Code.Hub

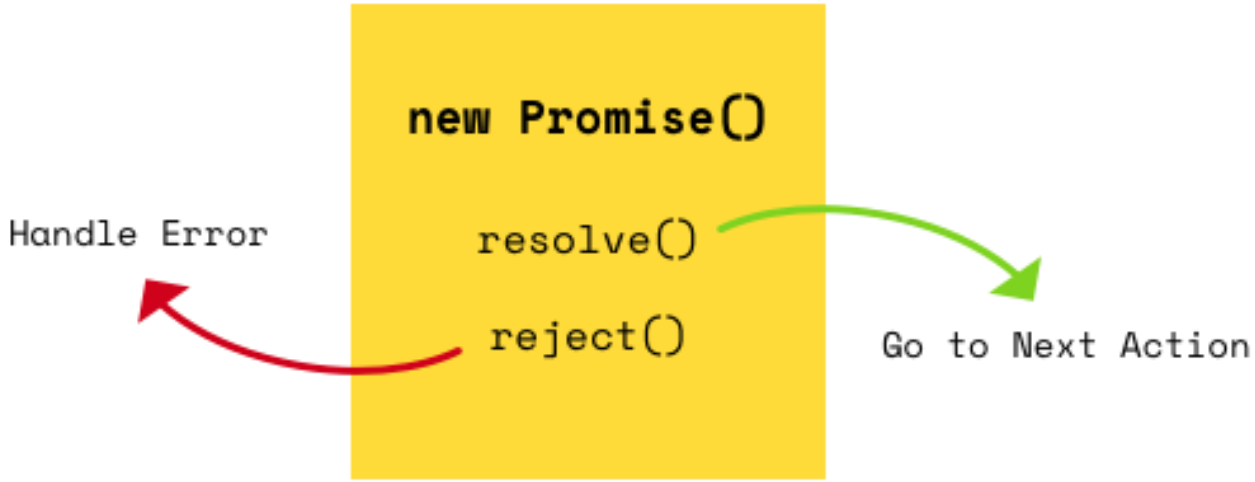
The first Hub for Developers

Ztoupis Konstantinos

HTTP Requests

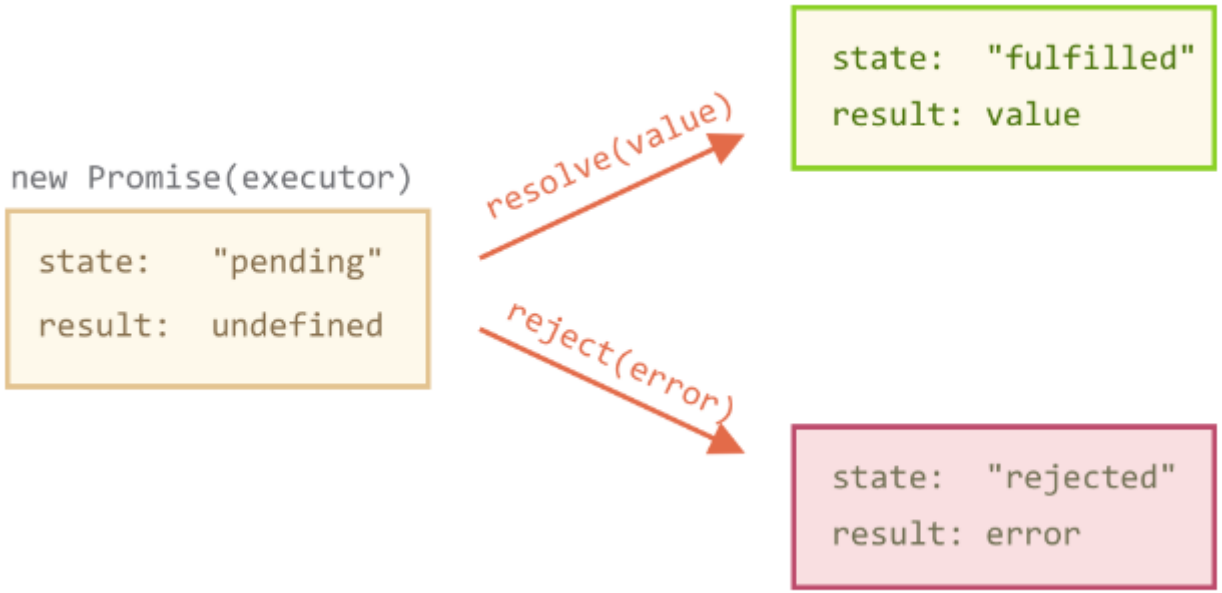
Code.Learn Program:
React

Promise




```
new Promise( /* executor */ function(resolve, reject) { ... } );
```

Promise states



- *pending*: initial state, neither fulfilled nor rejected
- *fulfilled*: meaning that the operation completed successfully
- *rejected*: meaning that the operation failed

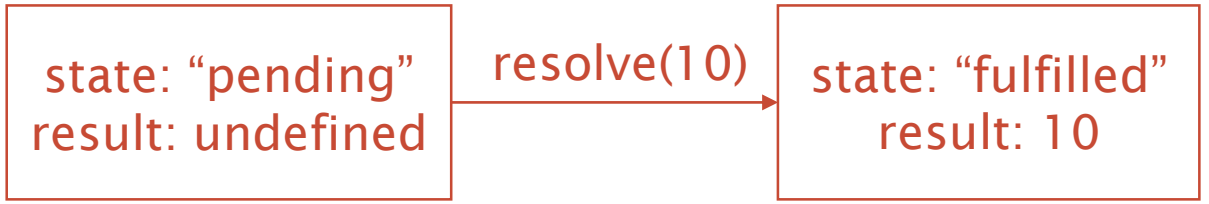
Promise

executor is called automatically and immediately (by the new Promise). Receives two arguments:

- resolve and reject: these functions are pre-defined by the JavaScript engine. So we don't need to create them. We only should call one of them when ready

Promise - resolve

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => resolve(10), 5000);
});
```



Promise - reject

```
const promise = new Promise((resolve, reject)
=> {
  setTimeout(() => reject(new Error("abort")),
```

Code.Hub

Making AJAX calls in React

- What's the React way to fetch data from the server?
- How should I make AJAX calls in React?

API - Fetch

API:

- basically a set of data, often in JSON format with specified endpoints
- when we access data from an API, we want to access specific endpoints within that API framework

Fetch:

- since all the data we want is stored in an API, the fetch is how we request that data
- we can also specify how we want the data to be returned. JSON data is often the easiest to use, so if the server returns anything other than JSON format, if that's what we've requested, we may get an error

Lifecycle Method

- there are several lifecycle methods
- we'll be using three of these lifecycle methods for fetching data from an api:
  - Constructor
  - componentDidMount
  - render

State

- same object, but has different states depending on conditions
- we can change their state depending on how we want to interact with them

3 step process

- Step 1: constructor method
- Step 2: componentDidMount
  - a: fetch
  - b: set state
- Step 3: render
  - a: map
  - b: key and return data

3 step process

Step 1: constructor method:
we only have to worry about two things

- once we call the constructor method, we use super to pass any props from the parent to the child component. Then we set the state. When you set the initial state, you want to set it as empty, or blank
- once we use the fetch to retrieve the data, we'll go in afterwards, and set the new state to the data that we've pulled

Step 2: componentDidMount

- lifecycle methods including "Will" are called before something happens
- lifecycle methods including "Did" are called after something happens
- in an api, we want to call the method after, and then pull the data, which is why we are using componentDidMount instead of componentWillMount
- fetch data and store them to state

Step 3: render

- last part of this lifecyle method and the most important
- map results and set the key. Key attribute is used by react to ensure that dom elements correspond with data objects

setState Problem

fetch data in the componentDidMount lifecycle method and updated the state in the callbacks:

- problem: What if before the data resolves that the component has been unmounted?

cancel the HTTP request when the component dismounts:

- componentWillUnMount: a lifecycle that lets us run some code before a component is to be dismounted

Data fetching with hooks

- The effect hook called useEffect is used to fetch the data from the API and to set the data in the local state of the component with the state hook's update function.

- The effect hook runs when the component mounts but also when the component updates. We only want to fetch data when the component mounts. That's why you can provide an empty array as second argument to the effect hook to avoid activating it on component updates but only for the mounting of the component.

Making AJAX calls in React

5 simple libraries for making AJAX calls

- jQuery $.ajax
- Fetch API
- Axios
- Superagent
- Request

jQuery

- a quick & dirty way to make AJAX calls
- if you are just starting out with React, this can save you a lot of time you are already familiar with jQuery

```
fetchData = () => {
  $.get(API, () => {})
    .done((response) => {
      this.setState({data: response})
    })
    .fail(() => {
       this.setState({error: 'Something went wrong ...'})
    })
    .always(() => {
      alert('finished');
    });
}
```

```
fetchData = () => {
  $.get(API, () => {})
    .done((response) => {
      setContacts(response);
    })
    .fail(() => {
      setError({error: 'Something went wrong ...'})
    })
    .always(() => {
      alert('finished');
    });
}
```

jQuery

- the same old jQuery's $.ajax used inside a React component
- a big library with many functionalities: it doesn't make sense to use it just for making API calls

Fetch API

- a new, simple and standardised API that aims to unify fetching across the web and replace XMLHttpRequest
- has a polyfill for older browsers and should be used in modern web apps
- provides a generic definition of Request and Response objects (and other things involved with network requests)
- provides a definition for related concepts such as CORS and the HTTP origin header semantics, supplanting their separate definitions elsewhere
- takes one mandatory argument, the path to the resource you want to fetch
- returns a Promise that resolves to the Response to that request, whether it is successful or not

Fetch API

- Headers: represents response/request headers, allowing you to query them and take different actions depending on the results
- Request: represents a resource request
- Response: represents the response to a request

```
fetchData = () => {
  fetch(API)
  .then(data => this.setState({data}))
  .catch(error =>
this.setState({error}));
```

```
fetchData = () => {
  fetch(API)
  .then(data => setContacts(data))
  .catch(error => setError(error));
}
```

Axios

- a promise based HTTP client for Node.js and browser
- like fetch, it can work on both client and server
- has many useful features

Axios - Features

- Make XMLHttpRequests from the browser
- Make http requests from node.js
- Supports the Promise API
- Intercept request and response
- Transform request and response data
- Cancel requests
- Automatic transforms for JSON data
- Client side support for protecting against XSRF

```
fetchData = () => {
  axios.get(API)
    .then(response => this.setState({data:
response.data}))
    .catch(error => this.setState({error}));
```

```
fetchData = () => {
  axios.get(API)
    .then(response =>
setContacts(response.data))
    .catch(error => setError(error));
```

Code.Hub

Superagent

- a light weight AJAX API library created for better readability and flexibility
- has a Node.js module with the same API
- as the APIs for client and server are the same, no code change is required in order to make it work in the browser

```
fetchData = () => {
  request.get(API)
    .then(response => this.setState({data:
response.body}))
    .catch(error => this.setState({error}));
```

```
fetchData = () => {
  request.get(API)
    .then(response =>
setContacts(response.body))
    .catch(error => setError({error}));
```

Request

- more that 23k GitHub stars
- one of the most popular Node.js modules

Features:

- Streaming
- Promises & Async/Await
- Forms
- HTTP Authentication
- Custom HTTP Headers
- OAuth Signing
- Proxies
- Unix Domain Sockets
- TLS/SSL Protocol
- Support for HAR 1.2

```
fetchData = () => {
  request(API, (error, response, body) => {
    if (response.statusCode === 200) {
      this.setState({data: JSON.parse(body)})
    } else {
      this.setState({
        error: 'Something went wrong ...'
      });
    }
  });
}
```

Async/Await

- Promises: give us an easier way to deal with asynchrony in our code in a sequential manner
- Async/await functions: a new addition with ES2017 (ES8), help us even more in allowing us to write completely synchronous-looking code while performing asynchronous tasks behind the scenes

Async functions

async keyword: can placed before a function

```
async function f() {
  return 10;
}
f().then(alert);
```

=

```
async function f() {
  return Promise.resolve(10);
}
f().then(alert);
```

“async”: a function always returns a promise. Even If a function actually returns a non-promise value, prepending the function definition with the “async” keyword directs JavaScript to automatically wrap that value in a resolved promise

Await

await keyword makes JavaScript
wait until that promise settles
and returns its result

```
async function f() {
  const promise = new Promise((resolve) => {
    setTimeout(() => resolve(1), 5000)
  });
  const result = await promise;
  alert(result);
}

f();
```

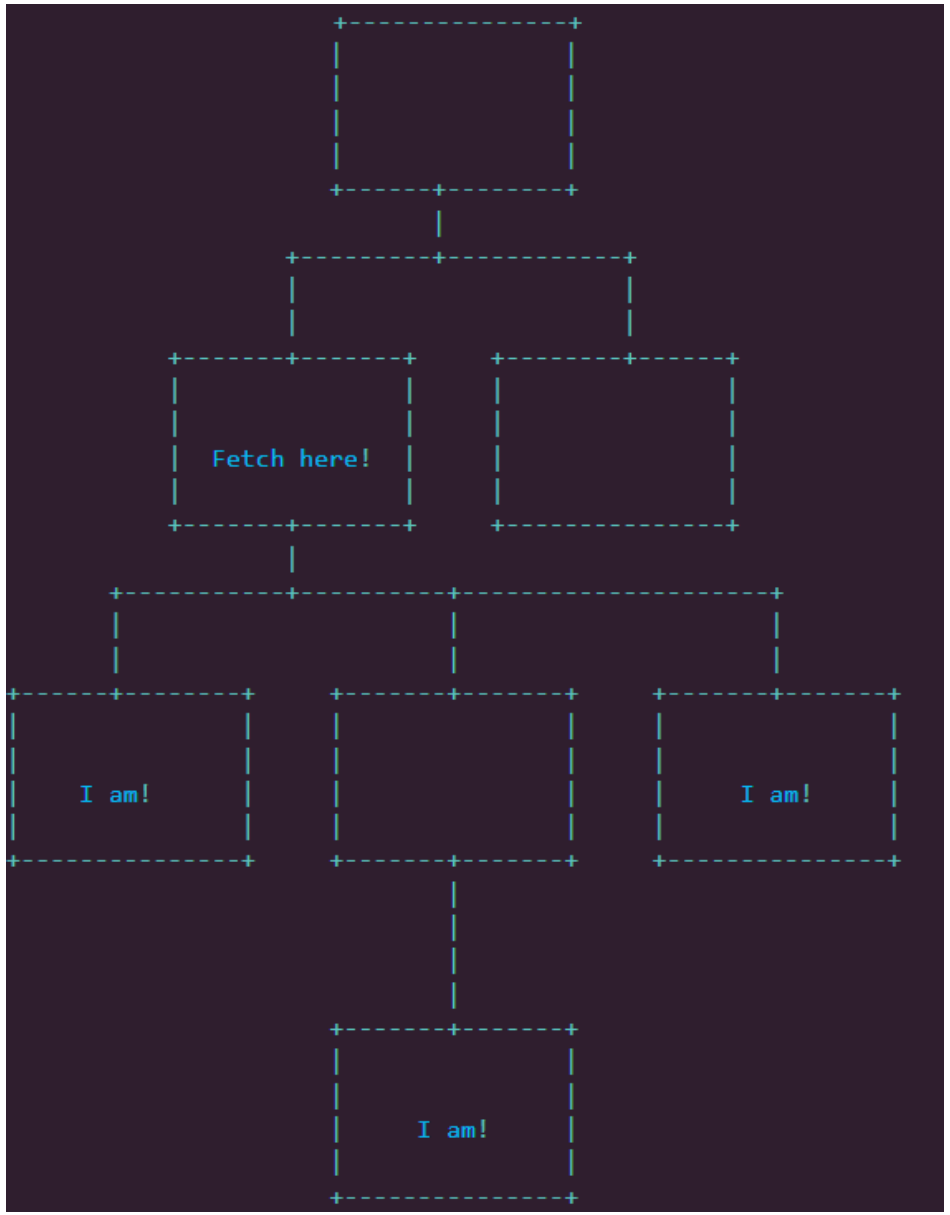
- makes JavaScript wait until the promise settles, and then go on with the result
- doesn't cost any CPU resources, because the engine can do other jobs meanwhile: execute other scripts, handle events etc
- a more elegant syntax of getting the promise result than promise.then, easier to read and write

Async/Await

```
async fetchData = () => {
  try {
    const response = await
axios.get(API);
    this.setState({data: response.data});
  } catch (error) {
    this.setState({error});
  }
```

```
const fetchData = async () => {
  try {
    const response = await
axios.get(API);
    setContacts(response.data);
  } catch (error) {
    setError(error);
  }
```

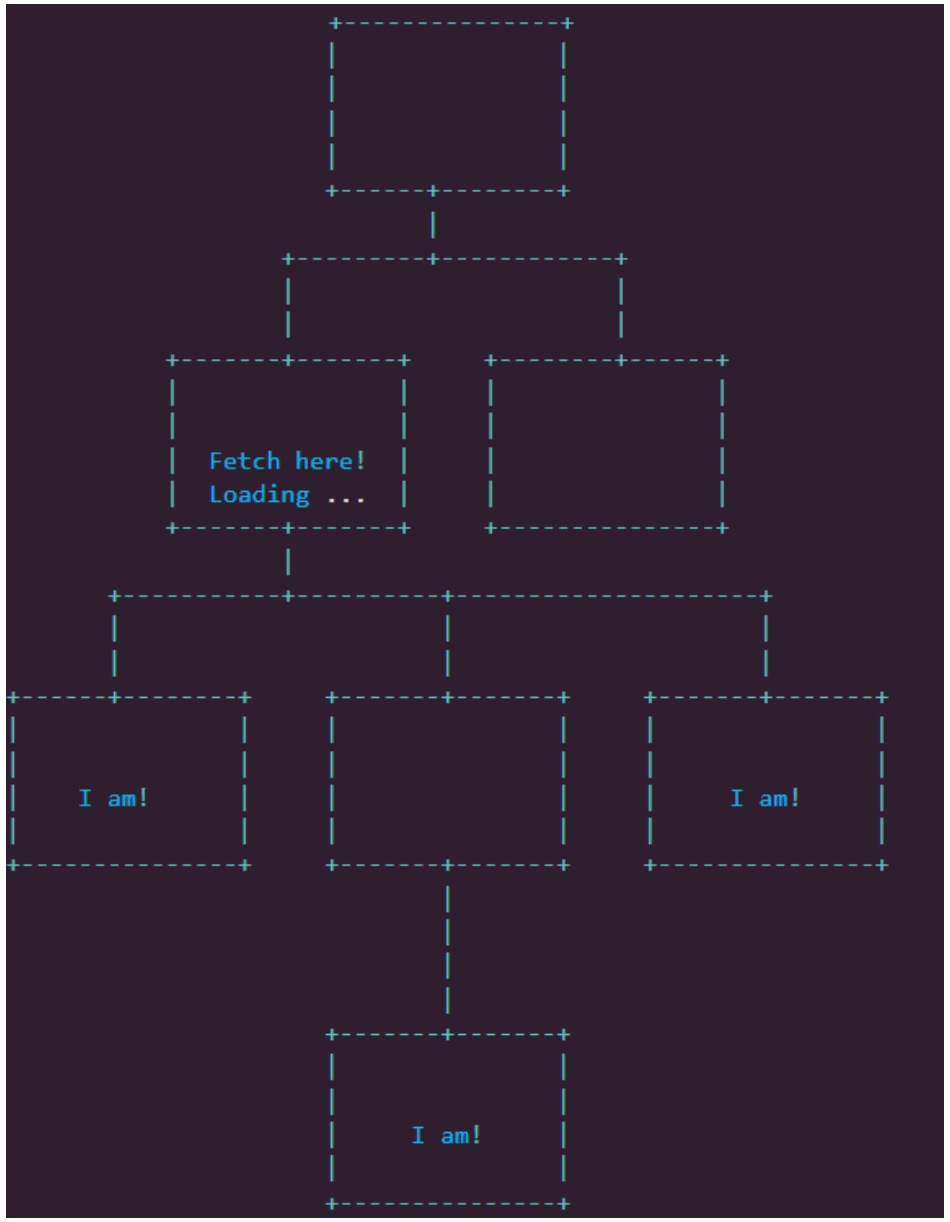
Where to fetch in component tree?



The fetching component should be a common parent component for all these components

Code.Hub

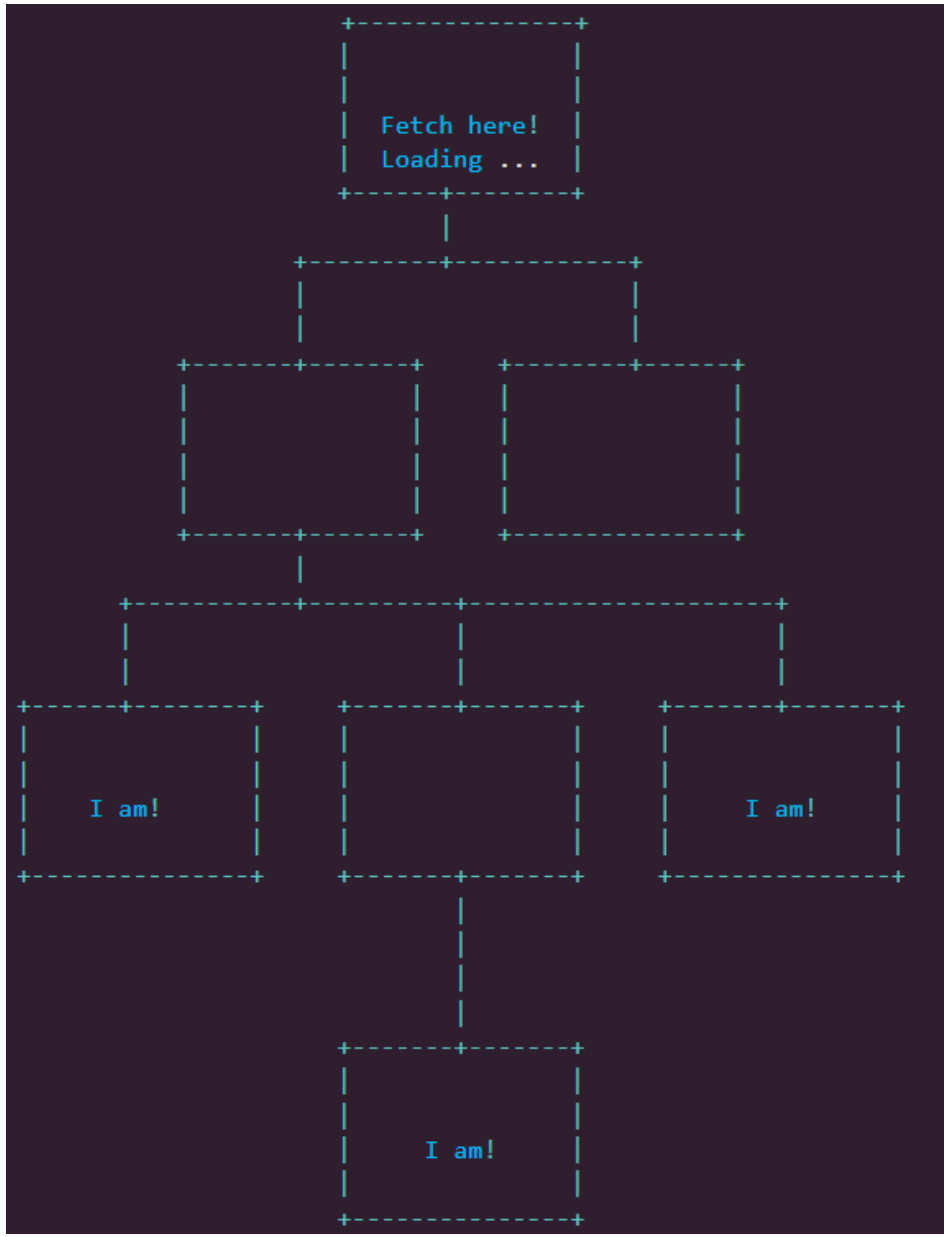
Where to show a loading indicator?



- loading indicator could be shown in the common parent component
- common parent component would still be the component to fetch the data

Code.Hub

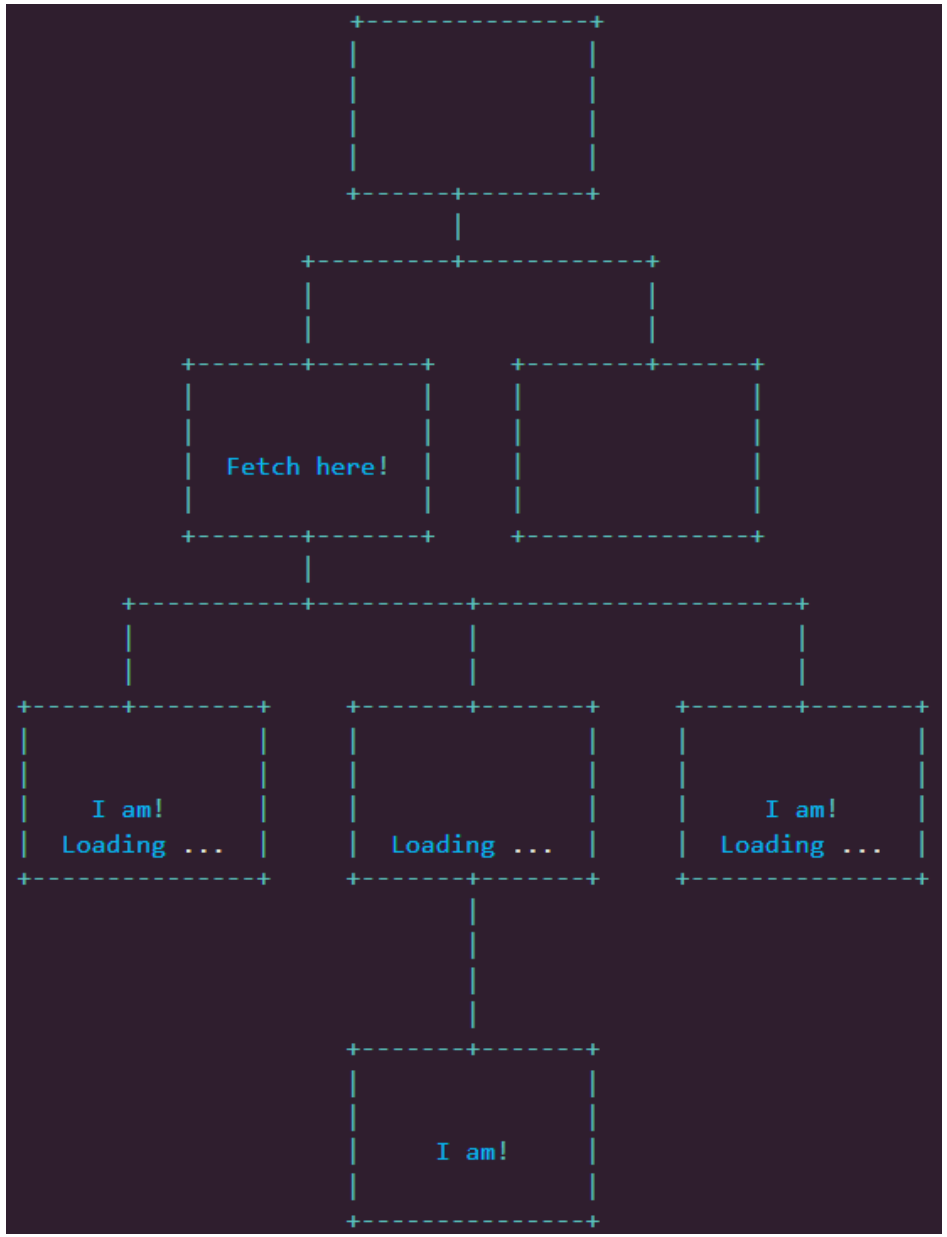
Loading in top level component



when the loading indicator should be shown in a more top level component, the data fetching needs to be lifted up to this component

Code.Hub

Loading in child components



- when the loading indicator should be shown in child components of the common parent component, it would still be the component to fetch the data
- the loading indicator could then be passed down to all child components

Code.Hub

Where to show an error message?

- the same rules for the loading indicator apply
- basically everything on where to fetch the data in your React component hierarchy