[How to cancel a fetch on componentWillUnmount](https://stackoverflow.com/questions/49906437/how-to-cancel-a-fetch-on-componentwillunmount)?

When you fire a Promise it might take a few seconds before it resolves and by that time user might have navigated to another place in your app. So when Promise resolves setState is executed on unmounted component and you get an error - just like in your case. This may also cause memory leaks.

Approach 1:

As a last resort technique (it's an antipattern) - you can keep a variable to check whether the component is still mounted:

Setting a \_isMounted property to true in componentDidMount and set it to false in componentWillUnmount, and use this variable to check your component’s status.

componentDidMount(){

this.\_isMounted  = true;

this.props.fetchData().then((response) => {

if(this.\_isMounted ) {

this.setState({ data: response })

}

})

}

componentWillUnmount(){

this.\_isMounted  = false;

}

Approach 2:

Using react hooks

import React, { useState, useEffect } from "react";

export default function Page() {

const value = usePromise("https://something.com/api/");

return (

<p>{value ? value : "fetching data..."}</p>

);

}

function usePromise(url) {

const [value, setState] = useState(null);

useEffect(() => {

let isMounted = true; // track whether component is mounted

request.get(url)

.then(result => {

if (isMounted) {

setState(result);

}

});

return () => {

// clean up

isMounted = false;

};

}, []); // only on "didMount"

return value;

}

Approach 3:

You can use [AbortController](https://developer.mozilla.org/en-US/docs/Web/API/AbortController) to cancel a fetch request. This API is provided by the DOM standard, and that's the entire API. It's deliberately generic so it can be used by other web standards and JavaScript libraries.

class FetchComponent extends React.Component{

state = { todos: [] };

controller = new AbortController();

componentDidMount(){

fetch('https://jsonplaceholder.typicode.com/todos',{

signal: this.controller.signal

})

.then(res => res.json())

.then(todos => this.setState({ todos }))

.catch(e => alert(e.message));

}

componentWillUnmount(){

this.controller.abort();

}

render(){

return null;

}

}

class App extends React.Component{

state = { fetch: true };

componentDidMount(){

this.setState({ fetch: false });

}

render(){

return this.state.fetch && <FetchComponent/>

}

}

ReactDOM.render(<App/>, document.getElementById('root'))

Approach 3:

The friendly people at React [recommend](https://reactjs.org/blog/2015/12/16/ismounted-antipattern.html) wrapping your fetch calls/promises in a cancelable promise. While there is no recommendation in that documentation to keep the code separate from the class or function with the fetch, this seems advisable because other classes and functions are likely to need this functionality, code duplication is an anti-pattern, and regardless the lingering code should be disposed of or canceled in componentWillUnmount(). As per React, you can call cancel() on the wrapped promise in componentWillUnmount to avoid setting state on an unmounted component.

const makeCancelable = (promise) => {

let hasCanceled\_ = false;

const wrappedPromise = new Promise((resolve, reject) => {

promise.then(

val => hasCanceled\_ ? reject({isCanceled: true}) : resolve(val),

error => hasCanceled\_ ? reject({isCanceled: true}) : reject(error)

);

});

return {

promise: wrappedPromise,

cancel() {

hasCanceled\_ = true;

},

};

};

const cancelablePromise = makeCancelable(fetch('LINK HERE'));

constructor(props){

super(props);

this.state = {

isLoading: true,

dataSource: [{

name: 'loading...',

id: 'loading',

}]

}

}

componentDidMount(){

cancelablePromise.

.then((response) => response.json())

.then((responseJson) => {

this.setState({

isLoading: false,

dataSource: responseJson,

}, () => {

});

})

.catch((error) =>{

console.error(error);

});

}

componentWillUnmount() {

cancelablePromise.cancel();

}

AbortController

The **AbortController** interface represents a controller object that allows you to abort one or more DOM requests as and when desired.

You can create a new AbortController object using the [AbortController.AbortController()](https://developer.mozilla.org/en-US/docs/Web/API/AbortController/AbortController" \o "The AbortController() constructor creates a new AbortController object instance.) constructor. Communicating with a DOM request is done using an [AbortSignal](https://developer.mozilla.org/en-US/docs/Web/API/AbortSignal" \o "The AbortSignal interface represents a signal object that allows you to communicate with a DOM request (such as a Fetch) and abort it if required via an AbortController object.) object.

[**AbortController()**](https://developer.mozilla.org/en-US/docs/Web/API/AbortController)

Creates a new AbortController object instance.

[**AbortController.signal**](https://developer.mozilla.org/en-US/docs/Web/API/AbortController/signal)

Returns a [AbortSignal](https://developer.mozilla.org/en-US/docs/Web/API/AbortSignal" \o "The AbortSignal interface represents a signal object that allows you to communicate with a DOM request (such as a Fetch) and abort it if required via an AbortController object.) object instance, which can be used to communicate with/abort a DOM request.

[**AbortController.abort()**](https://developer.mozilla.org/en-US/docs/Web/API/AbortController/abort)

Aborts a DOM request before it has completed. This is able to abort [fetch requests](https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/fetch), consumption of any response [Body](https://developer.mozilla.org/en-US/docs/Web/API/Body), and streams.

## Example:

In the following snippet, we aim to download a video using the [Fetch API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API).

We first create a controller using the [AbortController()](https://developer.mozilla.org/en-US/docs/Web/API/AbortController/AbortController" \o "The AbortController() constructor creates a new AbortController object instance.) constructor, then grab a reference to its associated [AbortSignal](https://developer.mozilla.org/en-US/docs/Web/API/AbortSignal" \o "The AbortSignal interface represents a signal object that allows you to communicate with a DOM request (such as a Fetch) and abort it if required via an AbortController object.) object using the [AbortController.signal](https://developer.mozilla.org/en-US/docs/Web/API/AbortController/signal" \o "The signal read-only property of the AbortController interface returns an AbortSignal object instance, which can be used to communicate with/abort a DOM request as desired.) property.

When the [fetch request](https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/fetch) is initiated, we pass in the AbortSignal as an option inside the request's options object (see {signal}, below). This associates the signal and controller with the fetch request and allows us to abort it by calling [AbortController.abort()](https://developer.mozilla.org/en-US/docs/Web/API/AbortController/abort" \o "The abort() method of the AbortController interface aborts a DOM request (e.g. a Fetch request) before it has completed. This is able to abort fetch requests, consumption of any response Body, and streams.), as seen below in the second event listener.

var controller = new AbortController();

var signal = controller.signal;

var downloadBtn = document.querySelector('.download');

var abortBtn = document.querySelector('.abort');

downloadBtn.addEventListener('click', fetchVideo);

abortBtn.addEventListener('click', function() {

controller.abort();

console.log('Download aborted');

});

function fetchVideo() {

...

fetch(url, {signal}).then(function(response) {

...

}).catch(function(e) {

reports.textContent = 'Download error: ' + e.message;

})

}

**Note**: When abort() is called, the fetch() promise rejects with an AbortError.

Anti-Pattern

An anti-pattern is a bad design that is worthy of documenting. Examples of anti-patterns in JavaScript are the following:

* Polluting the global namespace by defining a large number of variables in the global context.
* Passing strings rather than functions to either setTimeout or 0073etInterval, as this triggers the use of eval() internally.
* Modifying the Object class prototype (this is a particularly bad anti-pattern).
* Using JavaScript in an inline form as this is inflexible.

**Three JavaScript Anti-Patterns and How To Avoid Them**

**Extending Object**

Perhaps the most frowned-upon anti-pattern is the extending of the base Object's prototype. This is considered to be very bad practice mainly because it breaks *for in* loops. Consider the following:

var obj = {a: "A", b: "B", c: "C", d: "D"};

for (var key in obj) {

   alert(key +': '+obj[key]); //displays "a: A", "b: B", "c: C", "d: D"

}

Object.prototype.e = "E";

for (var key in obj) {

   alert(key +': '+obj[key]); //displays "a: A", "b: B", "c: C", "d: D", "e: E"

}

var obj2 = {a2: "A2", b2: "B2", c2: "C2", d2: "D2"};

for (var key in obj2) {

   alert(key +': '+obj2[key]); //displays "a2: A2", "b2: B2", "2c: C2", "d2: D2", "e: E"

}

The e property is now inherited by all objects, whether you want it or not.

The preferred way to add shared object attributes is to create your object first and extend it.