react-use-websocket

4.13.0 • Public • Published 3 months ago

React useWebSocket

Live Demo

Note: `wss://demos.kaazing.com/echo` has been down lately, so the demo will fail to connect when using that as the endpoint. On the plus side, this demonstrates the behavior of a connection failure.

Test in StackBlitz

React Hook designed to provide robust WebSocket integrations to your React Components. Experimental support for SocketIO (read documentation below for more information)

Pull requests welcomed!

New in 4.0.0

- `react-use-websocket` now supports (and depends on) React 18. If you are not ready to upgrade to React 18, please install version `3.0.0`:

```
npm install --save react-use-websocket@3.0.0

//or

yarn add react-use-websocket@3.0.0
```

New in 2.0.0

- `useWebSocket` now returns an object instead of an array. This allows you to pick out specific features/properties to suit your use-case as well as removing mental overhead of keeping track of item order.
- `lastJsonMessage` and `sendJsonMessage` added to return value to reduce need to stringify and parse outgoing and incoming messages at the component level.
- The optional object passed as the second parameter no longer needs to be static.
- Components can close/unsubscribe from a WebSocket by passing `false` as the third parameter. This provides a more explicit solution than the previous method of setting the `socketUrl` to `null`. Both methods work and are supported usage.

Example Implementation

```
import React, { useState, useCallback, useEffect } from 'react';
import useWebSocket, { ReadyState } from 'react-use-websocket';

export const WebSocketDemo = () => {
  //Public API that will echo messages sent to it back to the client
  const [socketUrl, setSocketUrl] = useState('wss://echo.websocket.org
  const [messageHistory, setMessageHistory] =
    useState < MessageEvent < any > [] > [];

  const { sendMessage, lastMessage, readyState } = useWebSocket(socket
```

```
useEffect(() => {
  if (lastMessage !== null) {
    setMessageHistory((prev) => prev.concat(lastMessage));
  }
}, [lastMessage]);

const handleClickChangeSocketUrl = useCallback(
  () => setSocketUrl('wss://demos.kaazing.com/echo'),
  []
);

const handleClickSendMessage = useCallback(() => sendMessage('Hello'

const connectionStatus = {
  [ReadyState.CONNECTING]: 'Connecting',
  [ReadyState.OPEN]: 'Open',
  [ReadyState.CLOSING]: 'Closing',
  [ReadyState.CLOSED]: 'Closed',
  [ReadyState.UNINSTANTIATED]: 'Uninstantiated',
}[readyState];

return (
  <div>
    <button onClick={handleClickChangeSocketUrl}>
      Click Me to change Socket Url
    </button>
    <button
      onClick={handleClickSendMessage}
      disabled={readyState !== ReadyState.OPEN}
    >
      Click Me to send 'Hello'
    </button>
    <span>The WebSocket is currently {connectionStatus}</span>
    {lastMessage ? <span>Last message: {lastMessage.data}</span> : r
```

```
      <ul>
        {messageHistory.map((message, idx) => (
          <span key={idx}>{message ? message.data : null}</span>
        ))}
      </ul>
    </div>
  );
};
```

From the example above, the component will rerender every time the `readyState` of the WebSocket changes, as well as when the WebSocket receives a message (which will change `lastMessage`). `sendMessage` is a memoized callback that will pass the message to the current WebSocket (referenced to internally with `useRef`).

A demo of this can be found **here**. Each component uses its own `useWebSocket` hook. This implementation takes advantage of passing an optional options object (documented below). Among setting event callbacks (for `onmessage`, `onclose`, `onerror`, and `onopen`) that will log to the console, it is using the `share` option -- if multiple components pass the same socketUrl to `useWebSocket` and with `share` set to true, then only a single WebSocket will be created and `useWebSocket` will manage subscriptions/unsubscriptions internally. `useWebSocket` will keep track of how many subscribers any given WebSocket has and will automatically free it from memory once there are no subscribers remaining (a subscriber unsubscribes when it either unmounts or changes its socketUrl). Of course, multiple WebSockets can be created with the same target url, and so components are not required to share the same communication pipeline.

Features

- Handles reconnect logic
- Multiple components can (optionally) use a single WebSocket, which is closed and cleaned up when all subscribed components have unsubscribed/unmounted
- Written in TypeScript
- Socket.io support
- Heartbeat support

- No more waiting for the WebSocket to open before messages can be sent. Pre-connection messages are queued up and sent on connection
- Provides direct access to unshared WebSockets, while proxying shared WebSockets. Proxied WebSockets provide subscribers controlled access to the underlying (shared) WebSocket, without allowing unsafe behavior
- Seamlessly works with server-sent-events and the EventSource API

Getting Started

```
npm install react-use-websocket
```

```
import useWebSocket from 'react-use-websocket';

// In functional React component

// This can also be an async getter function. See notes below on Async
const socketUrl = 'wss://echo.websocket.org';

const {
  sendMessage,
  sendJsonMessage,
  lastMessage,
  lastJsonMessage,
  readyState,
  getWebSocket,
} = useWebSocket(socketUrl, {
  onOpen: () => console.log('opened'),
  //Will attempt to reconnect on all close events, such as server shut
  shouldReconnect: (closeEvent) => true,
});
```

Interface

```
type UseWebSocket<T = unknown> = (
  //Url can be return value of a memoized async function.
  url: string | () => Promise<string>,
  options: {
    fromSocketIO?: boolean;
    queryParams?: { [field: string]: any };
    protocols?: string | string[];
    share?: boolean;
    onOpen?: (event: WebSocketEventMap['open']) => void;
    onClose?: (event: WebSocketEventMap['close']) => void;
    onMessage?: (event: WebSocketEventMap['message']) => void;
    onError?: (event: WebSocketEventMap['error']) => void;
    onReconnectStop?: (numAttempts: number) => void;
    shouldReconnect?: (event: WebSocketEventMap['close']) => boolean;
    reconnectInterval?: number | ((lastAttemptNumber: number) => numbe
    reconnectAttempts?: number;
    filter?: (message: WebSocketEventMap['message']) => boolean;
    disableJson?: boolean;
    retryOnError?: boolean;
    eventSourceOptions?: EventSourceInit;
    heartbeat?: boolean | {
      message?: "ping" | "pong" | string | (() => string);
      returnMessage?: "ping" | "pong" | string;
      timeout?: number;
      interval?: number;
    };
  } = {},
  shouldConnect: boolean = true,
): {
  sendMessage: (message: string, keep: boolean = true) => void,
  //jsonMessage must be JSON-parsable
  sendJsonMessage: (jsonMessage: T, keep: boolean = true) => void,
  //null before first received message
```

```
  lastMessage: WebSocketEventMap['message'] | null,
  //null before first received message. If message.data is not JSON pa
  lastJsonMessage: T | null,
  // -1 if uninstantiated, otherwise follows WebSocket readyState mapp
  readyState: number,
  // If using a shared websocket, return value will be a proxy-wrapped
  getWebSocket: () => (WebSocketLike | null),
}
```

Requirements

- React 16.8+
- Cannot be used within a class component (must be a functional component that supports React Hooks)

Async Urls

Instead of passing a string as the first argument to useWebSocket, you can pass a function that returns a string (or a promise that resolves to a string). It's important to note, however, that other rules still apply -- namely, that if the function reference changes, then it will be called again, potentially instantiating a new WebSocket if the returned url changes.

```
import useWebSocket from 'react-use-websocket';

// In functional React component
const getSocketUrl = useCallback(() => {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve('wss://echo.websocket.org');
    }, 2000);
  });
}, []);
```

```
const { sendMessage, lastMessage, readyState, getWebSocket } = useWebS
  getSocketUrl,
  STATIC_OPTIONS
);
```

If `getSocketUrl` throws an error and `Options#retryOnError` is `true`, then `getSocketUrl` will be called at an interval consistent with the retry behavior defined by `Options#reconnectAttempts` and `Options#reconnectInterval`.

API

sendMessage

```
type sendMessage = (message: string, keep: boolean = true) => void;
```

The argument sent through sendMessage will be passed directly to WebSocket# `send`. `sendMessage` will be static, and thus can be passed down through children components without triggering prop changes. Messages sent before the WebSocket is open will be queued up and sent on connection. If you don't want to use messages queue for a particular message you should use a 'keep' parameter.

sendJsonMessage

```
type sendJsonMessage = (message: any, keep: boolean = true) => void;
```

Message will first be passed through `JSON.stringify`.

lastMessage

```
type lastMessage = WebSocketEventMap['message'];
```

Will be an unparsed `MessageEvent` received from the WebSocket.

lastJsonMessage

```
type lastJsonMessage = any;
```

A `JSON.parse` d object from the `lastMessage` . If `lastMessage` is not a valid JSON string, `lastJsonMessage` will be an empty object. If `Options#disableJson` is `true` , `lastMessage` will not be automatically parsed, and `lastJsonMessage` will always be `null` .

readyState

```
enum ReadyState {
  UNINSTANTIATED = -1,
  CONNECTING = 0,
  OPEN = 1,
  CLOSING = 2,
  CLOSED = 3,
}
```

Will be an integer representing the `readyState` of the WebSocket. `-1` is not a valid WebSocket `readyState` , but instead indicates that the WebSocket has not been instantiated yet (either because the url is `null` or connect param is `false`)

getWebSocket

```
type getWebSocket = () => WebSocketLike | Proxy<WebSocketLike>;
```

If the WebSocket is shared, calling this function will lazily instantiate a `Proxy` instance that wraps the underlying WebSocket. You can get and set properties on the return value that will directly interact with the WebSocket, however certain properties/methods are protected (cannot invoke `close` or `send` , and cannot redefine any of the event handlers like `onmessage` , `onclose` , `onopen` and `onerror` . An example of using this:

```
const { sendMessage, lastMessage, readyState, getWebSocket } = useWebS
  'wss://echo.websocket.org',
  { share: true }
);

useEffect(() => {
```

```
    console.log(getWebSocket().binaryType);
    //=> 'blob'

    //Change binaryType property of WebSocket
    getWebSocket().binaryType = 'arraybuffer';

    console.log(getWebSocket().binaryType);
    //=> 'arraybuffer'

    //Attempt to change event handler
    getWebSocket().onmessage = console.log;
    //=> A warning is logged to console: 'The WebSocket's event handlers

    //Attempt to change an immutable property
    getWebSocket().url = 'www.google.com';
    console.log(getWebSocket().url);
    //=> 'wss://echo.websocket.org'

    //Attempt to call webSocket#send
    getWebSocket().send('Hello from WebSocket');
    //=> No message is sent, and no error thrown (a no-op function was r
  }, []);
```

If the WebSocket is not shared (via options), then the return value is the underlying WebSocket, and thus methods such as `close` and `send` can be accessed and used.

Reconnecting

By default, `useWebSocket` will not attempt to reconnect to a WebSocket. This behavior can be modified through a few options. To attempt to reconnect on error events, set `Options#retryOnError` to `true`. Because `CloseEvent`s are less straight forward (e.g., was it triggered intentionally by the client or by something unexpected by the server restarting?), `Options#shouldReconnect` must be provided as a callback, with the socket

`CloseEvent` as the first and only argument, and a return value of either `true` or `false`. If `true`, `useWebSocket` will attempt to reconnect up to a specified number of attempts (with a default of `20`) at a specified interval (with a default of `5000` (ms)). The option properties for attempts is `Options#reconnectAttempts` and the interval is `Options#reconnectInterval`. As an example:

```
const didUnmount = useRef(false);

const [sendMessage, lastMessage, readyState] = useWebSocket(
  'wss://echo.websocket.org',
  {
    shouldReconnect: (closeEvent) => {
      /*
      useWebSocket will handle unmounting for you, but this is an exam
      case in which you would not want it to automatically reconnect
    */
      return didUnmount.current === false;
    },
    reconnectAttempts: 10,
    reconnectInterval: 3000,
  }
);

useEffect(() => {
  return () => {
    didUnmount.current = true;
  };
}, []);
```

Alternatively, you can provide a function for `Options#reconnectInterval` that accepts as a parameter the nth last attempt and returns a number, which represents how long the next interval should be. This should enable a higher degree of control if you wish to employ more advanced reconnect strategies (such as **Exponential Backoff**):

```
const [sendMessage, lastMessage, readyState] = useWebSocket(
  'wss://echo.websocket.org',
  {
    shouldReconnect: (closeEvent) => true,
    reconnectAttempts: 10,
    //attemptNumber will be 0 the first time it attempts to reconnect,
    reconnectInterval: (attemptNumber) =>
      Math.min(Math.pow(2, attemptNumber) * 1000, 10000),
  }
);
```

Options

```
interface Options {
  share?: boolean;
  shouldReconnect?: (event: WebSocketEventMap['close']) => boolean;
  reconnectInterval?: number | ((lastAttemptNumber: number) => number)
  reconnectAttempts?: number;
  filter?: (message: WebSocketEventMap['message']) => boolean;
  disableJson?: boolean;
  retryOnError?: boolean;
  onOpen?: (event: WebSocketEventMap['open']) => void;
  onClose?: (event: WebSocketEventMap['close']) => void;
  onMessage?: (event: WebSocketEventMap['message']) => void;
  onError?: (event: WebSocketEventMap['error']) => void;
  onReconnectStop?: (numAttempted: number) => void;
  fromSocketIO?: boolean;
  queryParams?: {
    [key: string]: string | number;
  };
  protocols?: string | string[];
  eventSourceOptions?: EventSourceInit;
```

```
    heartbeat?:
      | boolean
      | {
          message?: 'ping' | 'pong' | string;
          returnMessage?: 'ping' | 'pong' | string;
          timeout?: number;
          interval?: number;
        };
  }
```

shouldReconnect

See section on **Reconnecting**.

reconnectInterval

Number of milliseconds to wait until it attempts to reconnect. Default is 5000. Can also be defined as a function that takes the last attemptCount and returns the amount of time for the next interval. See **Reconnecting** for an example of this being used.

Event Handlers: Callback

Each of `Options#onMessage`, `Options#onError`, `Options#onClose`, and `Options#onOpen` will be called on the corresponding WebSocket event, if provided. Each will be passed the same event provided from the WebSocket.

onReconnectStop

If provided in options, will be called when websocket exceeds reconnect limit, either as provided in the options or the default value of 20.

share: Boolean

If set to `true`, a new WebSocket will not be instantiated if one for the same url has already been created for another component. Once all subscribing components have either unmounted or changed their target socket url, shared WebSockets will be closed and cleaned up. No other APIs should be affected by this.

fromSocketIO: Boolean

SocketIO acts as a layer on top of the WebSocket protocol, and the required client-side implementation involves a few peculiarities. If you have a SocketIO back-end, or are converting a client-side application that uses the socketIO library, setting this to `true` might be enough to allow `useWebSocket` to work interchangeably. This is an experimental option as the SocketIO library might change its API at any time. This was tested with Socket IO `2.1.1`.

queryParams: Object

Pass an object representing an arbitrary number of query parameters, which will be converted into stringified query params and appended to the WebSocket url.

```
const queryParams = {
  user_id: 1,
  room_id: 5,
};
//<url>?user_id=1&room_id=5
```

useSocketIO

SocketIO sends messages in a format that isn't JSON-parsable. One example is:

```
"42["Action",{"key":"value"}]"
```

An extension of this hook is available by importing `useSocketIO`:

```
import { useSocketIO } from 'react-use-websocket';

//Same API in component
const { sendMessage, lastMessage, readyState } = useSocketIO(
  'http://localhost:3000/'
);
```

It is important to note that `lastMessage` will not be a `MessageEvent`, but instead an object with two keys: `type` and `payload`.

heartbeat

If the `heartbeat` option is set to `true` or has additional options, the library will send a 'ping' message to the server every `interval` milliseconds. If no response is received within `timeout` milliseconds, indicating a potential connection issue, the library will close the connection. You can customize the 'ping' message by changing the `message` property in the `heartbeat` object. If a `returnMessage` is defined, it will be ignored so that it won't be set as the `lastMessage`.

```
const { sendMessage, lastMessage, readyState } = useWebSocket(
  'ws://localhost:3000',
  {
    heartbeat: {
      message: 'ping',
      returnMessage: 'pong',
      timeout: 60000, // 1 minute, if no response is received, the cor
      interval: 25000, // every 25 seconds, a ping message will be ser
    },
  }
);
```

filter: Callback

If a function is provided with the key `filter`, incoming messages will be passed through the function, and only if it returns `true` will the hook pass along the `lastMessage` and update your component.

Example:

```
filter: (message) => {
  // validate your message data
  if (isPingMessage(message.data)) {
    // do stuff or simply return false
    updateHeartbeat()
    return false
```

```
    } else {
      return true
    }
  },
```

The component will rerender every time the WebSocket receives a message that does not match your conditional in this case `isPingMessage`, if the condition is true, you can do some stuff, for this example that is updating the heartbeat time, but you could just avoid unnecessary renders simply returning `false`.

disableJson: Boolean

If `true`, `lastMessage` will not be automatically parsed and returned as `lastJsonMessage`, in which case `lastJsonMessage` will always be `null`.

useEventSource

```
import { useEventSource } from 'react-use-websocket';

//Only the following three properties are provided
const { lastEvent, getEventSource, readyState } = useEventSource(
  'http://localhost:3000/',
  {
    withCredentials: true,
    events: {
      message: (messageEvent) => {
        console.log('This has type "message": ', messageEvent);
      },
      update: (messageEvent) => {
        console.log('This has type "update": ', messageEvent);
      },
    },
  }
);
```

If used, an **EventSource** will be instantiated instead of a WebSocket. Although it shares a very similar API with a WebSocket, there are a few differences:

- There is no onclose event, nor is there an event for readyState changes -- as such, this library can only 'track' the first two readyStates: CONNECTING (0) and OPEN (1). The `EventSource` will close when your component unmounts.
- Currently, the library will set the readyState to CLOSED on the underlying `EventSource` 's onerror callback, and will also trigger `Options#onClose` , if provided. In this case, reconnect logic is driven by `Options#retryOnError` , instead of `Options#shouldReconnect` .
- There is no 'CLOSING' readyState for `EventSource` , and as such, the CLOSED readyState is `2` for an `EventSource` , whereas it is `3` for a WebSocket. For purposes of internal consistency, the `readyState` returned by `useWebSocket` will follow the `WebSocket` enumeration and use `3` for the CLOSED event for both instance types.
- `getEventSource` will return the underlying EventSource, even if `Options#share` is used -- as opposed to the `WebSocket` equivalent which returns a `Proxy` .
- There is no concept of sending messages from the client, and as such `sendMessage` will not be provided.

Reset Global State

There are some cases when the global state of the library won't reset with the page. The main behavior relies on the fact that a single page application operates only in one window, but some scenarios allow us to make a new window via `window.open` and inject code there. In that case, child window will be closed, but the global state of the library remains the same in the main window. This happens because react does not finish components lifecycle on window close.

To avoid troubles with the new initialization of components related to the same URL, you can reset the global state for a specific connection based on your own logic.

```
import React, { useEffect } from 'react';
import { resetGlobalState } from 'react-use-websocket';

// insside second window opened via window.open
export const ChildWindow = () => {
  useEffect(() => {
```

```
    window.addEventListener('unload', () => {
      resetGlobalState('wss://echo.websocket.org');
    });
  }, []);
};
```

Keywords

react react-hooks websocket websockets

Install

```
> npm i react-use-websocket
```

Repository

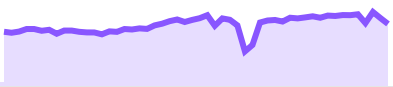
github.com/robtaussig/react-use-websocket

Homepage

github.com/robtaussig/react-use-websocket#readme

Weekly Downloads

189,255



Version	License
4.13.0	MIT

Unpacked Size	Total Files
200 kB	78

Last publish

3 months ago

Collaborators

>_Try on RunKit

Report malware

Support

Help

Advisories

Status

Contact npm

Company

About

Blog

Press

Terms & Policies

Policies

Terms of Use

Code of Conduct

Privacy