3rd Party API Request

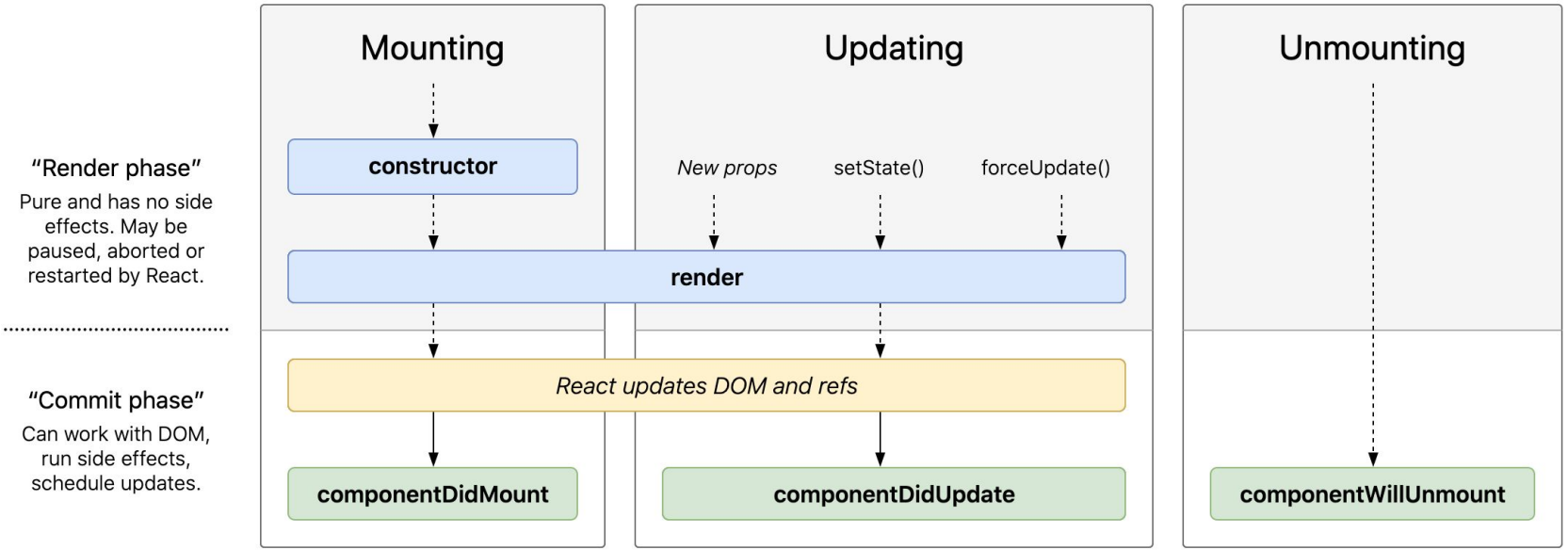
Hunter Jorgensen

React Component Lifecycle

While the concepts apply to frontend and backend, we're going to step back into React land to discuss some lifecycle concepts. As you think about your code, something important to understand is that React Components have a lifecycle.

React Components are created, inserted into the DOM (i.e., added to the page), updated with new data and eventually destroyed. The React Component class provides optional functions to implement that allow you to add logic based on certain states of your component. Should you update a component via props, or load up user data only after a component has successfully been inserted into the page?

React Component Lifecycle



React Component Lifecycle Methods

- constructor - this is like the constructor of a class in any programming language. This is the logic done before the object is instantiated
- render - this returns the view of the component. You should be familiar with this by now!
- componentDidMount - this optional method is called AFTER the component is inserted into the dom, and after rendering for the first time
- componentDidUpdate - this optional method is called AFTER the component updates (i.e., re-renders) as when the props or state is updated
- componentWillUnmount - when a component is being REMOVED from the DOM

Axios vs Fetch

We have a builtin API in JavaScript (part of the API from the browser, so not available in Node.js) known as fetch, but it can be a bit confusing to work with and it doesn’t work well with older browsers.

You are welcome to use fetch, but I recommend Axios, a wrapper over fetch that makes the code much easier and read.

```
npm install -s axios
```

Revisiting Asynchronous Methods

When performing some logic in JavaScript that might not happen immediately (such as making an API call), JavaScript we usually have to work with one of JavaScript's asynchronous tools. This includes:

- Callbacks
- Promises
- Async/Await

For today, we'll focus on promises (we'll explore async/await next session.)

What are Promises again?

Let’s say you have to make a network call in a programming language like Java. Typically, your code might look something like this:

```
some fancy Java logic...
Response response = amazingServiceImplementation.getImportantDataFromServer();
return response;
```

In this example, your code would generally pause until `getImportantDataFromServer()` resolves (even if that takes a few seconds), and then the code would resume.

Because a standard web page makes dozens of networks calls, pausing code execution on each call would DRASTICALLY increase page loading time. So, JavaScript does something different, where it will continue execution of the code, and then return to the waiting code when it finished resolving.

What are Promises again?

Looking at the last example, if the code execution were to continue before `getImportantDataFromServer()` finishes executing, then the response returned in the logic would be `undefined`, which could have bad repercussions on the code!

To avoid this, promise logic is ‘chained’ together (similar to streams in Java) using either .then or .catch methods.

Demo

Promises Can Be Chained!

You can use successive `.then(...)` calls to make the code easier; but your .catch(...), if you're using it, must appear at the end:

Demo

If you want to do something with the data (like setState) it must be done in a then block!

Demo

Hookify It

If we want to do this with React Hooks, then we can use the **useEffect** hook to ensure that this API is only called once on page load:

Demo

As you may recall, the second argument of useEffect is a list of values that, if changed, will call the function in the first argument. With an empty array, the method is only called once.