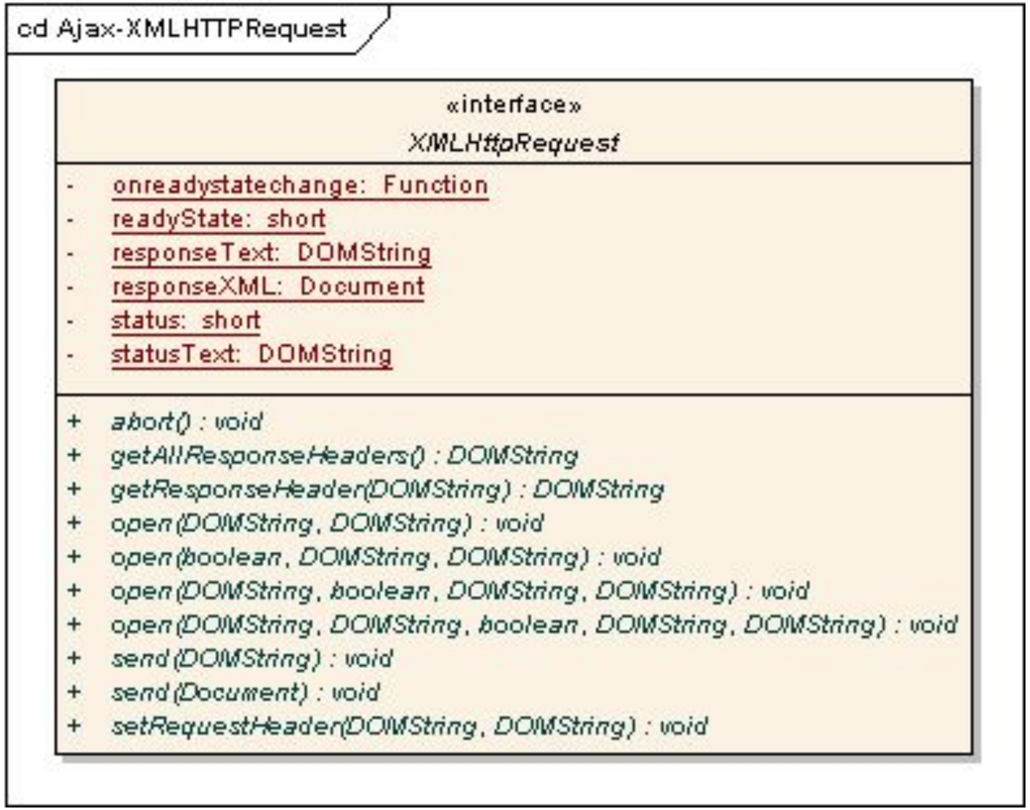
© 1999 Microsoft Corporation. All rights reserved. Terms of Use.

Asynchronous HTTP (continued)

Microsoft introduced asynchronous HTTP in 1999 with XMLHttpRequest.

The industry recognized a good thing.

But, it took another full decade before asynchronous responses were common and reliable.



```
cd Ajax-XMLHTTPRequest

«interface»
XMLHttpRequest

-  onreadystatechange:  Function
-  readyState:  short
-  responseText:  DOMString
-  responseXML:  Document
-  status:  short
-  statusText:  DOMString

+  abort() : void
+  getAllResponseHeaders() : DOMString
+  getResponseHeader(DOMString) : DOMString
+  open(DOMString, DOMString) : void
+  open(boolean, DOMString, DOMString) : void
+  open(DOMString, boolean, DOMString, DOMString) : void
+  open(DOMString, DOMString, boolean, DOMString, DOMString) : void
+  send(DOMString) : void
+  send(Document) : void
+  setRequestHeader(DOMString, DOMString) : void
```

Asynchronous HTTP (continued)

React takes full advantage of JavaScript’s asynchronous features.

- It’s possible to build an entire application with a single synchronous request (and many asynchronous requests).
- Async responses require us to think differently.
- We can’t rely on data if we don't know when it will arrive.
- React’s `useEffect` hook is one possible tool for async problem solving.

Learning Outcomes

By the end of this lesson, you will be able to:

01 Use `useEffect` to fetch initial state from a back-end service.

02 Send GET, POST, PUT, and DELETE `fetch` requests.

03 On `fetch` success, update state to the appropriate values.

04 Handle `fetch` failures.

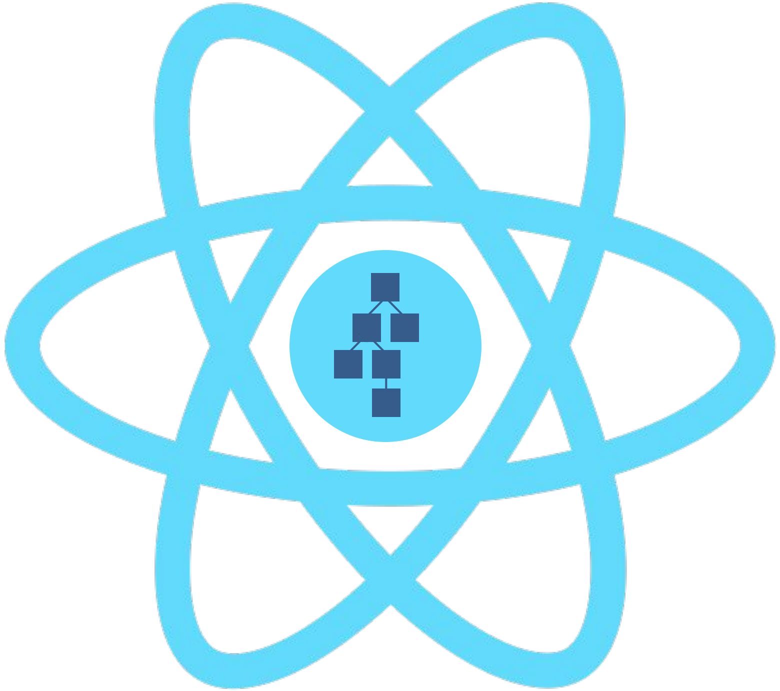
Fetch

React and DOM Updates

Before we talk about asynchronous HTTP, we need to understand how React thinks about the DOM.

React prefers that:

- We never use the DOM API directly.
- We build JSX expressions based on state.
- State updates are predictable.



Side Effects

It's possible this code will work, but it's full of side effects.

A **side effect** is something outside of the React pipeline.

In the first case, we're manipulating the DOM directly.

In the second, we're populating state from scratch every time the component renders.

`fetch` will run again and again any time the parent's props change or state changes. The code might work, but it's an infinite loop of data fetches.

```
function DomModifier() {

    const [state, setState] = useState([]);

    // Not great: direct DOM manipulation
    const div = document.createElement("div");
    div.textContent = `New div created: ${new Date()}.`
    document.querySelector("#target").appendChild(div);

    // Not great: Runs every time the component
    // renders, not just once.
    fetch(url)
        .then(response => response.json())
        .then(result => setState(result));

    return (
        <>
            <h1>I modify the DOM.</h1>
            <div>State: {state}</div>
            <div id="target"></div>
        </>
    );
}
```

useEffect

The `useEffect` hook manages side effects.

It lets React know to expect a side effect.

It follows the hook rules.

Warning! This is only an example. We still should never modify the DOM directly.

```
function DomModifier() {

    const [clicks, setClicks] = useState(0);

    useEffect(() => {
        const div = document.createElement("div");
        div.textContent = `New div created: ${new Date()}.`
        document.body.appendChild(div);
    }, [clicks]);

    return (
        <>
            <h1>I modify the DOM.</h1>
            <button onClick={() => setClicks(clicks + 1)}>
                Clicks: {clicks}
            </button>
        </>
    );
}
```

useEffect Arguments

Our use of the `useEffect` hook is pretty limited. We only ever initialize state with `fetch`. The second argument is always `[ ]`.

The first argument is a callback function that may cause a side effect.

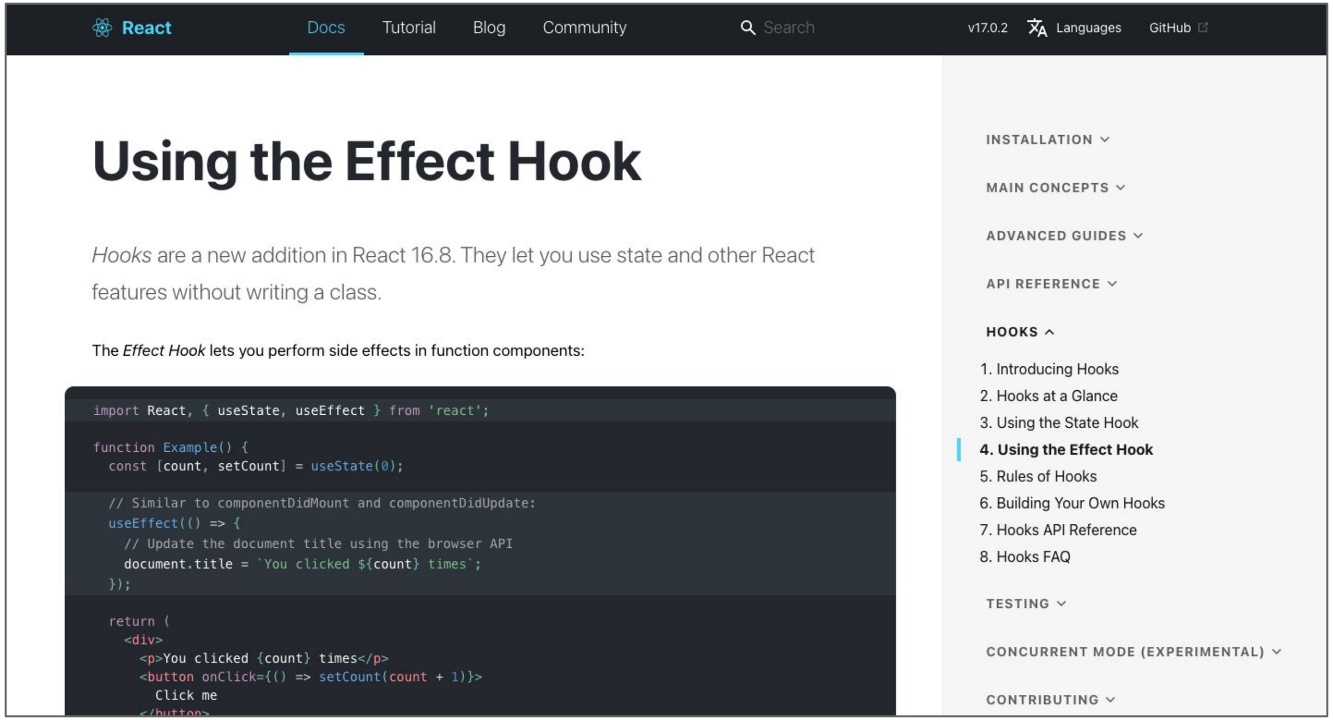
The second is an array of state variables.

- A change of a variable in the array executes the callback.
- Omitting the array executes the callback on every render.
- An empty array only executes the callback once.

```
// first: a callback
// second: array of state whose
// change triggers the callback
useEffect(() => {
    // do work
}, [state1, state2, stateN]);

// executes once
useEffect(() => {
}, []);

// executes on every render
useEffect(() => {
});
```

useEffect Arguments (continued)

Let's review the `document.title` example from React's useEffect docs.



Using the Effect Hook

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.

The *Effect Hook* lets you perform side effects in function components:

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
```

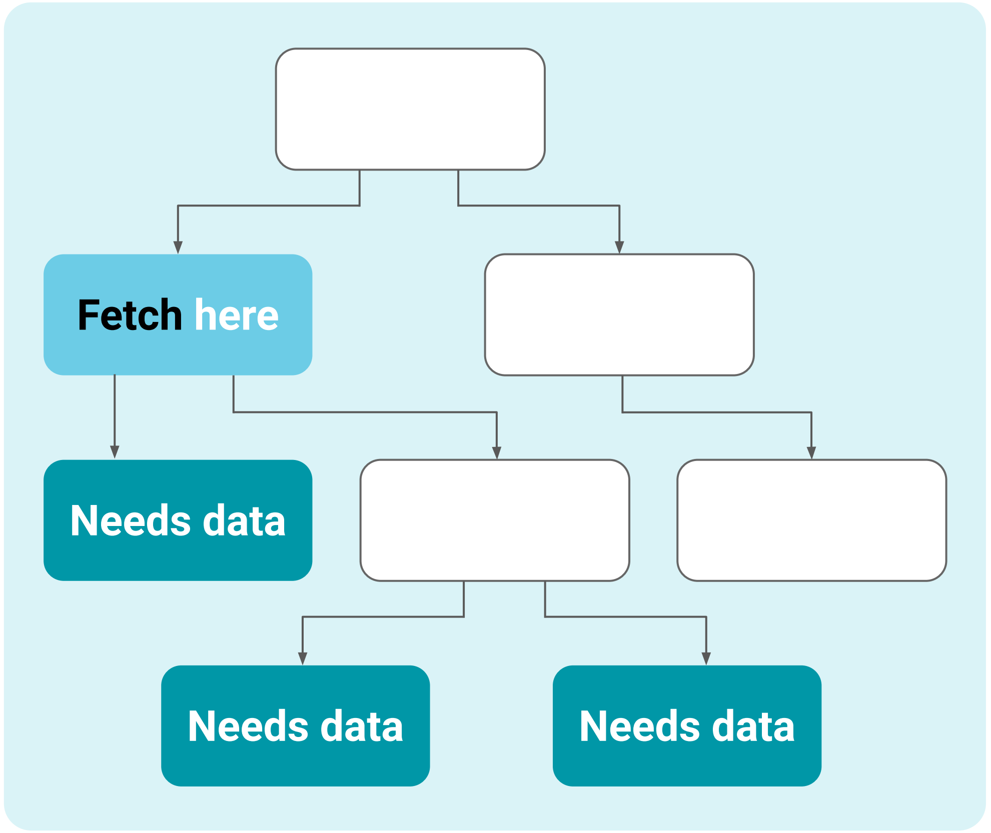
Fetching Initial State

- Use the `useEffect` hook with an empty state array to `fetch` data once and only once.
- It's also possible to force additional data `fetch`es by adding a state trigger, such as `(useEffect(() => {}, [forceRefresh]);`.
- Never omit the second argument, or your `fetch` will happen again and again.

```
const [todos, setTodos] = useState([]);
const [waiting, setWaiting] = useState(true);

// executes only once
useEffect(() => {
    fetch("http://localhost:8080/api/todo")
        .then(response => response.json())
        .then(result => {
            setTodos(result);
            setWaiting(false);
        })
        .catch(console.log);
}, []);
```

Where to Fetch Initial Data

- Components have a hierarchy.
- We want to share data with all components that need it instead of tracking independent state.
- Fetch initial data in the first common ancestor.



Create, Update, and Delete

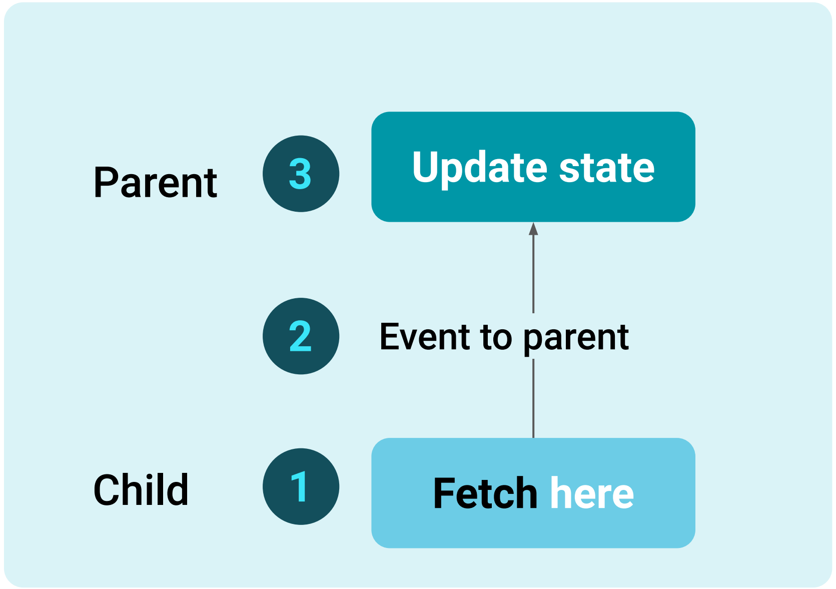
- Other CRUD operations should be protected with a function.
- `fetch` only fails on a network failure, so it's important to check the expected status.
- On failure or bad status, acknowledge this in the UI.
- On success, update state.

```
function add(todo) {

    const init = {
        method: "POST",
        headers: {
            "Content-Type": "application/json",
            "Accept": "application/json"
        },
        body: JSON.stringify(todo)
    };

    fetch("http://localhost:8080/api/todo", init)
        .then(response => {
            if (response.status === 201) {
                return response.json();
            }
            // could check other statuses...
            return Promise.reject("ToDo was not created.");
        })
        .then(result => setTodos([...todos, result]))
        .catch(console.log)
        .finally(() => setWaiting(false));
}
```

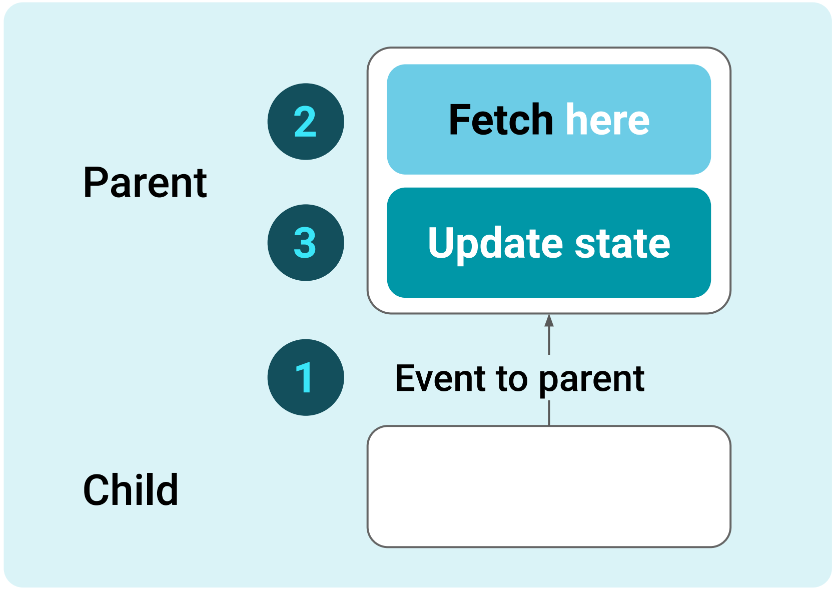
Where to Create, Update, or Delete

Two options exist, but most teams prefer one or the other.

POST, PUT, DELETE in the child; execute the parent's callback; the parent updates state.



Execute the parent's callback; POST, PUT, DELETE in the parent; the parent updates state.



Break

Fetch Activities

Time to Code

ToDo with a REST API Walkthrough

Suggested Time:

30 Minutes

Questions?

Activity: Paranormal Investigator

Suggested Time:

60 Minutes

Time’s Up! Let’s Review.

Questions?

RECAP

Recap

Recap questions:

- What is a side effect in React? (What is a side effect in other programming?)
- What problem does the `useEffect` function solve?
- What are `useEffect`'s arguments?
- What happens if we don't put our initial `fetch` inside of `useEffect`?
- Where is the correct place to put `fetch` GET, POST, PUT, and DELETE operations?
- When does the `fetch` promise resolve in relation to the current component's render?

Learning Outcomes

By the end of this lesson, you will be able to:

01 Use `useEffect` to fetch initial state from a back-end service.

02 Send GET, POST, PUT, and DELETE `fetch` requests.

03 On `fetch` success, update state to the appropriate values.

04 Handle `fetch` failures.

The End