Here are some best practices for developing with React:

1. Use functional components for simple logic, and use class-based components for stateful logic.
2. Use the `useState` hook for managing component state, rather than using the `this.state` property.
3. Use the `useEffect` hook for handling side effects, rather than using `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` lifecycle methods.
4. Use the `useContext` hook for managing context, rather than using the `contextType` property and the `this.context` property.
5. Use the `useMemo` and `useCallback` hooks to optimize performance by memoizing expensive calculations and functions.
6. Use the `React.memo` higher-order component for optimizing the performance of functional components.
7. Use the `PureComponent` base class for optimizing the performance of class-based components.
8. Use the `Fragment` component for returning multiple elements from a component's render method, rather than wrapping the elements in a `div`.
9. Use the `forwardRef` higher-order component for forwarding refs to DOM elements.
10. Use the `lazy` and `Suspense` components for code splitting and lazy loading.

By following these best practices, you can develop efficient and maintainable React applications.

Prop drilling, also known as "prop tunneling," occurs when a component deep in the component tree needs to access props that are only available to its ancestors. This can lead to a situation where props need to be passed down through multiple levels of components, even if those components don't need the props for their own rendering.

To avoid prop drilling, you can use a technique called "lifting state up." This involves moving the shared state to a common ancestor component, and then passing the state down to the components that need it as props.

For example, consider a component tree like this:

If the `UserProfile` component needs access to a user ID that is only available to the `App` component, you can lift the state up to the `App` component and pass the user ID down as a prop to the `UserProfile` component:

Copy code

```
// App component
function App() {
  const [userId, setUserId] = useState(null);
  return (
    <UserList userId={userId} setUserId={setUserId} />
  );
}

// UserList component
function UserList({ userId, setUserId }) {
  return (
    <User userId={userId} setUserId={setUserId} />
  );
}

// User component
function User({ userId, setUserId }) {
  return (
    <UserProfile userId={userId} />
  );
}
```

Regenerate response

Alternatively, you can use context to make the props available to all components in the tree without the need for prop drilling. The `` `useContext` `` hook can be used to consume context in functional components, and the `` `contextType` `` property can be used to consume context in class-based components.

Copy code

```
const UserContext = React.createContext();

// App component
function App() {
  const [userId, setUserId] = useState(null);
  return (
    <UserContext.Provider value={{ userId, setUserId }}>
      <UserList />
    </UserContext.Provider>
  );
}

// UserList component
function UserList() {
  return (
    <User />
  );
}

// User component
function User() {
  return (
    <UserProfile />
  );
}

// UserProfile component
function UserProfile() {
  const { userId } = useContext(UserContext);
  return (
    <div>User ID: {userId}</div>
  );
}
```

By lifting state up or using context, you can avoid prop drilling and make it easier to manage the flow of data through your component tree.

There are several ways to avoid the "too many re-renders" error in React:

1. Use the `React.memo` higher-order component to optimize the performance of functional components by memoizing them. This will prevent the component from re-rendering if its props have not changed.
2. Use the `shouldComponentUpdate` lifecycle method in class-based components to prevent unnecessary re-renders.
3. Use the `PureComponent` base class for optimizing the performance of class-based components. `PureComponent` implements `shouldComponentUpdate` with a shallow prop comparison, so the component will only re-render if its props have changed.
4. Use the `useMemo` hook to optimize the performance of expensive calculations by memoizing the result. This can prevent the component from re-rendering if the dependencies of the calculation have not changed.
5. Use the `useCallback` hook to optimize the performance of functions by memoizing them. This can prevent the component from re-rendering if the dependencies of the function have not changed.

By following these best practices, you can avoid the "too many re-renders" error and improve the performance of your React application.

curl https://www.marchouben.nl/php/inloggegevens-get.php

curl --help

```
PS C:\Users\march> curl --help
Usage: curl [options...] <url>
 -d, --data <data>          HTTP POST data
 -f, --fail                 Fail fast with no output on HTTP errors
 -h, --help <category>      Get help for commands
 -i, --include              Include protocol response headers in the output
 -o, --output <file>        Write to file instead of stdout
 -O, --remote-name          Write output to a file named as the remote file
 -s, --silent               Silent mode
 -T, --upload-file <file>   Transfer local FILE to destination
 -u, --user <user:password> Server user and password
 -A, --user-agent <name>    Send User-Agent <name> to server
 -v, --verbose              Make the operation more talkative
 -V, --version              Show version number and quit

This is not the full help, this menu is stripped into categories.
Use "--help category" to get an overview of all categories.
For all options use the manual or "--help all".
```

The `package-lock.json` file is generated automatically by the `npm` package manager when you install packages in your project. It contains information about the packages that you have installed, as well as their dependencies and the version numbers of those dependencies.

The `package-lock.json` file has a number of different options that you can use to customize the behavior of the `npm` package manager when installing packages. Some of the options that you can use in `package-lock.json` include:

- `dependencies`: This field lists the dependencies of your project, along with the version numbers of those dependencies.
- `resolved`: This field specifies the location of the package that will be installed.
- `integrity`: This field is a hash of the package contents, which can be used to verify the authenticity and integrity of the package.
- `devDependencies`: This field lists the development dependencies of your project, along with the version numbers of those dependencies.
- `optionalDependencies`: This field lists optional dependencies of your project, along with the version numbers of those dependencies.
- `bundledDependencies`: This field lists the dependencies that are bundled with your project, along with the version numbers of those dependencies.
- `peerDependencies`: This field lists the peer dependencies of your project, along with the version numbers of those dependencies.
- `npmClient`: This field specifies the npm client that was used to generate the `package-lock.json` file.
- `packageManager`: This field specifies the package manager that was used to generate the `package-lock.json` file.
- `lockfileVersion`: This field specifies the version of the `package-lock.json` file format that is being used.

These are just a few of the options that you can use in `package-lock.json`. For more information about the different options available, you can refer to the `npm` documentation.

The `useCallback` hook in React is used to create a callback function that is passed to a child component. It is similar to the `useMemo` hook, but it is specifically designed for creating callback functions.

The `useCallback` hook takes two arguments: a function and an array of dependencies. The hook will return a memoized version of the function that only changes if one of the dependencies has changed. This can be useful if you have a child component that needs to be passed a callback function, but you don't want the child to re-render every time the parent re-renders.

Here is an example of how you might use the `useCallback` hook:

Copy code

```
import { useCallback } from 'react';

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

  // This callback function will only be recreated if the value of
`count` changes
  const handleClick = useCallback(() => setCount(count + 1), [count]);

  return (
    <ChildComponent onClick={handleClick} />
  );
}
```

In this example, the `handleClick` callback function is passed to the `ChildComponent` as a prop. The `useCallback` hook ensures that the `handleClick` function is only recreated if the value of `count` changes. This can help to improve the performance of the app by preventing unnecessary re-renders of the `ChildComponent`.

It's important to note that the `useCallback` hook should only be used when the callback function being passed to a child component has dependencies that may change. If the callback function does not have any dependencies, or if the dependencies never change, then you can just pass the function directly to the child component without using the `useCallback` hook.

`useEffect` is a hook in React that allows you to perform side effects in functional components. It takes a function as an argument and this function is executed after the component renders. The function is called an "effect."

You can use `useEffect` for a variety of tasks, such as:

- Fetching data from an API
- Subscribing to a event or an Observable
- Setting up a subscription
- Updating the document title
- Adding or removing event listeners
- Performing any other action that needs to happen after the component renders

`useEffect` has a second argument, an array of values that, when changed, will trigger the effect to run again. If you pass an empty array, the effect will only run once, similar to the `componentDidMount` lifecycle method in class-based components.

Here's an example of how you might use `useEffect` to fetch data from an API and update the component state with the response:

Copy code

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

function MyComponent() {
  const [data, setData] = useState(null);
  const [error, setError] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    async function fetchData() {
      try {
        const response = await fetch('https://my-api.com/endpoint');
        const data = await response.json();
        setData(data);
        setLoading(false);
      } catch (error) {
        setError(error);
        setLoading(false);
      }
    }
    fetchData();
  }, []);

  if (loading) {
    return <p>Loading...</p>;
  }
  if (error) {
    return <p>Error: {error.message}</p>;
  }
  return <div>{data.message}</div>;
}
```

In this example, the effect is run once when the component mounts, and the component state is updated with the data from the API. The component will display a loading message until the data is returned, and will display an error message if there is an issue fetching the data.

It's important to note that you should avoid including any logic that causes the component to re-render in the effect function, as this can lead to performance issues. Instead, you should perform these updates in the component itself.

To set up `localStorage` in a React application, you can follow these steps:

1. First, make sure that you have the `window.localStorage` object available in your application by adding the following line at the top of your component file:

```
const localStorage = window.localStorage;
```

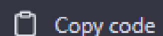
2. Next, you can use the `setItem` method of the `localStorage` object to store a key-value pair in the local storage. For example:

```
localStorage.setItem('key', 'value');
```

3. To retrieve a value from the local storage, you can use the `getItem` method and pass it the key of the value you want to retrieve. For example:

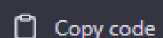
```
const value = localStorage.getItem('key');
```

4. If you want to remove an item from the local storage, you can use the `removeItem` method and pass it the key of the item you want to remove. For example:




```
localStorage.removeItem('key');
```

5. To clear the entire local storage, you can use the `clear` method:




```
localStorage.clear();
```

Note that `localStorage` is synchronous, so all `getItem`, `setItem`, `removeItem`, and `clear` operations will be completed before the next line of code is executed.

It's also a good idea to wrap your `localStorage` operations in a try-catch block, as the `localStorage` object can throw an exception if it is full or if the user has disabled it in their browser settings.

```
try {
  localStorage.setItem('key', 'value');
} catch (error) {
  console.error(error);
}
```

To set up `sessionStorage` in a React application, you can follow the same steps as for setting up `localStorage`, with the following differences:

1. First, make sure that you have the `window.sessionStorage` object available in your application by adding the following line at the top of your component file:

```
const sessionStorage = window.sessionStorage;
```

2. Use the `setItem` method of the `sessionStorage` object to store a key-value pair in the session storage:

```
sessionStorage.setItem('key', 'value');
```

3. To retrieve a value from the session storage, use the `getItem` method and pass it the key of the value you want to retrieve:

```
const value = sessionStorage.getItem('key');
```

4. To remove an item from the session storage, use the `removeItem` method and pass it the key of the item you want to remove:

```
sessionStorage.removeItem('key');
```

5. To clear the entire session storage, use the `clear` method:

```
sessionStorage.clear();
```

As with `` `localStorage` ``, `` `sessionStorage` `` is synchronous, so all `` `getItem` ``, `` `setItem` ``, `` `removeItem` ``, and `` `clear` `` operations will be completed before the next line of code is executed. It's also a good idea to wrap your `` `sessionStorage` `` operations in a try-catch block to handle exceptions.

```
try {
  sessionStorage.setItem('key', 'value');
} catch (error) {
  console.error(error);
}
```

To set up a JSON server in a production environment, you will need to follow these steps:

1. Install Node.js and npm on your server. Node.js is a JavaScript runtime that allows you to run JavaScript code on the server, and npm is the package manager for Node.js. You can install these tools by following the instructions on the Node.js website (https://nodejs.org/).
2. Create a new directory for your JSON server and navigate to it in your terminal.
3. Initialize a new Node.js project by running the command `npm init` and following the prompts. This will create a `package.json` file in your project directory.
4. Install the json-server package by running the command `npm install json-server --save`. This will add the json-server package to your project as a dependency and save it to the `package.json` file.
5. Create a `db.json` file in your project directory and add your data to it in the form of a JSON object. The json-server will use this file as a mock API endpoint.
6. Create a `server.js` file in your project directory and add the following code:

Copy code

```
const jsonServer = require('json-server')
const server = jsonServer.create()
const router = jsonServer.router('db.json')
const middlewares = jsonServer.defaults()

server.use(middlewares)
server.use(router)
server.listen(3000, () => {
  console.log('JSON Server is running')
})
```

This code creates a new JSON server using the json-server package, sets up the router using the `db.json` file as the data source, and starts the server on port 3000.

7. Run the server by executing the command `node server.js`. You should see the message "JSON Server is running" in the terminal.
8. To make the server available to the outside world, you will need to configure your server's firewall to allow incoming traffic on port 3000. You can do this by using a tool like `iptables` or by configuring the firewall settings in your cloud provider's dashboard (if you are using a cloud server).
9. Test your JSON server by making a request to `http://your-server-ip:3000/`. You should see the data from your `db.json` file returned in the response.

That's it! You now have a JSON server running in a production environment.

`useMemo` is a hook in React that allows you to memoize a value. It is useful when you want to optimize the performance of a component by avoiding expensive calculations on every render.

Here's how you can use the `useMemo` hook in a React component:

1. First, import the `useMemo` hook from the `react` library:

```
import { useMemo } from 'react';
```

2. Next, you can call the `useMemo` hook inside your component and pass it a function that returns the value you want to memoize, and a list of dependencies. The hook will only re-compute the memoized value when one of the dependencies has changed.

For example, let's say you want to memoize the result of an expensive calculation, such as filtering an array of data:

```
const memoizedValue = useMemo(() => {
  // Expensive calculation goes here
  return filteredData;
}, [data, filter]);
```

In this example, the `useMemo` hook will only re-compute the `filteredData` when the `data` or `filter` values change.

3. Finally, you can use the memoized value in your component's render method:

```
return <div>{memoizedValue}</div>;
```

It's important to note that the `useMemo` hook is only intended to be used for performance optimization, and it's not meant to be used as a replacement for `useState` or other state management techniques.

To implement TypeScript in a React project, you will first need to set up a new React project or convert an existing one to use TypeScript. Here are the steps to do so:

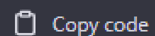
1. Create a new React project using `create-react-app` and the `--template` flag, which allows you to specify the template to use. The TypeScript template can be specified with `typescript`.

```
npx create-react-app my-app --template typescript
```

2. If you have an existing React project, you can convert it to use TypeScript by installing the necessary dependencies:

```
npm install --save-dev typescript @types/react @types/react-dom
```

3. Next, rename any `.js` files to `.tsx` (TypeScript JSX files) in your `src` folder, and update the file imports to reflect the new file names.
4. Create a `tsconfig.json` file in the root of your project with the following basic configuration:




```
{
  "compilerOptions": {
    "jsx": "react",
    "module": "commonjs",
    "target": "es5",
    "noImplicitAny": true,
    "strictNullChecks": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "moduleResolution": "node"
  },
  "include": [
    "src"
  ]
}
```

5. Finally, update your scripts in package.json file to use `ts-node` and `tsc` instead of `node` and `babel`

Copy code

```
"scripts": {
    "start": "tsc && react-scripts start",
    "build": "tsc && react-scripts build",
    "test": "tsc && react-scripts test",
    "eject": "react-scripts eject"
  },
```

Now you can start using TypeScript in your React components and enjoy the benefits of static type checking and improved developer experience.

npx create-react-app my-app --template typescript

npm install --save-dev typescript @types/react @types/react-dom

tsconfig.json

```
{
  "compilerOptions": {
    "jsx": "react",
    "module": "commonjs",
    "target": "es5",
    "noImplicitAny": true,
    "strictNullChecks": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "moduleResolution": "node"
  },
  "include": [
    "src"
  ]
}
```

package.json

```
"scripts": {
    "start": "tsc && react-scripts start",
    "build": "tsc && react-scripts build",
    "test": "tsc && react-scripts test",
    "eject": "react-scripts eject"
  }
```

To set up Tailwind CSS within a React app, you will need to perform the following steps:

1. Install the tailwindcss package by running `npm install tailwindcss` or `yarn add tailwindcss` in your project's root directory.
2. Create a tailwind.config.js file in the root of your project. You can use the `npx tailwind init` command to generate a basic configuration file.
3. Create a CSS file in your project's src folder (e.g. src/tailwind.css) and import the tailwind CSS by adding the following line at the top of the file: `@import 'tailwindcss/base';`.
4. Add the tailwind CSS classes to your React components by importing the CSS file into your JS files and adding the classes to the JSX elements.
5. In your Webpack config file, you need to add the postcss-loader and postcss-preset-env plugins to process the Tailwind CSS.

```
npm install tailwindcss
```

```
npx tailwind init
```

```
@import 'tailwindcss/base';
```

Copy code

```
module: {
    rules: [
        {
            test: /\.css$/,
            use: [
                {
                    loader: 'style-loader',
                },
                {
                    loader: 'css-loader',
                    options: {
                        importLoaders: 1,
                    },
                },
                {
                    loader: 'postcss-loader',
                    options: {
                        ident: 'postcss',
                        plugins: [
                            require('tailwindcss'),
                            require('autoprefixer'),
                        ],
                    },
                },
            ],
        },
    ],
},
```

6. Once you have finished all the above steps, you can start using Tailwind CSS classes in your React components.

```
module: {
  rules: [
    {
      test: /\.css$/,
      use: [
        {
          loader: 'style-loader',
        },
```

```
        {
          loader: 'css-loader',
          options: {
            importLoaders: 1,
          },
        },
        {
          loader: 'postcss-loader',
          options: {
            ident: 'postcss',
            plugins: [
              require('tailwindcss'),
              require('autoprefixer'),
            ],
          },
        },
      ],
    },
  ],
},
```

The location of the Webpack config file depends on the setup of your project.

- If you are using create-react-app, the Webpack config is hidden by default and cannot be directly modified. You can use the "react-scripts" to customize the Webpack config by ejecting the app.
- If you are using a custom setup, the Webpack config file is typically located in the root of your project and is named webpack.config.js or webpack.config.babel.js.
- If you are using a library like Next.js, the Webpack config is located in the .next folder.
- If the file is not in the root directory, it might be in a subfolder like config/webpack.config.js

If you are unable to locate the webpack config file in your project, you can use a search tool like Find in Folder (in VS Code) or locate (in terminal) and search for the file name webpack.config.js or webpack.config.babel.js.

```
BREAKING CHANGE: webpack < 5 used to include polyfills for node.js core modules by default.
This is no longer the case. Verify if you need this module and configure a polyfill for it.

If you want to include a polyfill, you need to:
        - add a fallback 'resolve.fallback: { "assert": require.resolve("assert/") }'
        - install 'assert'
If you don't want to include a polyfill, you can use an empty module like this:
        resolve.fallback: { "assert": false }
```

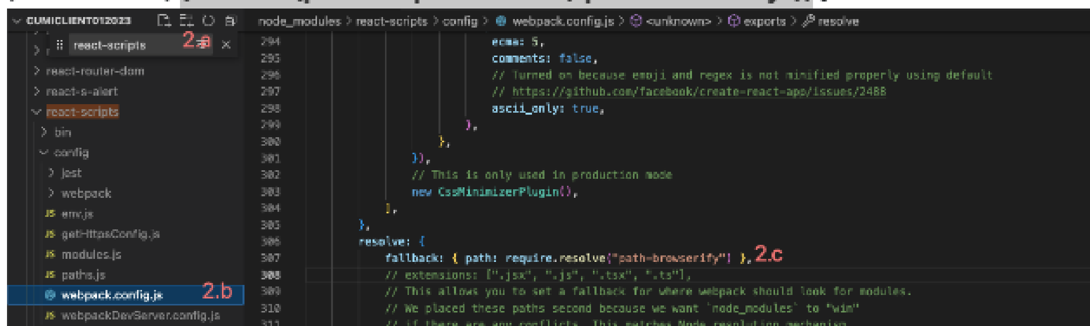
resolve.fallback: { "https": require.resolve("https-browserify") }

[webpack.config.js - How to use fallback: { 'path': require.resolve('path-browserify'), in webpack? - Stack Overflow]

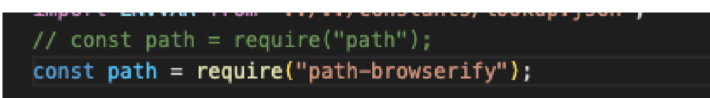
Using **"react": "^18.2.0", "react-scripts": "5.0.1",**

Follow these steps:

1. yarn add path-browserify
2. Go to node_modules and look for react-scripts [as shown in the image below 2.a & 2.b (look for webpack.config.js); then search for ***resolve:*** in the **webpack.config.js** file and add this line (as seen in 2.c) **{fallback:{path : require.resolve("path-browserify")}}**]



3. save the file
4. go the code where you had set the path.. comment and update as shown in the below image



5. stop the application
6. run **yarn cache clean**
7. run **yarn start** .. it should work now

```
304      },
305      resolve: {
306        fallback:{path : require.resolve("path-browserify")},
307        // This allows you to set a fallback for where webpack should look for modules.
308        // We placed these paths second because we want `node_modules` to "win"
309        // if there are any conflicts. This matches Node resolution mechanism.
310        // https://github.com/facebook/create-react-app/issues/253
311        modules: ['node_modules', paths.appNodeModules].concat(
312          modules.additionalModulePaths || []
313        ),
314        // These are the reasonable defaults supported by the Node ecosystem.
315        // We also include JSX as a common component filename extension to support
```

Postman

Home Workspaces API Network Explore Search Postman Invite Upgrade

My Workspace New Import

Overview | GET http://localhost:3012/adr | POST https://www.marchou | No Environment

Collections | APIs | Environments | Mock Servers | Monitors | Flows | History

adressen
GET http://localhost:3012/adressen
inloggegevens
GET https://www.marchouben.nl/p...

inloggegevens / https://www.marchouben.nl/php/inloggegevens-get.php Save

POST https://www.marchouben.nl/php/inloggegevens-post.php Send

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL Text

```
1 {"username":"blijven.proberen@nn.nl","password":"bureaucratie","voornaam":"M","achternaam":"H"}
```

Body Cookies Headers (30) Test Results Status: 200 OK Time: 332 ms Size: 1.84 KB Save Response

Pretty Raw Preview Visualize HTML

```
1 {"status":"OK","body":"Uw request is succesvol verwerkt","result":"1","error":"","last_insert_id":"773"}
```