Getting Started with Vite + React

This guide covers the fundamentals of setting up a Vite + React project, using Tailwind CSS for modern styling, React Router for navigation, and creating reusable components with advanced features.

Table of Contents

Setting Up a Vite + React Project

What is Vite?

Vite is a modern build tool that provides a faster and leaner development experience for modern web projects. It offers instant server start, lightning-fast Hot Module Replacement (HMR), and optimized builds.

Creating a New Vite + React Project

1. **Create the project** using npm:

   ```
   npm create vite@latest my-project-name -- --template react
   ```

   Replace `my-project-name` with your desired project name.

2. **Navigate to the project directory**:

   ```
   cd my-project-name
   ```

3. **Install dependencies**:

   ```
   npm install
   ```

4. **Start the development server**:

   ```
   npm run dev
   ```

Project Structure

After creation and setup, your project will have the following structure:

```
countdown-timer-activity/
├── node_modules/
├── public/
├── src/
│   ├── assets/
│   │   ├── pause-icon.svg
│   │   ├── restart-icon.svg
│   │   ├── play-icon.svg
│   │   └── reset-icon.svg
│   ├── components/
│   │   ├── TimeInput.jsx
│   │   ├── TimeDisplay.jsx
│   │   ├── Timer.jsx
│   │   └── SetTimerPage.jsx
│   ├── App.css
│   ├── App.jsx
│   ├── index.css
│   └── main.jsx
├── index.html
├── package.json
├── postcss.config.js
├── tailwind.config.js
└── vite.config.js
```

Key directories and files:

- `src/assets/` - SVG icons and static assets
- `src/components/` - Reusable React components
- `tailwind.config.js` - Tailwind CSS configuration
- `postcss.config.js` - PostCSS configuration for Tailwind

Setting Up Tailwind CSS

What is Tailwind CSS?

Tailwind CSS is a utility-first CSS framework that allows you to build modern, responsive designs using pre-defined classes directly in your markup. Instead of writing custom CSS, you compose designs using utility classes like `flex`, `text-center`, `bg-blue-500`, etc.

Installation

1. **Install Tailwind CSS and its peer dependencies:**

```
npm install -D tailwindcss@4.1.17 postcss autoprefixer
```

2. **Create Tailwind configuration file:**

```
npx tailwindcss init -p
```

This creates two files:

- `tailwind.config.js` - Tailwind configuration
- `postcss.config.js` - PostCSS configuration

3. **Configure your template paths** in `tailwind.config.js`:

```
/** @type {import('tailwindcss').Config} */
export default {
  content: [
    "./index.html",
    "./src/**/*.{js,ts,jsx,tsx}",
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

4. **Add Tailwind directives to your CSS** in `src/index.css`:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

5. **Start using Tailwind classes** in your components:

```
function MyComponent() {
  return (
    <div className="flex items-center justify-center h-screen bg-gray-900">
      <h1 className="text-4xl font-bold text-blue-500">Hello Tailwind!</h1>
    </div>
  )
}
```

Benefits of Using Tailwind CSS

- **Rapid Development**: Build UIs faster with utility classes
- **Consistent Design**: Pre-defined spacing, colors, and sizing scales

- **No CSS File Management**: Styles are co-located with components
- **Tree-Shaking**: Unused styles are automatically removed in production
- **Responsive Design**: Built-in responsive modifiers (sm:, md:, lg:, xl:)
- **Dark Mode Support**: Easy dark mode implementation

Application in This Project

This countdown timer project uses Tailwind CSS extensively for:

- Glassmorphic container designs with backdrop blur
- Golden glow effects on the timer display (#fdc700)
- Button styling with hover states
- Responsive layouts with flexbox and grid
- Custom gradients and shadows
- Consistent spacing and typography

Using React Router DOM (v5.2.0)

What is React Router?

React Router DOM is a library that enables navigation between different views/pages in a React application. It allows you to create single-page applications with multiple routes.

Installation

Install React Router DOM v5.2.0:

```
npm install react-router-dom@5.2.0
```

Core Components

1. **BrowserRouter**: Wraps your entire application to enable routing
2. **Route**: Defines a path and the component to render
3. **Switch**: Renders only the first matching route
4. **Link**: Creates navigation links without page reload

Basic Setup

Step 1: Wrap Your App with BrowserRouter

In `main.jsx`, import and wrap your App component:

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import { BrowserRouter } from 'react-router-dom'
import './index.css'
import App from './App.jsx'
```

```
createRoot(document.getElementById('root')).render(
  <StrictMode>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </StrictMode>,
)
```

Step 2: Define Routes in Your App

In `App.jsx`, import routing components and define your routes. For this countdown timer project, we use a two-route system with the Timer component on the main page and SetTimerPage for configuration.

Application in This Project

In this countdown timer project, routing is set up with a **two-route system** for better user experience:

```
// App.jsx
<Switch>
  <Route exact path="/">
    <Timer
      hours={hours}
      minutes={minutes}
      seconds={seconds}
      inputHours={inputHours}
      inputMinutes={inputMinutes}
      inputSeconds={inputSeconds}
      isRunning={isRunning}
      timerCompleted={timerCompleted}
      justRestarted={justRestarted}
      setHours={setHours}
      setMinutes={setMinutes}
      setSeconds={setSeconds}
      setInputHours={setInputHours}
      setInputMinutes={setInputMinutes}
      setInputSeconds={setInputSeconds}
      setIsRunning={setIsRunning}
      setTimerCompleted={setTimerCompleted}
      setJustRestarted={setJustRestarted}
    />
  </Route>
  <Route path="/set-timer">
    <SetTimerPage
      inputHours={inputHours}
      inputMinutes={inputMinutes}
      inputSeconds={inputSeconds}
      setInputHours={setInputHours}
      setInputMinutes={setInputMinutes}
      setInputSeconds={setInputSeconds}
      setHours={setHours}
```

```
        setMinutes={setMinutes}
        setSeconds={setSeconds}
        setIsRunning={setIsRunning}
        setTimerCompleted={setTimerCompleted}
        setJustRestarted={setJustRestarted}
        isRunning={isRunning}
      />
    </Route>
  </Switch>
```

Why use two routes for a timer?

1. **Separation of Concerns**: The main timer page (`/`) focuses on displaying the countdown and controls, while the set timer page (`/set-timer`) handles configuration
2. **Prevents Accidental Changes**: Users can't accidentally modify timer values while the countdown is running
3. **Cleaner UI**: Each page has a focused purpose without cluttering the interface
4. **Better User Flow**: Clear workflow: Set → Start → Monitor → Restart or Set Again
5. **State Persistence**: All state is managed in App.jsx, so navigating between routes doesn't lose data
6. **Scalability**: Easy to add more routes in the future (e.g., timer history, settings, presets)

Navigation Pattern:

- Click "Set Timer" button on main page → navigates to `/set-timer`
- Click "Start Timer" on set timer page → starts countdown and navigates back to `/`
- Click "Go Back" on set timer page → returns to `/` without saving changes

---

Creating Reusable Components

What are Reusable Components?

Reusable components are self-contained pieces of UI that can be used multiple times throughout your application with different data. They follow the DRY (Don't Repeat Yourself) principle and make your code more maintainable.

Key Principles

1. **Single Responsibility**: Each component should do one thing well
2. **Props for Customization**: Use props to make components flexible
3. **Separation of Concerns**: Keep logic, styling, and presentation organized

Example 1: TimeInput Component

This component creates a reusable input field for time values with advanced features:

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

function TimeInput({ label, value, onChange, max, disabled }) {
  const [isTyping, setIsTyping] = useState(false);
```

```
  const handleChange = (e) => {
    const inputValue = parseInt(e.target.value) || 0;
    // Ensure value doesn't exceed max
    if (inputValue >= 0 && inputValue <= max) {
      onChange(inputValue);
    }
  };

  const handleClick = (e) => {
    // Auto-select all text on click for easy replacement
    e.target.select();
    setIsTyping(true);
  };

  const handleBlur = () => {
    setIsTyping(false);
  };

  const handleKeyDown = (e) => {
    // Auto-select on number key press if not already typing
    if (!isTyping && /^[0-9]$/.test(e.key)) {
      e.target.select();
      setIsTyping(true);
    }
  };

  return (
    <div className="flex flex-col items-center gap-2">
      <label className="text-sm font-semibold text-gray-300">
        {label}
      </label>
      <input
        type="number"
        min="0"
        max={max}
        value={value}
        onChange={handleChange}
        onClick={handleClick}
        onBlur={handleBlur}
        onKeyDown={handleKeyDown}
        disabled={disabled}
        className="w-20 px-4 py-2 text-center text-xl font-bold bg-gray-800
                   border-2 border-gray-700 rounded-lg text-white
                   focus:outline-none focus:border-blue-500 focus:ring-2
                   focus:ring-blue-500/50 transition-all
                   disabled:opacity-50 disabled:cursor-not-allowed"
      />
    </div>
  );
}

export default TimeInput;
```

Usage:

```
<TimeInput
  label="Hours"
  value={hours}
  onChange={setHours}
  max={99}
/>
<TimeInput
  label="Minutes"
  value={minutes}
  onChange={setMinutes}
  max={59}
  disabled={isRunning}  // Optional: disable during countdown
/>
```

Advanced Features:

- **Smart Text Selection**: Automatically selects all text on click for easy replacement
- **Typing State Tracking**: Monitors whether user is actively typing
- **Keyboard Shortcuts**: Auto-selects on number key press
- **Validation**: Enforces min (0) and max (configurable) values
- **Tailwind Styling**: Modern dark theme with focus states
- **Disabled State**: Can be disabled via prop with visual feedback
- **Accessibility**: Proper labels and focus management

Why it's reusable:

- Used for hours (0-99), minutes (0-59), and seconds (0-59) with different max values
- Accepts custom labels via props
- Handles its own validation and UX logic
- Styled with Tailwind for consistent appearance
- Can be used in any project that needs number inputs

Example 2: TimeDisplay Component

This component displays time in a formatted way with modern glassmorphic design:

```
import React from 'react';

function TimeDisplay({ hours, minutes, seconds }) {
  // Format numbers to always show 2 digits
  const formatTime = (time) => {
    return time.toString().padStart(2, '0');
  };

  return (
    <div
```

```
        className="relative p-8 rounded-3xl"
        style={{
          background: `
            radial-gradient(circle at 30% 30%,
              rgba(253, 199, 0, 0.15) 0%,
              transparent 50%),
            radial-gradient(circle at 70% 70%,
              rgba(100, 108, 255, 0.15) 0%,
              transparent 50%),
            rgba(17, 17, 17, 0.6)
          `,
          backdropFilter: 'blur(10px)',
          boxShadow: `
            inset 0 1px 2px rgba(255, 255, 255, 0.1),
            inset 0 -1px 2px rgba(0, 0, 0, 0.5),
            0 10px 30px rgba(0, 0, 0, 0.5)
          `
        }}
      >
        <div className="flex flex-col items-center">
          {/* Timer display */}
          <div
            className="text-8xl font-bold tracking-wider"
            style={{
              fontFamily: 'Consolas, Monaco, "Courier New", monospace',
              color: '#fdc700',
              textShadow: `
                0 0 10px rgba(253, 199, 0, 0.8),
                0 0 20px rgba(253, 199, 0, 0.6),
                0 0 30px rgba(253, 199, 0, 0.4),
                2px 2px 4px rgba(0, 0, 0, 0.8)
              `
            }}
          >
            {formatTime(hours)}:{formatTime(minutes)}:{formatTime(seconds)}
          </div>

          {/* Time unit labels */}
          <div className="flex gap-16 mt-4 text-sm text-gray-400 uppercase tracking-
widest">
            <span className="w-20 text-center">hours</span>
            <span className="w-20 text-center">minutes</span>
            <span className="w-20 text-center">seconds</span>
          </div>
        </div>
      </div>
    );
}

export default TimeDisplay;
```

Usage:

```
<TimeDisplay hours={2} minutes={30} seconds={45} />
// Displays: 02:30:45 with golden glow effect
```

Advanced Features:

- **Glassmorphic Design**: Semi-transparent background with backdrop blur for modern aesthetic
- **Golden Glow Effect**: Multiple text shadows create a luminous golden (#fdc700) appearance
- **Radial Gradients**: Subtle color overlays add depth and visual interest
- **Inset Shadows**: Create a 3D recessed effect for the container
- **Monospace Font**: Consolas/Monaco ensures consistent digit width
- **Unit Labels**: Displays "hours", "minutes", "seconds" below the time
- **Responsive Layout**: Flexbox ensures proper alignment
- **Zero-Padding**: Always shows 2 digits (e.g., 05:09:03)

Why it's reusable:

- Takes any time values as props
- Handles formatting internally
- Professional, consistent styling
- Can be reused for stopwatches, clocks, timers, etc.
- Visually striking without being distracting

Example 3: Timer Component (Composition & Control Logic)

The Timer component demonstrates composition by combining reusable components with countdown logic and navigation:

```
import { useEffect, useRef } from 'react';
import { useHistory } from 'react-router-dom';
import TimeDisplay from './TimeDisplay';
import pauseIcon from '../assets/pause-icon.svg';
import restartIcon from '../assets/restart-icon.svg';

function Timer({
  hours, minutes, seconds,
  inputHours, inputMinutes, inputSeconds,
  isRunning, timerCompleted, justRestarted,
  setHours, setMinutes, setSeconds,
  setIsRunning, setTimerCompleted, setJustRestarted
}) {
  const intervalRef = useRef(null);
  const history = useHistory();

  // Countdown logic with useEffect
  useEffect(() => {
    if (isRunning && !timerCompleted) {
      intervalRef.current = setInterval(() => {
        // Check if timer reached 00:00:00
        if (hours === 0 && minutes === 0 && seconds === 0) {
```

```
            clearInterval(intervalRef.current);
            setIsRunning(false);
            setTimerCompleted(true);
            alert('Time is up!');
          } else {
            // Decrement time
            if (seconds > 0) {
              setSeconds(seconds - 1);
            } else if (minutes > 0) {
              setMinutes(minutes - 1);
              setSeconds(59);
            } else if (hours > 0) {
              setHours(hours - 1);
              setMinutes(59);
              setSeconds(59);
            }
          }
        }, 1000);
      }

      // Cleanup on unmount or when timer stops
      return () => clearInterval(intervalRef.current);
    }, [isRunning, hours, minutes, seconds, timerCompleted]);

    // Smart button label logic
    const getButtonLabel = () => {
      if (justRestarted || (hours === 0 && minutes === 0 && seconds === 0 &&
!isRunning)) {
        return 'Start';
      }
      return isRunning ? 'Pause' : 'Resume';
    };

    // Button handlers
    const handleSetTimer = () => {
      history.push('/set-timer');
    };

    const handleToggleTimer = () => {
      if (justRestarted) setJustRestarted(false);
      setIsRunning(!isRunning);
    };

    const handleRestart = () => {
      setHours(inputHours);
      setMinutes(inputMinutes);
      setSeconds(inputSeconds);
      setIsRunning(false);
      setTimerCompleted(false);
      setJustRestarted(true);
    };

    // Check if restart button should be enabled
    const isRestartEnabled =
```

```
    hours !== inputHours ||
    minutes !== inputMinutes ||
    seconds !== inputSeconds;

  return (
    <div className="flex flex-col items-center gap-8 p-8">
      <h1 className="text-4xl font-bold text-white">Countdown Timer</h1>

      {/* TimeDisplay component */}
      <TimeDisplay hours={hours} minutes={minutes} seconds={seconds} />

      {/* Control buttons in grid layout */}
      <div className="grid grid-cols-3 gap-4 w-full max-w-md">
        {/* Set Timer button */}
        <button
          onClick={handleSetTimer}
          className="px-6 py-3 bg-yellow-500 hover:bg-yellow-600 text-black
                     font-semibold rounded-lg transition-colors"
        >
          Set Timer
        </button>

        {/* Pause/Resume/Start button */}
        <button
          onClick={handleToggleTimer}
          disabled={timerCompleted || (inputHours === 0 && inputMinutes === 0 &&
inputSeconds === 0)}
          className="px-6 py-3 bg-gray-700 hover:bg-gray-600 text-white
                     font-semibold rounded-lg transition-colors flex items-center
                     justify-center gap-2 disabled:opacity-50 disabled:cursor-not-
allowed"
        >
          {isRunning && <img src={pauseIcon} alt="Pause" className="w-5 h-5" />}
          {getButtonLabel()}
        </button>

        {/* Restart button */}
        <button
          onClick={handleRestart}
          disabled={!isRestartEnabled}
          className="px-6 py-3 bg-gray-700 hover:bg-gray-600 text-white
                     font-semibold rounded-lg transition-colors flex items-center
                     justify-center gap-2 disabled:opacity-50 disabled:cursor-not-
allowed"
        >
          <img src={restartIcon} alt="Restart" className="w-5 h-5" />
          Restart
        </button>
      </div>
    </div>
  );
}

export default Timer;
```

Key Features:

1. **Countdown Logic with useEffect**:
   - Runs interval every second when `isRunning` is true
   - Hierarchical decrement: seconds → minutes → hours
   - Auto-stops and alerts when reaching 00:00:00
   - Proper cleanup on unmount
2. **useRef for Interval Storage**:
   - Stores interval ID without causing re-renders
   - Enables cleanup in useEffect return function
3. **Three Control Buttons**:
   - **Set Timer** (yellow): Navigates to `/set-timer` route
   - **Pause/Resume/Start** (gray with icon): Dynamic label based on timer state
   - **Restart** (gray with icon): Resets to input values without auto-starting
4. **Smart Button States**:
   - Pause/Resume button shows "Start" after restart or when starting fresh
   - Pause/Resume disabled when timer completes or no input values set
   - Restart only enabled when current time differs from input values
5. **Navigation with useHistory**:
   - Imports `useHistory` from react-router-dom
   - Uses `history.push('/set-timer')` to navigate to Set Timer page
6. **SVG Icons**:
   - Pause icon displayed when timer is running
   - Restart icon displayed on restart button
   - Icons imported from assets folder
7. **Component Composition**:
   - Uses TimeDisplay component for countdown display
   - All state managed in parent App.jsx and passed as props
   - Separates display from logic for better organization

Example 4: SetTimerPage Component (Form Pattern & Navigation)

This component provides a dedicated page for configuring timer values with local state management and confirmation dialogs:

```
import { useState } from 'react';
import { useHistory } from 'react-router-dom';
```

```
import TimeInput from './TimeInput';

function SetTimerPage({
  inputHours, inputMinutes, inputSeconds,
  setInputHours, setInputMinutes, setInputSeconds,
  setHours, setMinutes, setSeconds,
  setIsRunning, setTimerCompleted, setJustRestarted,
  isRunning
}) {
  // Local state for form inputs (only updates App state on "Start Timer")
  const [localHours, setLocalHours] = useState(inputHours);
  const [localMinutes, setLocalMinutes] = useState(inputMinutes);
  const [localSeconds, setLocalSeconds] = useState(inputSeconds);

  const history = useHistory();

  const handleStartTimer = () => {
    // Confirmation dialog if overwriting a running timer
    if (isRunning) {
      const confirmed = window.confirm(
        'Starting a new timer will stop the current one. Continue?'
      );
      if (!confirmed) return;
    }

    // Update App state with local values
    setInputHours(localHours);
    setInputMinutes(localMinutes);
    setInputSeconds(localSeconds);
    setHours(localHours);
    setMinutes(localMinutes);
    setSeconds(localSeconds);

    // Start the timer and navigate back
    setIsRunning(true);
    setTimerCompleted(false);
    setJustRestarted(false);
    history.push('/');
  };

  const handleGoBack = () => {
    // Navigate back without saving changes
    history.push('/');
  };

  return (
    <div className="flex flex-col items-center justify-center min-h-screen p-8">
      <h1 className="text-4xl font-bold text-white mb-8">Set Timer</h1>

      {/* Time input fields */}
      <div className="flex gap-8 mb-8">
        <TimeInput
          label="Hours"
          value={localHours}
```

```
            onChange={setLocalHours}
            max={99}
          />
          <TimeInput
            label="Minutes"
            value={localMinutes}
            onChange={setLocalMinutes}
            max={59}
          />
          <TimeInput
            label="Seconds"
            value={localSeconds}
            onChange={setLocalSeconds}
            max={59}
          />
        </div>

        {/* Action buttons */}
        <div className="flex gap-4">
          <button
            onClick={handleStartTimer}
            className="px-8 py-3 bg-yellow-500 hover:bg-yellow-600 text-black
                       font-semibold rounded-lg transition-colors"
          >
            Start Timer
          </button>
          <button
            onClick={handleGoBack}
            className="px-8 py-3 bg-gray-700 hover:bg-gray-600 text-white
                       font-semibold rounded-lg transition-colors"
          >
            Go Back
          </button>
        </div>
      </div>
    );
}

export default SetTimerPage;
```

Key Features:

1. **Local State Pattern**:
   - Uses separate `localHours`, `localMinutes`, `localSeconds` state
   - Only updates parent App state when "Start Timer" is clicked
   - Allows users to cancel changes by clicking "Go Back"
   - Prevents accidental modifications to the running timer
2. **Confirmation Dialog**:
   - Checks if timer is currently running (`isRunning` prop)

   - Shows confirm dialog: "Starting a new timer will stop the current one. Continue?"
   - Returns early if user cancels, preserving current timer
   - Prevents accidental timer overwrites

3. **Two Action Buttons**:

   - **Start Timer** (yellow): Saves values, starts countdown, navigates to `/`
   - **Go Back** (gray): Returns to main page without saving changes

4. **Navigation with useHistory**:

   - Imports `useHistory` hook from react-router-dom
   - Uses `history.push('/')` to navigate back to main timer page
   - Programmatic navigation after user actions

5. **Component Reuse**:

   - Uses TimeInput component three times
   - Hours range: 0-99 (extended from typical 0-23)
   - Minutes and seconds: 0-59
   - Consistent styling and validation

6. **State Synchronization**:

   - Updates both input state (for restart) and active state (for countdown)
   - Sets `isRunning` to true and `timerCompleted` to false
   - Resets `justRestarted` flag
   - Ensures all flags are properly synchronized

Why This Pattern?

- **User Protection**: Prevents accidental overwrites with confirmation
- **Form Flexibility**: Users can experiment with values before committing
- **Cancel Support**: "Go Back" abandons changes, maintaining current timer
- **Clear Intent**: Explicit "Start Timer" action shows user's commitment
- **Separation of Concerns**: Configuration UI separate from monitoring UI

Using SVG Icons as Assets

This project uses SVG icons to enhance the visual communication of button actions. SVG (Scalable Vector Graphics) files are ideal for icons because they:

- Scale perfectly at any size without pixelation
- Have small file sizes
- Can be styled with CSS
- Support transparency

Available Icons in `src/assets/`:

1. **pause-icon.svg** - Displayed on Pause button when timer is running
2. **restart-icon.svg** - Displayed on Restart button

How to Import and Use SVG Icons:

```
import pauseIcon from '../assets/pause-icon.svg';
import restartIcon from '../assets/restart-icon.svg';

function MyComponent() {
  return (
    <button className="flex items-center gap-2">
      <img src={pauseIcon} alt="Pause" className="w-5 h-5" />
      Pause
    </button>
  );
}
```

Key Points:

- Import SVG files like any other asset
- Use `<img>` tag with imported path as `src`
- Always provide descriptive `alt` text for accessibility
- Control size with Tailwind classes (`w-5 h-5` = 20px × 20px)
- Combine with text using flexbox (`flex items-center gap-2`)

Creating Custom SVG Icons:

You can create or download SVG icons from:

- Heroicons - Free MIT-licensed icons
- Lucide Icons - Beautiful, consistent icons
- Iconoir - Open source icon library
- Adobe Illustrator or Figma for custom designs

Simply save the SVG file to `src/assets/` and import it in your components.

Best Practices for Reusable Components

1. **Use Props for Data**: Pass data to components via props rather than hardcoding values

   ```
   // Good
   <Button text="Click Me" onClick={handleClick} />

   // Bad
   <Button /> // with hardcoded "Click Me" inside
   ```

2. **Keep Components Small**: Each component should have one clear purpose

   ```
   // Good: Separate components
   <TimeInput />
   <TimeDisplay />
   ```

```
// Bad: One giant component doing everything
<TimeInputAndDisplay />
```

3. **Provide Sensible Defaults**: Use default props when appropriate

```
function Button({ text = "Submit", type = "button" }) {
  return <button type={type}>{text}</button>
}
```

4. **Name Props Clearly**: Use descriptive prop names

```
// Good
<TimeInput label="Hours" max={23} />

// Bad
<TimeInput l="Hours" m={23} />
```

5. **Extract Repeated UI**: If you're copying and pasting JSX, make it a component

```
// Before: Repeated code
<div className="card">
  <h2>{title1}</h2>
  <p>{content1}</p>
</div>
<div className="card">
  <h2>{title2}</h2>
  <p>{content2}</p>
</div>

// After: Reusable component
function Card({ title, content }) {
  return (
    <div className="card">
      <h2>{title}</h2>
      <p>{content}</p>
    </div>
  )
}

<Card title={title1} content={content1} />
<Card title={title2} content={content2} />
```

Running and Building Your Project

Development Mode

Start the development server with hot module replacement:

```
npm run dev
```

The application will typically run at `http://localhost:5173/`

Features in dev mode:

- Instant updates when you save files
- Detailed error messages
- Fast refresh without losing component state

Production Build

Create an optimized production build:

```
npm run build
```

This command:

- Bundles your code
- Minifies JavaScript and CSS
- Optimizes assets
- Creates a `dist` folder with production files

Preview Production Build

Test the production build locally:

```
npm run preview
```

Project Scripts Summary

Command	Purpose
`npm run dev`	Start development server
`npm run build`	Create production build
`npm run preview`	Preview production build locally
`npm run lint`	Run ESLint to check code quality

Summary

You've learned how to:

1. **Set up a Vite + React project** for fast development with instant HMR
2. **Install and configure Tailwind CSS** for utility-first styling
3. **Use React Router DOM v5.2.0** to create multi-page applications with client-side routing and navigation
4. **Create reusable components** that follow best practices:
    - TimeInput with smart text selection and validation
    - TimeDisplay with glassmorphic design and golden glow effects
    - Timer with countdown logic, control buttons, and navigation
    - SetTimerPage with local state management and confirmation dialogs
5. **Use SVG icons as assets** for enhanced visual communication
6. **Implement advanced features** like confirmation dialogs, smart button states, and protected timer setting
7. **Run and build your project** for development and production

Key Technologies & Packages

Production Dependencies:

- `react` (^18.3.1) - UI library
- `react-dom` (^18.3.1) - React rendering for web
- `react-router-dom` (5.2.0) - Client-side routing

Development Dependencies:

- `vite` (^7.2.2) - Build tool and dev server
- `tailwindcss` (^4.1.17) - Utility-first CSS framework
- `postcss` - CSS transformation tool
- `autoprefixer` - Automatically add vendor prefixes
- `@vitejs/plugin-react` (^5.1.1) - React support for Vite
- `eslint` (^9.17.0) - Code quality and linting

Next Steps

Enhance Timer Functionality:

- Add sound/notification when timer reaches 00:00:00 using Web Audio API
- Implement browser notifications for timer completion
- Add timer presets (Pomodoro: 25min, Short break: 5min, Long break: 15min)
- Save timer history to localStorage
- Add stopwatch mode alongside countdown timer

Improve User Experience:

- Add dark/light mode toggle with Tailwind's dark mode feature
- Implement keyboard shortcuts (Space to pause/resume, R to restart, etc.)
- Add timer animations using CSS transitions or Framer Motion
- Create timer templates for common use cases (workout, study, cooking)
- Add progress bar visualization

Learn Advanced Concepts:

- Explore React Context API for state management across routes
- Learn about React hooks in depth (useState, useEffect, useRef, useCallback, useMemo)
- Implement custom hooks (useTimer, useLocalStorage, useKeyboardShortcut)
- Add unit tests with Vitest and React Testing Library
- Implement advanced routing features (route guards, nested routes, lazy loading)

Styling & Design:

- Experiment with Tailwind's animation utilities
- Create custom Tailwind theme extensions in tailwind.config.js
- Add responsive design for mobile devices
- Implement glassmorphism effects on more components
- Create a design system with consistent colors, spacing, and typography

Happy coding!