Building Reusable React Components: A Hands-On Tutorial

Objective

The objective of this hands-on tutorial is to guide learners through building a simple, scalable React application from scratch. Participants will learn how to create reusable and customizable UI components—specifically a Header, Footer, and Card—using modern React best practices. The tutorial emphasizes clean component design, prop-driven customization, styling techniques, and composition, laying a strong foundation for building maintainable and extensible React applications. Additionally, it introduces forward-thinking concepts such as component reusability, flexibility, and preparation for future enhancements like theming and testing.

Prerequisites

Before diving into this tutorial, ensure you have the following prerequisites to follow along smoothly:

- **Node.js (v14 or newer)** installed along with **npm** or **yarn**. These tools are essential for managing packages and running the React development server.
- **Basic knowledge of HTML, CSS, and modern JavaScript (ES6+)**. React builds on these core web technologies, so understanding them will help you grasp component structure, styling, and interactivity.
- **Familiarity with the command line**. You will use the terminal to create projects, install dependencies, and run your app during development.

Having these skills and tools set up ensures a seamless learning experience as you build and customize React components from scratch.

Installing Node.js and npm in VS Code

1. Open the Integrated Terminal

1. Launch VS Code

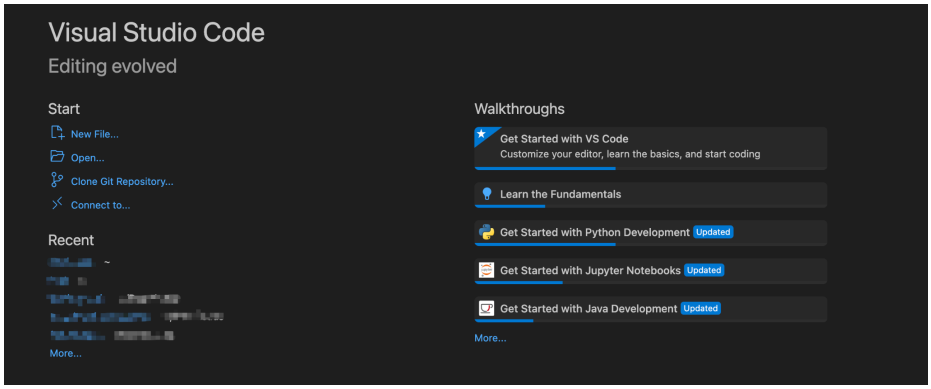

***Figure 1**: VS Code Default Window*

2. Open your existing project folder or create a new one:

    a. Click on “Open”.

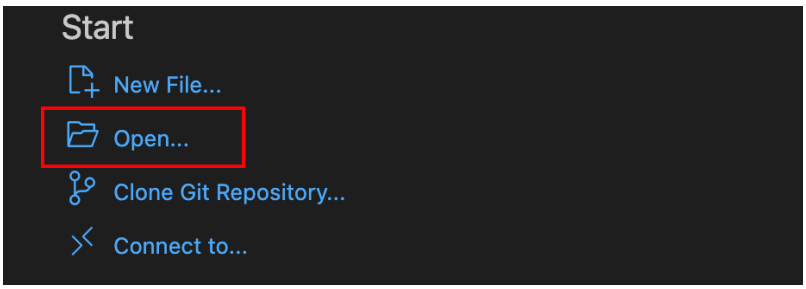

***Figure 2**: Opening the project directory in VS Code*

    b. Select the folder and click on “Open”.

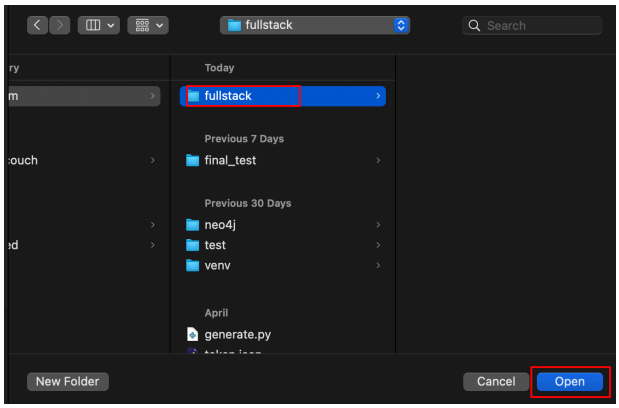

***Figure 3**: Selecting the project directory*

3. Open the terminal with Ctrl+` (backtick)

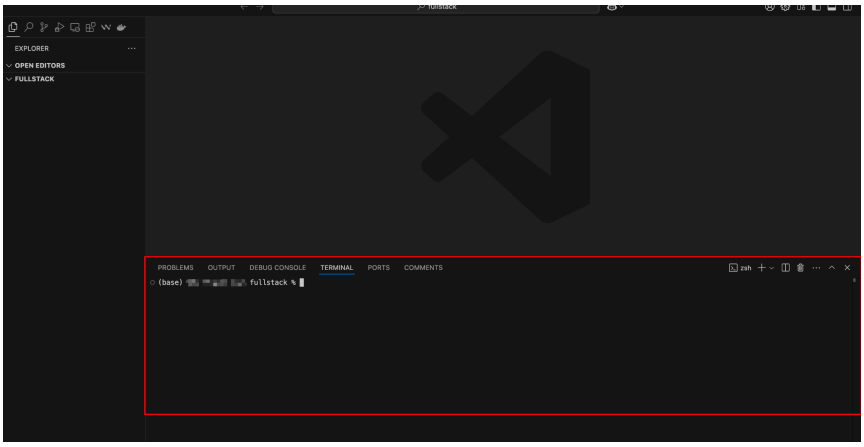

***Figure 4**: VS Code's Terminal*

2. For macOS / Linux (using nvm)

1. Install nvm (if not already installed)

```
curl -o-
https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.3/install.sh | bash
```

```
(base) shivam@cjkllmn fullstack % curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.3/install.sh | bash
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 16631  100 16631    0     0   106k      0 --:--:-- --:--:-- --:--:--  106k
=> Downloading nvm from git to '/Users/shivam/.nvm'
=> Cloning into '/Users/shivam/.nvm'...
remote: Enumerating objects: 382, done.
remote: Counting objects: 100% (382/382), done.
remote: Compressing objects: 100% (325/325), done.
remote: Total 382 (delta 43), reused 180 (delta 29), pack-reused 0 (from 0)
Receiving objects: 100% (382/382), 385.06 KiB | 2.07 MiB/s, done.
Resolving deltas: 100% (43/43), done.
* (HEAD detached at FETCH_HEAD)
  master
=> Compressing and cleaning up git repository

=> Appending nvm source string to /Users/shivam/.zshrc
=> Appending bash_completion source string to /Users/shivam/.zshrc
=> Close and reopen your terminal to start using nvm or run the following to use it now:

export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh"  # This loads nvm
[ -s "$NVM_DIR/bash_completion" ] && \. "$NVM_DIR/bash_completion"  # This loads nvm bash_completion
```

***Figure 5**: NVM installation*

2. Load nvm into your current shell session

```
source ~/.nvm/nvm.sh
```

3. Install the latest LTS version of Node.js (includes npm)

```
nvm install --lts
```

```
(base) shivam@cjkllmn fullstack %  nvm install --lts
Installing latest LTS version.
Downloading and installing node v22.16.0...
Downloading https://nodejs.org/dist/v22.16.0/node-v22.16.0-darwin-x64.tar.xz...
######################################################################################################################
Computing checksum with sha256sum
Checksums matched!
Now using node v22.16.0 (npm v10.9.2)
Creating default alias: default -> lts/* (-> v22.16.0)
```

***Figure 6**: Node installation*

4. Verify installation

```
node --version    # should print something like v18.x.x
npm --version     # should print something like 9.x.x
```

```
(base) shivam@cjkllmn fullstack % node --version
npm --version
v22.16.0
10.9.2
```

***Figure 7**: Checking node and npm versions*

3. For Windows

1. Download the windows installer from [nodejs' official website.]

2. Locate the downloaded .msi file and double-click to run it.

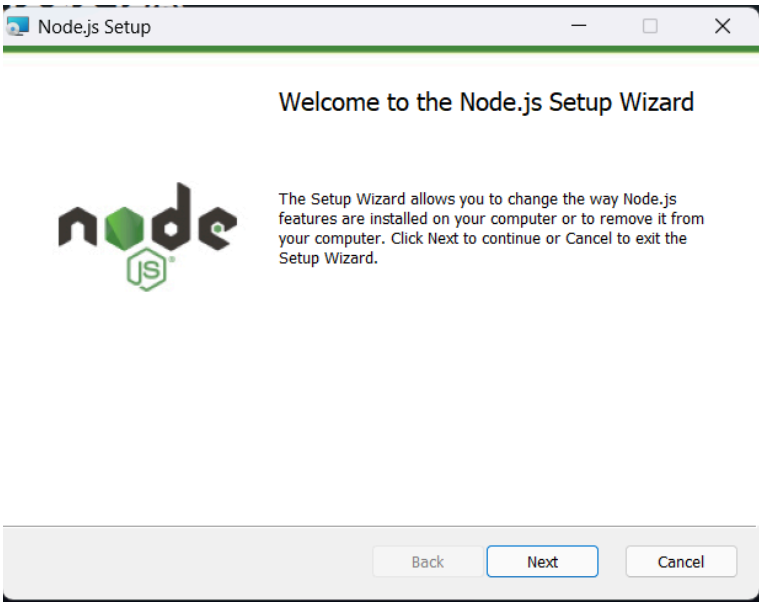

***Figure 8**: Node. js window's installer*

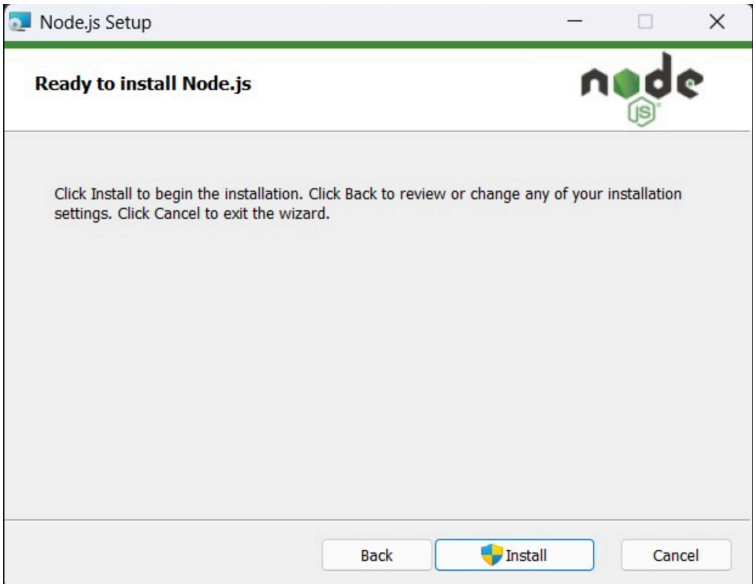

***Figure 9**: Node.js setup*

3. Follow the prompts in the setup wizard, accept the license agreement, and use the default settings for installation.

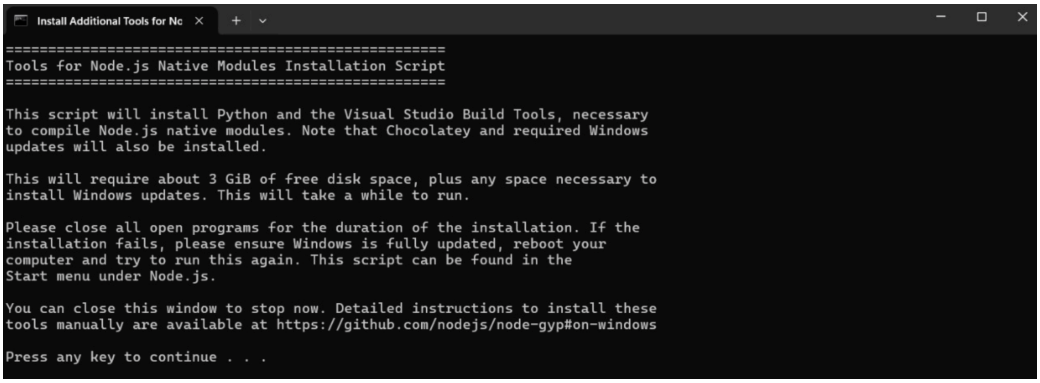

***Figure 10**: Node. js native module installation script*

4. Wait for "Finish" to complete the setup.

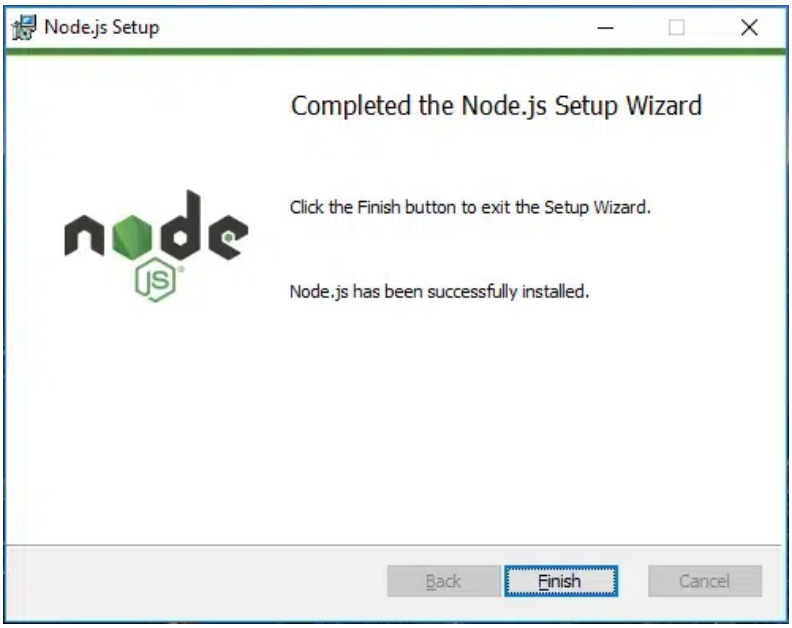

***Figure 11**: Node. js native module installation script*

5. Verify installation

```
node --version    # should print something like v18.x.x
npm --version     # should print something like 9.x.x
```

Project Setup

*To start building your React application, you first need to create a new project. This tutorial offers two popular methods: **Create React App**. This tool quickly scaffold a ready-to-use React environment with minimal configuration.*

Creating the Project

Open your terminal and navigate to the directory where you want to create your project. Then run one of the following commands:

- **Create React App:**

```
npx create-react-app my-app
```

Press “y” to continue the installation.

This command creates a new folder named my-app with all React dependencies and scripts configured.

```
Success! Created my-app at /Users/shivam/fullstack/my-app
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm run build
    Bundles the app into static files for production.

  npm test
    Starts the test runner.

  npm run eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd my-app
  npm start

Happy hacking!
npm notice
npm notice New major version of npm available! 10.9.2 -> 11.4.1
npm notice Changelog: https://github.com/npm/cli/releases/tag/v11.4.1
npm notice To update run: npm install -g npm@11.4.1
npm notice
```

***Figure 12**: Successful creation of React app*

You can verify the installation by checking the files in my-app directory.

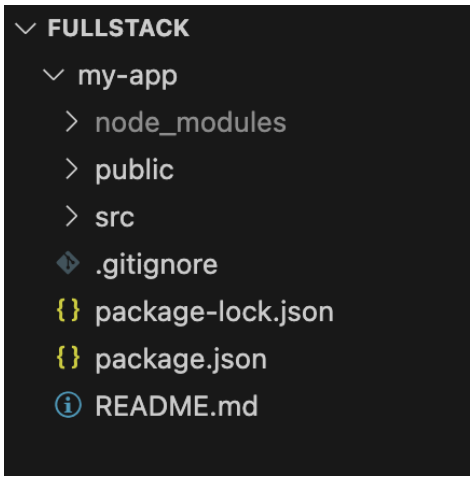

***Figure 13**: Structure of my-app directory.*

Navigating and Running the App

Change into the project directory and start the development server.

```
cd my-app
npm start   (for Create React App)
```

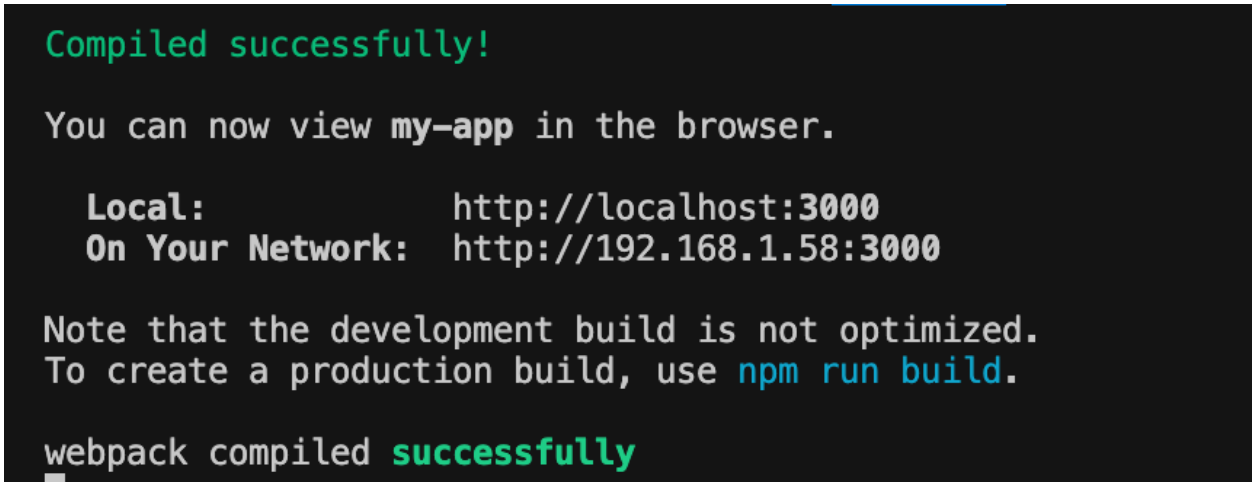

***Figure 14**: React app ready to be launched.*

Once running, open your browser and visit http://localhost:3000 (Create React App's default). You should see a React welcome page indicating your project is up and running.

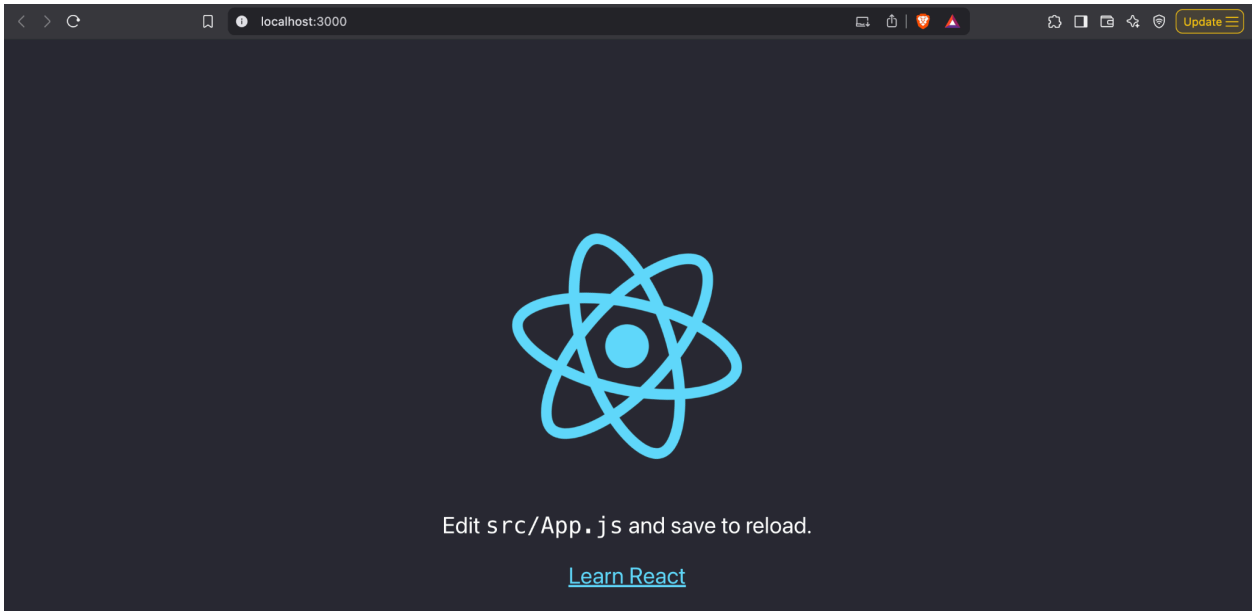

***Figure 15**: Default React app UI.*

Understanding the Project Structure

Your newly created React project contains several important folders and files:

- **src/** — This is where you will write your React components, styles, and other JavaScript or TypeScript code.
- **public/index.html** — The single HTML file serving as the entry point for your React app.
- **package.json** — Defines your project's dependencies, scripts, and metadata.

Exploring these folders will help you understand where to add your own components and how React assembles your project during development.

Building Reusable Components

One of React's greatest strengths is the ability to create *reusable components*—self-contained, modular pieces of UI that can be composed together to build complex interfaces. Reusability enhances maintainability and scalability while keeping your code DRY (Don't Repeat Yourself).

In this section, we will build three fundamental reusable components: **Header**, **Footer**, and **Card**. Each will demonstrate key React concepts including props, styling, composition, and accessibility.

Press ctrl + ` to close the running project and create a new folder "**components**" inside the **src** directory.

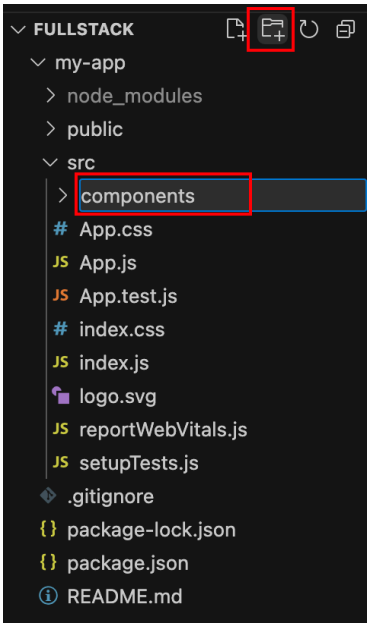

***Figure 16**: Creating components directory.*

1. Header Component

Inside the component create a Header.jsx file.

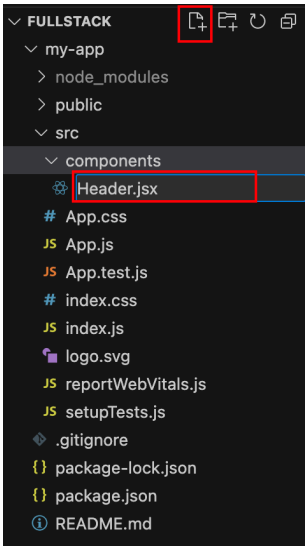

***Figure 17**: Creating Header Component.*

The Header will display your app's title and an optional navigation bar. It accepts two props:

- title (*string*) — The title text displayed prominently in the header.
- navLinks (*optional array* of objects with label and href) — Navigation links to display as a responsive menu.

Create the file src/components/Header.jsx.

```
import React from 'react';
import PropTypes from 'prop-types';
import styles from './Header.module.css';
/**
 * Header component with title and optional navigation links.
 */
const Header = ({ title, navLinks }) => {
  return (
    <header className={styles.header}>
      <h1 className={styles.title}>{title}</h1>
      {navLinks && navLinks.length > 0 && (
        <nav className={styles.nav}>
          <ul className={styles.navList}>
            {navLinks.map(({ label, href }) => (
              <li key={href} className={styles.navItem}>
```

```
                <a href={href} className={styles.navLink}>
                  {label}
                </a>
              </li>
            ))}
          </ul>
        </nav>
      )}
    </header>
  );
};
Header.propTypes = {
  title: PropTypes.string.isRequired,
  navLinks: PropTypes.arrayOf(
    PropTypes.shape({
      label: PropTypes.string.isRequired,
      href: PropTypes.string.isRequired,
    })
  ),
};
export default Header;
```

Explanation:

- **Props:**
    - `title`: A required string that sets the main heading of the header.
    - `navLinks`: An optional array of objects, each containing `label` and `href` for navigation links.
- **Structure:**
    - The header displays the title prominently.
    - If `navLinks` are provided, it renders a navigation bar with the provided links.
- **Styling:**

- Utilizes CSS Modules (`Header.module.css`) for scoped and maintainable styles.

Create Header.jsx's CSS file inside the components folder

CSS Modules Example (Header.module.css):

```css
/* Header.module.css */
.header {
  display: flex;
  align-items: center;
  justify-content: space-between;
  padding: 1rem 2rem;
  background-color: #282c34;
  color: white;
}

.title {
  font-size: 1.8rem;
}

.nav {
  flex-grow: 1;
  margin-left: 2rem;
}
.navList {
  display: flex;
  gap: 1.5rem;
  list-style: none;
  padding: 0;
  margin: 0;
}


.navLink {
  color: white;
  text-decoration: none;
  font-weight: 600;
}
```

```
.navLink:hover,
.navLink:focus {
  text-decoration: underline;
}

/* Responsive navigation for smaller screens */
@media (max-width: 600px) {
  .navList {
    flex-direction: column;
    gap: 0.75rem;
  }
}
```

Explanation: The Header component accepts required title and optional navLinks. Using CSS Modules provides scoped styles preventing naming conflicts. The nav links are rendered only if the navLinks array is non-empty. Responsive styles ensure the navigation transforms into a vertical list on narrow screens. The design keeps the header clean and semantic with accessible link structure.

2. Footer Component

The Footer component will appear at the bottom of your page. It accepts:

- year (*number*) — Typically the current year, displayed in the footer.
- text (*optional string*) — Additional text to show.
- children — Allows nested composition, so you can inject extra elements if needed.

Create the file src/components/Footer.jsx.

```
import React from 'react';
import PropTypes from 'prop-types';
import styles from './Footer.module.css';

/**
 * Footer component with year, optional text, and children support.
 */
const Footer = ({ year, text, children }) => {
  return (
    <footer className={styles.footer}>
      <p>
        &copy; {year} {text}
```

```
      </p>
      {children}
    </footer>
  );
};

// Provide default props for year (current year) and empty text
Footer.defaultProps = {
  year: new Date().getFullYear(),
  text: '',
};

Footer.propTypes = {
  year: PropTypes.number,
  text: PropTypes.string,
  children: PropTypes.node,
};

export default Footer;
```

Explanation:

- **Props:**
  - `year`: A number representing the year; defaults to the current year.
  - `text`: An optional string for additional footer text.
  - `children`: Allows insertion of additional elements or components within the footer.
- **Structure:**
  - Displays the year and optional text.
  - Renders any child elements passed to it.
- **Styling:**
  - Uses CSS Modules (`Footer.module.css`) to ensure styles are scoped to the component.

Create Module.jsx's CSS file inside the components folder

CSS Modules Example (Footer.module.css):

```
/* Footer.module.css */
.footer {
  background-color: #f5f5f5;
  padding: 1rem 2rem;
  text-align: center;
  font-size: 0.9rem;
  color: #555;
  border-top: 1px solid #ddd;
}
```

Explanation: This Footer component ensures that a year is always displayed using a default prop set to the current year. The optional text prop can be used for additional copyright or branding. Accepting children enables flexible extensions, such as social media links or small print, allowing further composability. The styles keep the footer simple, clean, and visually separated from page content.

3. Card Component

The Card component offers a flexible UI block to showcase content like images, titles, and descriptions. It accepts the following props:

- imageSrc (*string*) — URL or path of the image to display.
- title (*string*) — The card's title.
- description (*string*) — The card's descriptive text.
- onClick (*optional function*) — Callback for click or tap events.
- children — Allows nested additional content inside the card.
- styleOverrides (*optional object*) — Inline styles to customize the card container.

Create src/components/Card.jsx.

```
import React from 'react';
import PropTypes from 'prop-types';
import styles from './Card.module.css';

/**
* Card component with image, title, description, optional click
handler,
* children for extra content, style overrides, and accessibility
```

```
features.
*/
const Card = ({
 imageSrc,
 title,
 description,
 onClick,
 children,
 styleOverrides,
}) => {
 const handleKeyDown = (e) => {
   if (onClick && (e.key === 'Enter' || e.key === ' ')) {
     e.preventDefault();
     onClick();
   }
 };

 return (
   <div
     className={styles.card}
     style={styleOverrides}
     onClick={onClick}
     onKeyDown={handleKeyDown}
     role={onClick ? 'button' : undefined}
     tabIndex={onClick ? 0 : undefined}
   >
     <img src={imageSrc} alt={title} className={styles.image} />
     <div className={styles.content}>
       <h3 className={styles.title}>{title}</h3>
       <p className={styles.description}>{description}</p>
       {children}
     </div>
   </div>
 );
};

Card.propTypes = {
 imageSrc: PropTypes.string.isRequired,
 title: PropTypes.string.isRequired,
```

```
 description: PropTypes.string.isRequired,
 onClick: PropTypes.func,
 children: PropTypes.node,
 styleOverrides: PropTypes.object,
};

export default Card;
```

Explanation:

- **Props:**
    - `imageSrc`: URL or path of the image to display.
    - `title`: The card's title.
    - `description`: The card's descriptive text.
    - `onClick`: Optional function to handle click events.
    - `children`: Allows insertion of additional elements within the card.
    - `styleOverrides`: Optional object to override default styles.
- **Structure:**
    - Displays an image, title, and description.
    - If `onClick` is provided, the card becomes interactive, handling both click and keyboard events for accessibility.
- **Styling:**
    - Employs CSS Modules (`Card.module.css`) for scoped styling.
    - Includes hover and focus effects for better user experience.

Create Card.jsx's CSS file inside the components folder

CSS Modules Example (Card.module.css):

```
/* Card.module.css */
```

```
.card {
   border: 1px solid #ddd;
   border-radius: 6px;
   overflow: hidden;
   box-shadow: 0 2px 6px rgba(0,0,0,0.1);
   max-width: 300px;
   cursor: pointer;
   transition: transform 0.2s ease, box-shadow 0.2s ease;
 }
  .card:hover,
 .card:focus {
   transform: translateY(-5px);
   box-shadow: 0 8px 16px rgba(0,0,0,0.2);
   outline: none;
 }
  .image {
   display: block;               /* Removes inline spacing */
   margin: 0 auto;               /* Horizontally center the image */
   max-width: 70%;              /* Ensure it doesn't overflow */
   height: auto;                 /* Maintain aspect ratio */
   object-fit: contain;          /* Ensures full image is shown
without cropping */
   padding: 10px;              /* Optional spacing */
 }

  .content {
   padding: 1rem;
 }
  .title {
   margin: 0 0 0.5rem 0;
   font-size: 1.4rem;
 }
  .description {
   font-size: 1rem;
   color: #555;
 }
```

Explanation: The Card component handles several advanced use-cases: click handlers make the entire card interactive; keyboard handlers ensure accessibility by allowing keyboard users to activate the card with Enter or Space keys; and aria-pressed signals button semantics when

interactive. Supporting children and styleOverrides provides flexibility for users to enhance or customize cards without breaking encapsulation. The CSS hover animation gives a subtle uplift effect improving user experience and visual feedback.

Putting It All Together

Now that you have created your Header, Footer, and Card components, it's time to assemble them into a complete React application in src/App.jsx (or App.tsx if using TypeScript).

Importing Components and Preparing Data

Create the **assets** folder inside src and put all the images inside it from the given link. Update **App.js to App.jsx.**

Begin by importing the three components and defining an array of card data objects. Each object will hold properties like imageSrc, title, and description, which will be passed dynamically to each Card:

File: src/App.jsx

```
import React from 'react';
import Header from './components/Header';
import Footer from './components/Footer';
import Card from './components/Card';
import reactlogo from './assets/react.png';
import vueLogo from './assets/vue.png';
import angularLogo from './assets/angular.png';

const cardData = [
  {
    imageSrc: reactlogo,
    title: 'React',
    description: 'A JavaScript library for building user
interfaces.',
  },
  {
    imageSrc: vueLogo,
    title: 'Vue',
    description: 'The Progressive JavaScript Framework.',
  },
  {
    imageSrc: angularLogo,
```

```
    title: 'Angular',
    description: 'One framework. Mobile & desktop.',
  },
];

const App = () => {
  const handleCardClick = (title) => {
    alert(`You clicked on ${title} card!`);
  };

  return (
    <div>
      <Header
        title="My App"
        navLinks={[
          { label: 'Home', href: '/' },
          { label: 'About', href: '/about' },
          { label: 'Contact', href: '/contact' },
        ]}
      />
      <main style={{ display: 'flex', gap: '1rem', padding: '1rem'
}}>
        {cardData.map(({ imageSrc, title, description }) => (
          <Card
            key={title}
            imageSrc={imageSrc}
            title={title}
            description={description}
            onClick={() => handleCardClick(title)}
          />
        ))}
      </main>
      <Footer text="All rights reserved." />
    </div>
  );
};

export default App;
```

Explanation:

- **Structure:**
    - Renders the `Header` at the top with navigation links.
    - Displays a series of `Card` components in the main section, each representing a JavaScript framework.
    - Includes the `Footer` at the bottom with additional text.
- **Interactivity:**
    - Each `Card` component is clickable, triggering an alert displaying the card's title.
- **Styling:**
    - Uses inline styles for layout in the `main` section; however, for a production application, consider using CSS Modules or styled-components for better maintainability.

Verifying the Application

Start your development server and open your browser to view the app. You should see:

- The **Header** component at the top showing your app title.
- A responsive grid or row of **Card** components in the center displaying images, titles, and descriptions. Clicking a card triggers an alert.
- The **Footer** component at the bottom with the current year and additional text.

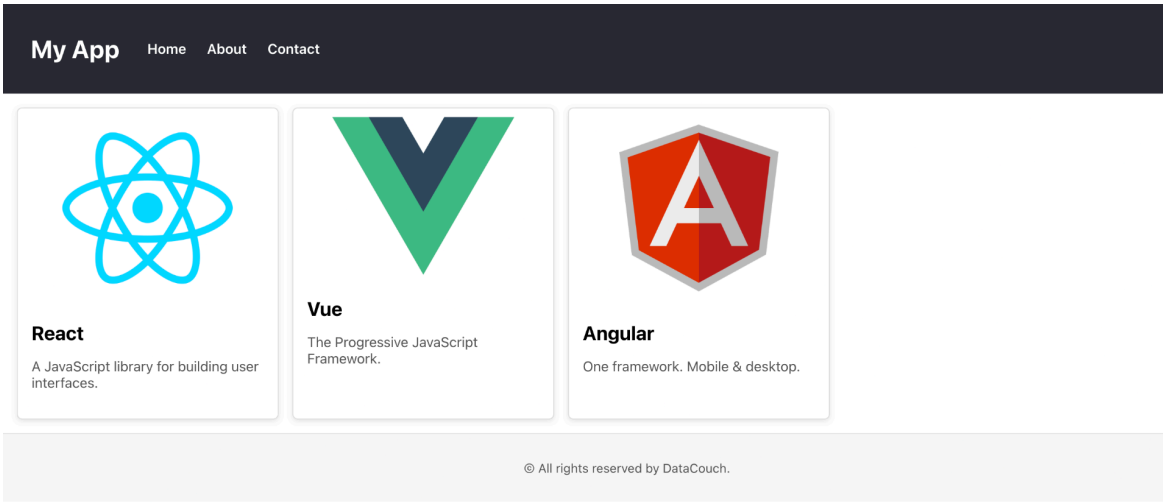

***Figure 18**: Project's UI.*

This layout demonstrates how to compose reusable components, pass dynamic props, and handle events effectively.

Best Practices and Next Steps

Project Structure and Organization

Organizing your React project with clear, consistent folder structures greatly improves maintainability and scalability. A common approach is:

- src/components/: Contains all reusable React components like Header, Footer, and Card.
- src/assets/: Holds images, fonts, icons, and other static files that your components reference.
- src/styles/ (optional): For global CSS or theme files if not using CSS-in-JS.

Global Theming and Context

To manage consistent styles and enable features like dark mode, employ React Context or libraries such as styled-components with a ThemeProvider. For example, a dark-mode toggle can switch themes globally:

- Define light and dark theme objects containing colors and fonts.
- Wrap your app with ThemeProvider and toggle theme state in context.
- Use theme values in styled components or CSS-in-JS styles to dynamically adapt UI.

Testing Strategies

Use React Testing Library alongside Jest for reliable component tests:

- **Props Testing:** Verify components render expected output when passed different props.

- **Event Testing:** Simulate user interactions like clicks and keyboard events to check handlers.
- **Snapshot Testing:** Capture rendered markup snapshots to detect unintended UI changes.

Performance Tips

- **React.memo:** Memoize pure components to avoid unnecessary re-rendering when props do not change.
- **React.lazy and Suspense:** Lazy-load components or routes to reduce initial bundle size and improve load times.

Conclusion:

In this hands-on session, you've successfully set up a React application and built foundational, reusable components—`Header`, `Footer`, and `Card`. These components demonstrate how to structure UI elements with flexibility and clarity, using props to ensure reusability and separation of concerns.

You now have a working React app that follows best practices in component design, styling, and organization. This structure sets the stage for more advanced features like routing, global state management, dark/light theming, and testing frameworks.

Forward-Thinking Notes

Consider adopting advanced React features and modern tooling to future-proof your projects:

- ***React Server Components:*** *Hybrid rendering approaches allowing data fetching and rendering on the server for performance.*
- ***GraphQL Integration:*** *Simplifies data fetching with declarative queries and reduces over-fetching.*
- ***CI/CD Pipelines:*** *Automate testing, building, and deployment with tools like GitHub Actions, ensuring quality and fast iteration.*
- ***Deployment Platforms:*** *Use Vercel, Netlify, or similar services optimized for React apps with support for serverless functions and global CDN.*