Started web application using react, node.js, mongodb, express, tailwindcss  
  
npm create vite@latest client

Framework: React

Variant: JS+SWC

Here we will have all dependencies: npm i

Search for tailwind vite and start following steps from documentation

DO GIT COMMIT

05/04/2025

Npm I react-router-dom

Check how many pages are need while creating website, here we need home,about,sign-in,sign-up, then create router with these pages

Make sure to check router is connecting with all pages or not

DO GIT COMMIT

Next is to create component folder as we checked in original website we have header for every pages, so we create header component

So we add them in APP.jsx after browserroute

When added tailwind, if there is not happening css in browser responsive, you can check in google developer tool (responsive browser)

DO GIT COMMIT

Create backend folder name as “api”

Install express, nodemon

Add type:”module” as it is ES6 module we need to add while using import express from “express”-----

  "name": "api",

  "version": "1.0.0",

  "description": "",

  "main": "index.js",

  "type":"module”

Delete test text in scripts in package.json and add dev, start   
then only we add npm run dev

DO GIT COMMIT

As we created git in client we need to move to api also

mv .git ../

created mongodb database -project and cluster, added in env (mongourl)

installed dotenv and mongoose

DO GIT COMMIT

We need to create test api route in backend by creating folders of router, controller and add json file in controller whether it is working or not. Check as we are using ES module, we need to import

DO GIT COMMIT

When create auth router check all slashes,

While req.body from controller we need to make sure to use

app.use(express.json()); add in server.js

we need to save data in db so we deconstruction the body (postman) and save in user schema which we have created then we need to use “user.save()”, as we know for every computer there will be change of time in delay to save in database, so we use await then async

Next we need to check for password as we save directly in database we need to encrypt it using bcryptsjs and add

const hashedPassword = bcrypt.hashSync(password, 8); after deconstruction

to get error visible in postman we use try and catch method

res.status(500).json(error.message)

but above error message is correct but the way of keeping is different so we will next commit

DO GIT COMMIT

We create middleware of file error.middleware.js

export const errhander = (err, req, res, next) => {

    const statusCode = err.statusCode || 500;

    const message = err.message || 'Internal Server Error';

    res.status(statusCode).json({

        status: 'error',

        statusCode,

        message,

    });

}

And add app.use(errhander) in server.js

Change the arrow funcntion controller by adding next and next(error) in catch method(try/catch)

Here error is coming from json file ,

We can create our own error code and message by using Utils folder

export const errorHandler = (statusCode, message) => {

    const error = new Error();

    error.statusCode = statusCode;

    error.message = message;

    return error;

}

Then in controller we can we give our statuscode and message

next(errorHandler(550, "User signup failed"));

**✅ What You Did in utils/error.js:**

// utils/error.js

export const errorHandler = (statusCode, message) => {

const error = new Error();

error.statusCode = statusCode;

error.message = message;

return error;

};

This is **not** the error-handling middleware itself — this is just a **utility** to create consistent error objects you can throw or pass into next().

**✅ Using It in Your Controller:**

import { errorHandler } from '../utils/error.js';

export const signupController = (req, res, next) => {

try {

const { username, email, password } = req.body;

if (!username || !email || !password) {

return next(errorHandler(400, "All fields are required"));

}

// rest of logic...

res.status(201).json({ message: "Signed up!", user: { username, email } });

} catch (err) {

next(err);

}

};

This creates a clean, centralized way to throw custom errors. 💯

**✅ Then Combine With Your Middleware:**

// middleware/error.middleware.js

export const globalErrorHandler = (err, req, res, next) => {

const status = err.statusCode || 500;

const message = err.message || "Something went wrong";

res.status(status).json({

success: false,

message,

});

};

And use in server.js:

import { globalErrorHandler } from './middleware/error.middleware.js';

app.use(globalErrorHandler);

**🧠 Summary: You're Combining 2 Things**

| **File** | **Purpose** | **Example** |
| --- | --- | --- |
| utils/error.js | Creates a custom error object | next(errorHandler(400, "Bad input")) |
| middleware/error.middleware.js | Handles any error thrown or passed with next(err) | app.use(globalErrorHandler) |

This is **best practice**. You’re doing it right. ✅

DO GIT COMMIT

We have added bg-color in index.css as color of body are same for all pages

Here margin mean “m” and padding means “p”

My means margin of y-axis and mt means margin at top

For username,email and password we need to change type for similarily. We need to consider disabled option also

Create Link option where we click it navigates to signin page

DO GIT COMMIT

**✅ Corrected & Clearer Version:**

You're using the useState hook to manage form data in React:

const [formData, setFormData] = useState({});

Then, in your change handler:

const handleChange = (e) => {

setFormData({

...formData,

[e.target.id]: e.target.value,

});

};

**🔍 Explanation (cleaned up):**

* **useState**: The correct spelling is useState (camelCase, starting with a lowercase u).
* **setFormData**: This function updates the state.
* **Spread operator (...formData)**: Copies the existing form data to keep previous values.
* **Dynamic property [e.target.id]:** This updates only the input field that triggered the change.
* **e.target.value**: The new value for the changed input.

**📝 Cleaned-up version of your explanation:**

We're using the useState hook to manage the form data. Inside the handleChange function, we use the spread operator ...formData to preserve the existing values. Then we dynamically update the changed input field using [e.target.id]: e.target.value.

So after form data is created , we need to submit to backend, so we use localhost url in vite.config.js as all starts from /api.

**✅ What is setLoading(true) doing?**

When the user submits the form, you call:

setLoading(true);

This does **two things**:

1. **Disables the button** so users can’t click it again while the request is still processing.
2. **Shows a loading state** (i.e., button text becomes "loading..." instead of "Sign Up").

This prevents **double submissions** and improves UX by giving feedback that something is happening.

Then, after the request is done (whether success or fail), you do:

setLoading(false);

That resets the button back to normal so the user can try again or continue.

**❌ What about setError(...)?**

This is for **handling errors that come from your server (backend)**.

So inside:

if (data.success === false) {

setLoading(false);

setError(data.message);

return;

}

You're saying:

* "If the backend responded **with success: false** (e.g., username exists, email invalid, etc), show that error message on screen."

You are **not catching exceptions here** (like network errors) — you're just handling **expected errors** returned by the server.

**💥 What does the catch block handle?**

} catch (error) {

setLoading(false);

setError(error.message);

}

This is for **unexpected errors**, like:

* The server is down
* Network connection fails
* Your fetch() throws an error

So you need **both**:

* if (data.success === false) handles *expected* errors from the API
* catch (...) handles *unexpected* errors like network or JS bugs

**🔘 The Button**

<button disabled={loading}

className='bg-slate-700 text-white p-3 rounded-lg uppercase hover:opacity-95 disabled:opacity-80'>

{loading ? 'loading...' : 'Sign Up'}

</button>

Explanation:

* **disabled={loading}**: Disables the button while waiting for the server response.
* **Text shows "loading..."** when loading === true.
* **Styling change with disabled:opacity-80** dims the button to indicate it's inactive.

👉 This is why setLoading(true) is **needed immediately before your fetch**: so the UI knows to **show a spinner or loading text** and **disable interaction**.

**🤖 Your Middleware**

export const errMiddleware = (err, req, res, next) => {

const statusCode = err.statusCode || 500;

const message = err.message || 'Internal Server Error';

res.status(statusCode).json({

success: false,

statusCode,

message,

});

}

This ensures that **whenever an error happens in your backend**, the client receives a consistent response like:

{

"success": false,

"statusCode": 400,

"message": "Email already exists"

}

So on the frontend, your check:

if (data.success === false) {

setError(data.message);

}

makes perfect sense.

**TL;DR Summary**

| **Concept** | **Purpose** |
| --- | --- |
| setLoading(true) | Start loading UI, disable button |
| setLoading(false) | End loading UI |
| setError(...) | Show error message to user |
| if (data.success === false) | Handle backend's expected errors |
| catch(...) | Handle network/fetch-level errors |

DO GIT COMMIT

We installed jasonwebtoken as it generates token

We res.cookie where we create cookie inside response of cookie then send to json

Checked validuser from await User.findOne({email})

Then using error handler from utlis we check validuser and valid password

Token generation

const token = jwt.sign({id:validUser.\_id,},process.env.JWT\_secret)

as passoword showing in postman is risk so we used deconstructing

 const {password:pass, ...rest} = validUser.\_doc

where “.\_doc” is created already from database (refer to documents from cluster)

res.cookie(

            'access\_token',

            token,

            {httpOnly:true},

            {expiresIn: 3600}

        ).status(200).json(rest)

DO GIT COMMIT

<p>Dont have an account</p>

Keep in mind as we should not “’” like Don’t , it may occur errors

Here we copied whole page of signup and change signin accordingly.

DO GIT COMMIT

Redux — Global Centralized State

* With Redux, you manage auth.loading, auth.error, auth.user in one place (the Redux store).
* Any component (Navbar, Home, Dashboard) can access those values via useSelector.

✅ You can:

* Show a global loading bar in any route
* Display an error message anywhere
* Disable buttons or routes globally while a login is in progress

**When Redux Makes Sense**

Use Redux if:

* You need to **share auth state** across multiple components
* You want consistent global **loading/error** UI
* Your app has **user roles, permissions, dashboards**
* You're building a **medium to large app** with multiple pages

In smaller components like a sign-in form, useState is enough for managing loading and error. But for larger apps, especially with authentication, I use Redux to handle global state like user, loading, and error, so I can access them across different components and maintain consistent behavior.

We follow installation from redux website then add folder app/store.js

Add extra lines of middleware

 middleware: (getDefaultMiddleware) =>

    getDefaultMiddleware({

        serializableCheck:false

    })

Understanding Redux Middleware and `serializableCheck: false`

In the Redux Toolkit configuration you've shown, you're customizing the middleware setup with a specific option to disable serializability checks. Here's what it means:

**The Middleware Configuration**

```javascript

middleware: (getDefaultMiddleware) =>

getDefaultMiddleware({

serializableCheck: false

})

```

**Breakdown**:

1**. `getDefaultMiddleware**`:

- This is a function provided by Redux Toolkit that returns an array of default middleware that Redux Toolkit includes automatically.

- The default middleware includes important pieces like Redux Thunk (for async logic) and development checks.

2. **`serializableCheck: false`:**

- This disables one of the default middleware called the "serializable state invariant middleware".

- When enabled (default), this middleware checks that:

- All actions are plain objects with a `type` field

- State and actions are serializable (can be converted to JSON)

- This helps catch common mistakes and ensures compatibility with Redux's core principles

**Why Disable Serialization Checks?**

You might want to disable this check when:

- Your state or actions contain non-serializable values (like functions, Promises, or React components)

- You're using libraries that put non-serializable values in the Redux store

- For performance reasons in production (though the check is removed in production builds automatically)

**Trade-offs:**

- With **`serializableCheck: false`:**

- More flexibility in what you can put in the store

- Potential loss of some debugging benefits

- Possible issues with time-travel debugging and persistence

- **With default checks (enabled):**

- Better enforcement of Redux best practices

- Early detection of potential problems

- But might be too restrictive for some use cases

**Common Use Cases for Disabling:**

1. Storing non-serializable values like:

- Class instances

- Functions

- React components

- Promises

2. When using libraries like:

- Redux-Persist

- Connected React Router

- Other libraries that store non-serializable values in state

Remember that while disabling these checks gives you more flexibility, it's generally recommended to keep your Redux state serializable when possible to maintain all of Redux's benefits.

# Redux Toolkit Slice Explanation: `userSlice`

This code creates a Redux slice for managing user authentication state using Redux Toolkit's `createSlice` API. Here's a detailed breakdown:

## 1. Initial State

```javascript

const initialState = {

currentUser: null, // Stores the authenticated user data

loading: false, // Tracks if an auth operation is in progress

error: null // Stores any error that occurs during auth

}

```

## 2. Slice Creation

The `createSlice` function automatically generates action creators and action types:

```javascript

const userSlice = createSlice({

name: 'user', // Prefix for generated action types

initialState, // Initial state object

reducers: { // Defines reducer functions and auto-generates actions

signInStart: (state) => {

state.loading = true;

},

signInSuccess: (state, action) => {

state.currentUser = action.payload;

state.loading = false;

state.error = null;

},

signInFailure: (state, action) => {

state.error = action.payload;

state.loading = false;

}

}

})

```

## 3. Reducer Functions

### `signInStart`

- \*\*Purpose\*\*: Indicates authentication process has begun

- \*\*State Changes\*\*:

- Sets `loading` to `true`

- \*\*Typical Usage\*\*: Dispatched when login API call starts

### `signInSuccess`

- \*\*Purpose\*\*: Handles successful authentication

- \*\*Parameters\*\*:

- `action.payload`: Expected to contain user data

- \*\*State Changes\*\*:

- Sets `currentUser` to the payload (user data)

- Sets `loading` to `false`

- Clears any previous `error`

### `signInFailure`

- \*\*Purpose\*\*: Handles failed authentication

- \*\*Parameters\*\*:

- `action.payload`: Expected to contain error information

- \*\*State Changes\*\*:

- Sets `error` to the payload (error details)

- Sets `loading` to `false`

## 4. Exports

```javascript

export const { signInStart, signInSuccess, signInFailure } = userSlice.actions;

export default userSlice.reducer;

```

- \*\*Actions\*\*: The automatically generated action creators

- \*\*Reducer\*\*: The combined reducer function for this slice

## Typical Usage Flow

1. Dispatch `signInStart()` when beginning authentication

2. Make API call to authenticate user

3. On success:

- Dispatch `signInSuccess(userData)`

4. On failure:

- Dispatch `signInFailure(errorMessage)`

## Benefits of This Approach

- Automatic action type generation (e.g., `"user/signInStart"`)

- Immutable updates handled by Immer (notice direct state mutation)

- Clear separation of auth states (loading, success, error)

- Minimal boilerplate code

This pattern is commonly used with Redux Thunk or RTK Query for handling async operations like authentication.

**Destructuring `loading` and `error` from Redux State**

The line `const {loading, error} = useSelector((state) => state.user)` is using destructuring assignment to extract specific values from the Redux store's user state. Here's why this is important:

## Purpose of This Line

1. \*\*Accessing Redux State\*\*:

- `useSelector` is a React-Redux hook that extracts data from the Redux store state

- It takes a selector function `(state) => state.user` that:

- Receives the entire Redux state tree

- Returns just the `user` slice of state (from your userSlice)

2. \*\*Destructuring Assignment\*\*:

- The user slice has this structure (from your userSlice):

```javascript

{

currentUser: null,

loading: false,

error: null

}

```

- Destructuring `{loading, error}` extracts just these two properties

## Why We Do This

1. \*\*Component Needs Only Specific State\*\*:

- Your component only needs access to:

- `loading`: To show loading state/disable button

- `error`: To display any authentication errors

- It doesn't need `currentUser` in this component

2. \*\*Performance Optimization\*\*:

- The component will only re-render when either `loading` or `error` changes

- If we did `useSelector(state => state.user)`, the component would re-render for any change in the user state (including `currentUser` changes)

3. \*\*Cleaner Code\*\*:

- Direct access to the values needed without repeatedly typing `state.user.loading`

- Makes the JSX more readable when using these values

## Where These Values Are Used

1. \*\*`loading`\*\*:

```jsx

<button disabled={loading}>

{loading ? 'loading...' : 'Sign In'}

</button>

```

2. \*\*`error`\*\*:

```jsx

{error && <p className='text-red-500 mt-5'>{error}</p>}

```

## Alternative Without Destructuring

Without destructuring, you would write:

```javascript

const userState = useSelector((state) => state.user);

// Then use userState.loading and userState.error in your component

```

But this is less efficient (as explained above) and slightly more verbose to use in the component.

But Here we have problem when reload the page the credentials will remove automatically so for that we redux-presists as it saves file in local storage until browser is open it will not closed

DO GIT COMMIT

Installed redux persist and created files related to that

DO GIT COMMIT

For button in react , type was ‘submit’ in default and we don’t need like as we are doing google firebase wo we changed type as ‘button’

After installation of firebase

We get

const app = initializeApp(firebaseConfig);

Add export before as we connect with all componentns and make sure to add api key in .env as we are using vite

We use to get env from as import.meta.env.VITE\_FIREBASE\_API\_KEY

After adding google auth in oauth

export default function OAuth() {

    const handleGoogleClick = async() => {

        try {

           const provider =new GoogleAuthProvider()

           const auth = getAuth(app)

           const res = await signInWithPopup(auth,provider)

           console.log(res)

        } catch (error) {

            console.log("could not sign in with google",error)

        }

    }

We check in console by using google developer tool, where we can see displayname,email and photourl(as we save our image in gmail)



Next we use that data and save in globally where we use redux

For this reason we need add backend /google

const res = await fetch('/api/auth/google',{

                method: 'POST',

                headers: {

                    'Content-Type': 'application/json',

                },

                body: JSON.stringify({

                    name:result.user.displayName,

                    email:result.user.email,

                    photo:result.user.photoURL,

                })

            })

            const data = await res.json()

            dispatch(signInSucess(data))

for backend as we are taking photourl so we need to add in models and while we are using google we did not provide password as it takes directly but in mangoose we provided that password is required , so we create an ordinary password so that we should not get an error from mongodb

so we create a copy image address of image

while creating username as from firebase it will create like “Konakalla Chaitanya” as there was a space so to remove them and add some random number we use

username:req.body.name.split(' ').join('').toLowerCase()+Math.random().toString(36).slice(-8),

DO GIT COMMIT

DOUBTS:

** Add a "remember me" checkbox that stores email in localStorage ?**

** Add a loading spinner while authenticating ?**

We start creating profile page as per website instructions, we need to show avatar image when it is authenticated and when it is not we show normal header with ‘signin’

1. **ProtectedRoute.jsx checks currentUser**
   * If logged in (currentUser exists) → Render <Outlet />
   * If not logged in → Redirect to /sign-in
2. **Outlet renders a "specific component"**
   * It renders the **matched child route's component** from the parent <Route> in your router config.

**Outlet renders the component of the *matched child route* defined inside the parent <Route>.**

* When you visit /profile:
  + **Parent route (ProtectedRoute)** renders first.
  + If currentUser exists → <Outlet /> looks for **which child route matches /profile**.
  + Finds <Profile /> and injects it into the Outlet spot.

DO GIT COMMIT

Added profile UI make sure button is also inside form as we submitted it

DO GIT COMMIT

We create image functionality as we upload image we will get as seconds and image uploading text in browser

<img onClick={()=>fileRef.current.click()}

The arrow function () => acts like saying:  
"Don't do this now - remember to do it later when I actually click."

**How This Works Step-by-Step**

**1. useRef(null) Creates a "Box"**

* Think of fileRef as an empty box that can hold something
* Initially, it holds { current: null }
* This box persists between re-renders

**2. The ref={fileRef} Attachment**

* When React creates the actual <input> element in the DOM:
  + It puts a reference to that real DOM element into your box
  + Now fileRef.current = the actual HTML input element

**3. Why We Need .current**

* fileRef is always the box { current: ... }
* fileRef.current gives us what's inside the box (the DOM element)
* We need .current to access the actual input element

**Timeline of What Happens**

| **Time** | **What Happens** | **fileRef Contents** |
| --- | --- | --- |
| Initial Render | useRef(null) creates box | { current: null } |
| After DOM Created | React puts input element in box | { current: <inputElement> } |
| When Image Clicked | We access fileRef.current | Gets the real input element |

**Why Not Just fileRef.click()?**

JavaScript needs us to be explicit:

* fileRef is the container (always an object with current)
* fileRef.current is the actual DOM element
* DOM elements have .click() method, but the container doesn't

**Real-World Analogy: TV Remote**

* fileRef = The remote control
* fileRef.current = The actual TV it controls
* You can't press buttons on the remote itself (fileRef.click() won't work)
* You need to press buttons on the TV (fileRef.current.click())

**Key Points to Remember**

1. useRef gives you a persistent box ({ current: value })
2. React automatically puts the DOM element in .current
3. Always use .current to access the real DOM element
4. The arrow function ensures we access .current at the right time

So after updating image , change occurs must done in firebase

So we need **storage from firebase**