2024 EDITION REACT SPA DEVELOPMENT WITH V18

MASTERING REACT 18

SPA DEVELOPMENT

A FIVE-PART SERIES FROM BEGINNER TO ADVANCED

BY BRAUN GOODSON

MASTERING REACT

A FIVE-PART SERIES FROM BEGINNER TO ADVANCED

2024 EDITION REACT SPA DEVELOPMENT WITH V18.3

BY BRAUN GOODSON

About this book

Throughout the five-part series, you'll be building a **Personal Task Manager**—a to-do list application

This series will guide you through the essentials of building a React application, starting from the basics and progressing to advanced topics. By the end of the series, you'll have a solid understanding of React and single-page app development, along with a deployable project to showcase your skills.

Part I: Introduction to React and Single-Page Applications

Part II: React Fundamentals – Components, Props, and JSX

1.1

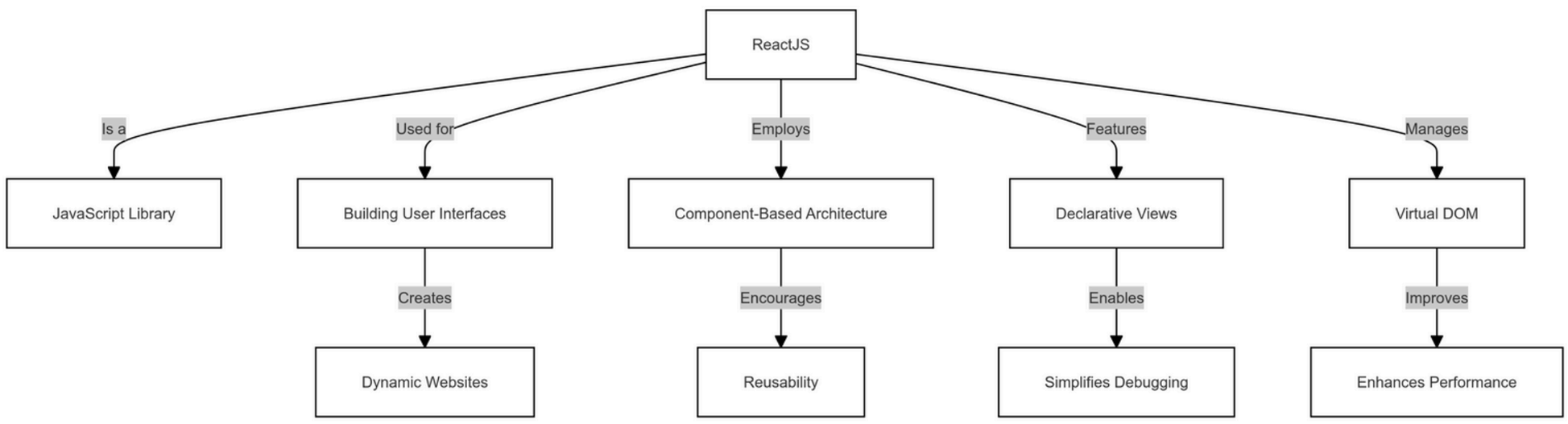
What is React?

Overview of React and Its Benefits

By definition React is an open-source JavaScript library for building user interfaces, developed by Facebook. Its Component-Based Architecture allows you to build encapsulated components that manage their own state. Its Virtual DOM efficiently updates and renders just the right components when your data changes.

Some of the benefits of using react are:

- *Reusable Components*: Write code once and reuse it throughout your app.
- *Performance*: Faster rendering with the Virtual DOM.
- *Developer Tools*: Strong ecosystem with tools like React Developer Tools.



The Role of React in Modern Web Development

React’s popularity is seen widely adopted in the industry, used by companies like Facebook, Instagram, and Netflix. Its ecosystem has extensive community support, libraries, and tutorials. React’s versatility enables use for web, mobile (React Native), and even desktop applications.

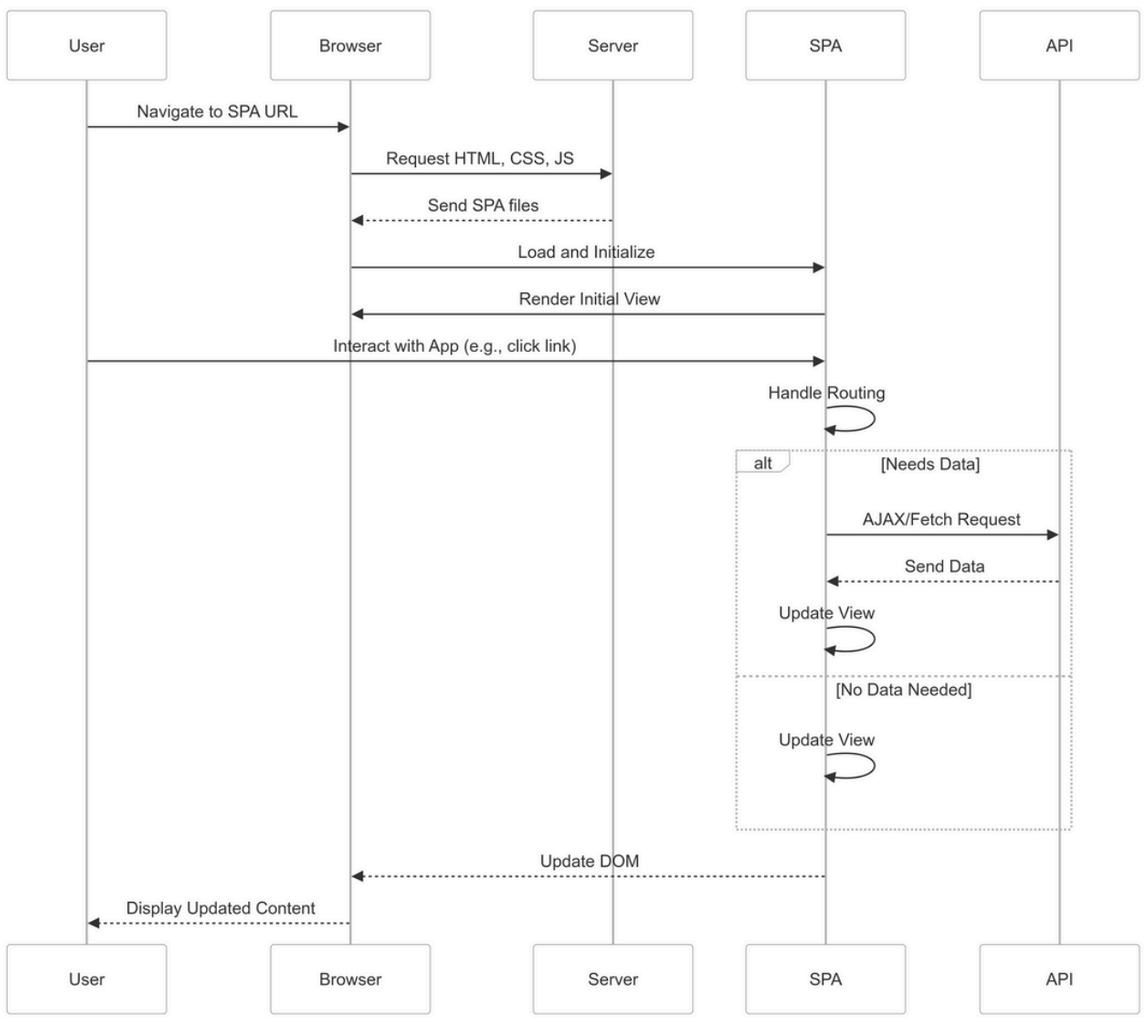
1.2

Understanding Single-Page Applications (SPAs)

Difference Between SPAs and Traditional Web Apps

With Traditional Web Apps every new page request loads a new HTML page from the server. Page reloads can be slow and disrupt the user experience.

Whereas Single-Page Applications load a single HTML page and dynamically updates content as the user interacts. It uses AJAX and HTML5 to create fluid and responsive web apps.



Advantages of Using SPAs

At the top, is performance and faster load times after the initial load. Next is User Experience for smooth transitions without full page reloads. The Development Efficiency enables code reuse and easier debugging. They have Offline Capabilities meaning better support for offline scenarios using service workers.

1.3

Setting Up the Development Environment

Installing Node.js and NPM

Download Node.js by visiting Node.js website and download the LTS version. Install Node.js by running the installer and following the prompts. Verify installation by opening your terminal and typing:

```
node -v && npm -v
```

to check the installed versions.

Using Create React App to Bootstrap Your Project

Install Create React App globally by running:

```
npm install -g create-react-app
```

Create a new React App by navigating to your desired directory and running:

```
npx create-react-app task-manager-app
```

Then navigate into the new `task-manager-app` directory. Start the development server by running:

```
npm start
```

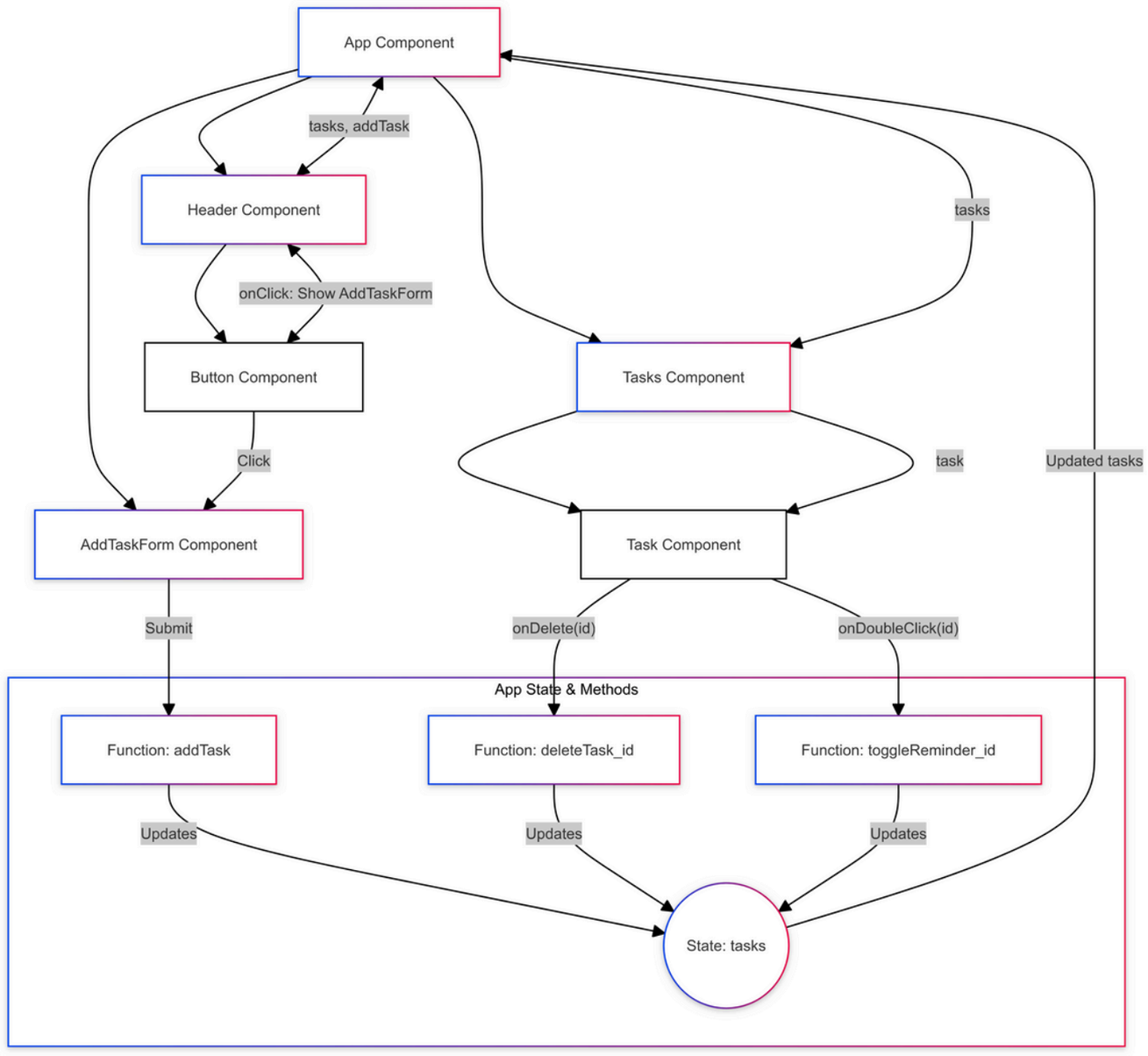
Finally, open http://localhost:3000 in your browser to view the app.

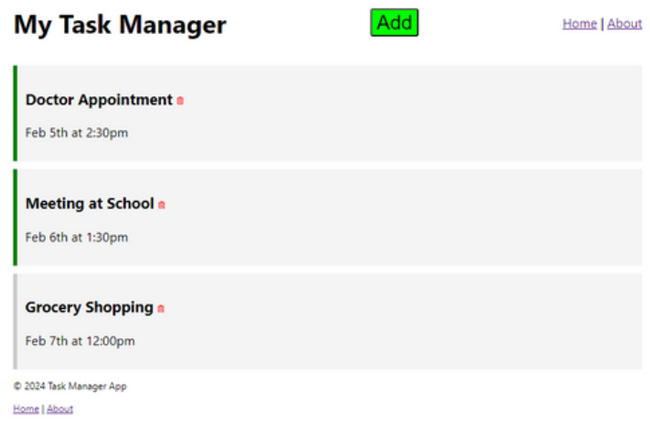
1.4

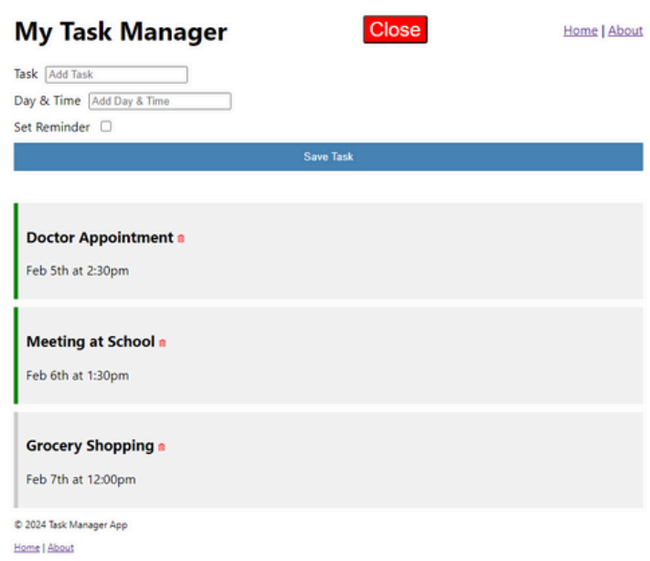
Preview of the Project Series

Brief Overview of the Application You'll Build

Build a personal Task Manager App. You can add new tasks by entering task detail and add them to your list. View Task List to see all your tasks displayed neatly. Manage Tasks as complete or delete them. Navigate Between Views using routing to switch between different pages like 'All Tasks' and 'Completed Taks'.







What You'll Learn

React Fundamentals such as Components, props, and JSX. State Management using hooks like `useState` and `useEffect`. Advanced Concepts with Context API, and routing with React Router. Performance Optimization by code splitting, and memoization. Finally Deployment to deploy your app so the world can see it.

Closing Remarks

That's it for this introductory book for Part I in the series Mastering React! You've set up your development environment and have a roadmap of what we'll achieve together. In the next book Part II, we'll start diving into React fundamentals by building components and understanding JSX. Don't forget to like, share, and subscribe so you don't miss out on the next part of the series. See you soon!

Additional Resources:

- *React Documentation*: reactjs.org
- *Node.js Downloads*: nodejs.org
- *Code Editor Suggestions*:
  i. Visual Studio Code
  ii. Atom
  iii. Sublime Text

Call to Action:

- *Subscribe*: Stay updated with the latest books in this series.
- *Comments*: If you have any questions or need clarifications, drop a comment Braun's e-mail `brn.gdsn@gmail.com`.
- *Share*: If you found this book helpful, share it with others who might benefit.

The End

Thank you for Reading! Up next: "React Fundamentals – Components, Props, and JSX".

2.1

Understanding Components

What Are Components?

By definition Components are the building blocks of any React application. They encapsulate reusable code that represents a part of the user interface.

Here we focus on two types of Components:

- *Functional Components*: JavaScript functions that return JSX.
- *Class Components*: ES6 classes that extend `React.Component` (less common in modern React due to hooks).

Creating Your First Functional Component

Begin with creating a new file. Change directory into the `src` folder, and create a new directory called `components`. Then, create a file named `Header.js` inside the `components` folder. In that file, and in every react file you'll import `React`, write the component code with a function that returns a JSX view, and export the function as the component.

```
// Header.js
import React from 'react';

const Header = () => {
 return (
  <header>
   <h1>Personal Task Manager</h1>
  </header>
 );
};

export default Header;
```

Import and use the component by opening `App.js` in the `src` folder, and including it in your `<App />` view's JSX.

```
// App.js
import React from 'react';
import Header from './components/Header';

function App() {
 return (
  <div className="App">
   <Header />
  </div>
 );
}

export default App;
```

2.2

Working with JSX

What is JSX?

By definition JSX stands for JavaScript XML. It allows you to write HTML-like syntax within JavaScript, hich React then transforms into actual DOM elements.

Here we focus on two reasons:

- *Readability*: Makes the code easier to understand.
- *Integration*: Allows you to write HTML and JavaScript together.

Embedding Expressions in JSX

An example of embedding expressions in JSX:

```
const userName = 'John Doe';

const Greeting = () => {
 return <p>Hello, {userName}!</p>;
};
```

Note: Use curly braces `{}` to embed JavaScript expressions with JSX. You can perform calculations, call functions, and access object properties.

JSX Rules to Remember

Return a Single Parent Element, wrapping elements inside a single parent, like our `Greeting` use of `p` or `React.Fragment`. Class vs className, use `className` instead of `class` for assigning CSS classes. Se;f-Closing Tags, elements without children should be self-closed (e.g., `<img />`, `<br />`).

2.3

Props in React

What Are Props?

By definition “Props” is short for properties. They are read-only inputs that are passed into components, similar to function arguments. The purpose here is to pass data from a parent component to a child component, and to make components dynamic and reusable.

Passing Props to Components

By modifying the `task-manager-app`’s parent `App` component:

```
function App() {
 const title = ’My Task Manager’;

 return (
  <div className=“App“>
   <Header title={title} />
  </div>
 );
}
```

Access Props in Child Component

In the `Header.js`, access the `title` prop.

```
import React from ’react’;

const Header = (props) => {
 return (
  <header>
   <h1>{props.title}</h1>
  </header>
 );
}; // ... rest of code
```

Destructuring Props

Destructuring props simplify access:

```
const Header = ({ title }) => {
 return (
  <header>
   <h1>{title}</h1>
  </header>
 );
};
```

Default Props

React is now expected to leverage native default props instead of using `Header.defaultProps = { title: ‘Task Manager’ }`:

```
const Header = ({ title = ‘Task Manager’ }) => {
 return (
  <header>
   <h1>{title}</h1>
  </header>
 );
};
```

2.4

Building Reusable Components

Creating a Button Component

In this section, we create a reusable Button component in React. Buttons are a crucial part of user interfaces, and building a custom button component allows for greater flexibility and consistency across a project. We'll also demonstrate how to integrate this component into other parts of this application, such as the header.

To begin, navigate to the `components`, and create a new file named `Button.js`. This file will contain the code for a button component.

```
import React from 'react';

const Button = ({ color = 'steelblue', text }) => {
 return <button style={{ backgroundColor: color }}>{text}</button>;
};

export default Button;
```

In this snippet, we're defining a functional component named `Button`. Let's break it down:

- *Props*: This component accepts two props: `color` and `text`. The `color` prop controls the background color of the button, and `text` defines the label displayed on the button.
- *Default Props*: We have set a default value for the `color` prop to `"steelblue"` in case no color is passed in when the button is used.

By using props, this button component becomes reusable in various contexts, allowing consumers to create buttons with different colors and labels.

Destructuring Props

Destructuring props simplify access:

```
const Header = ({ title }) => {
 return (
  <header>
   <h1>{title}</h1>
  </header>
 );
};
```

Default Props

React is now expected to leverage native default props instead of using `Header.defaultProps = { title: ‘Task Manager’ }`:

```
const Header = ({ title = ‘Task Manager’ }) => {
 return (
  <header>
   <h1>{title}</h1>
  </header>
 );
};
```

Using the Button Component

Now that the `Button.js` component ready, the next step is to integrate it into other parts of the application. Here we use the `Button` component inside the `Header` component.

```
import Button from ’./Button’;

const Header = ({ title }) => {
 return (
  <header>
   <h1>{title}</h1>
   <Button color=“green“ text=“Add Task“ />
  </header>
 );
};
```

Here’s what’s happening:

- *Importing the Button Component*: The `Header` component imports the `Button` component from the `Button.js` file.
- *Rendering the Button*: Inside the `Header`'s JSX, we call the `Button` component, passing in `"green"` as the background color and `"Add Task"` as the button text.

This is a simple example of how components can be reused and customized by passing different props. In this case, the button will appear with a green background and the label "Add Task."

By following these steps, you've created a reusable button component that can be easily styled and adapted based on the context in which it's used. You've also learned how to integrate this button into a header, laying the groundwork for more complex interactions in your React application.

Styling Components

Styling is a key part of creating visually appealing user interfaces in React applications, and deserves a book of its own. Here, we will touch swiftly on two common approaches to styling components: *inline styling* and *CSS stylesheets*. Both methods allow you to control the appearance of your components, giving you flexibility based on the complexity and scale of your project.

```
const Button = ({ color, text }) => {
 return (
  <button
   style={{
    backgroundColor: color,
    color: '#fff',
    padding: '10px 20px',
    border: 'none',
    borderRadius: '5px',
    cursor: 'pointer',
   }}
  >
   {text}
  </button>
 );
};
```

Inline styling involves adding styles directly to elements through the `style` prop in React. The `style` prop accepts a JavaScript object

where CSS properties are written in camelCase. This approach is useful for simple, dynamic styles that need to change based on component state or props.

In the example above, we revisit the `Button.js` component and see what it looks like to add inline styling to enhance its appearance (we won't use this pattern in the `task-manager-app`).

While inline styles are great for simple use cases, CSS stylesheets provide a more scalable solution, especially when dealing with more complex or reusable styles. By separating style definitions into a `.css` file, you can maintain clean and organized code while keeping styles reusable across different components (we won't use this pattern in the `task-manager-app`).

In the `components` folder, we would create a new file called `Header.css` and add the following styles:

```
header {
  display: flex;
  align-items: center;
  justify-content: space-between;
}
```

In the `Header.js` file, we would import the CSS file to apply the styles.

```
import './Header.css';
import Button from './Button';

const Header = ({ title }) => {
  return (
    <header>
      <h1>{title}</h1>
      <Button color="green" text="Add Task" />
    </header>
  );
};
```

By importing the CSS file, React automatically applies the styles defined in `Header.css` to the `header` element in the `Header.js` component.

After mastering both inline styles and CSS stylesheets, you can choose the right approach for each scenario in your React applications, maintaining a balance between flexibility and maintainability.

2.5

Building the Task List UI

Creating the Task Component

In this section, we will apply the concepts discussed so far by building a functional Task List user interface (UI) using React. This example will guide you through creating reusable components, managing state, and structuring your application effectively. By the end of this section, you will have a clear understanding of how to assemble a simple yet powerful task management UI.

The foundation of our Task List UI lies in the Task component, which is responsible for rendering individual tasks. To begin, navigate to the components folder within your project directory and create a new file named `Task.js`. This component will receive task data as props and display the task's text and scheduled day.

```
// Task.js
import React from 'react';

const Task = ({ task }) => {
 return (
  <div className="task">
   <h3>{task.text}</h3>
   <p>{task.day}</p>
  </div>
 );
};

export default Task;
```

In the above code:

- *Import Statements*: We import React to utilize JSX.
- *Task Functional Component*: The `Task` component receives a task-
  -

- --object as a prop. It returns a div containing the task's text and day, styled appropriately with class names for potential CSS styling.
- *Export Statement*: Finally, we export the `Task` component to make it available for import in other parts of the application.

Creating the Tasks Component to List Multiple Tasks

With the `Task` component in place, the next step is to create a `Tasks` component that will manage and render a list of tasks. This component will iterate over an array of task objects and render each one using the `Task` component we previously defined.

Create a new file named `Tasks.js` within the components folder and add the following code:

```
// Tasks.js
import React from 'react';
import Task from './Task';

const Tasks = ({ tasks }) => {
 return (
  <>
  {tasks.map((task) => (
   <Task key={task.id} task={task} />
  ))}
  </>
 );
};

export default Tasks;
```

Here's a breakdown of the `Tasks` component:

- *Import Statements*: We import `React` and the `Task` component to utilize within this component.
- *Tasks Functional Component*: The `Tasks` component receives an array of tasks as a prop. Using the map function, it iterates over each task object and renders a `Task` component for each one.
- *Key Prop*: The key prop is crucial for optimizing rendering performance in `React`. By providing a unique identifier (`task.id`), `React` can efficiently manage and update the list of tasks.
- *Fragment Wrapper*: The empty `<>...</>` fragments are used to wrap the list of `Task` components without adding extra nodes to the DOM.

Including the Tasks Component

With both the `Task` and `Task`s components established, it's time to integrate them into the main application. This involves updating the `App.js` file to include the `Tasks` component and provide it with the necessary task data.

Open `App.js` and modify it as follows:

```
import React from 'react';
import Header from './components/Header';
import Tasks from './components/Tasks';

function App() {
 const tasks = [
  {
   id: 1,
   text: 'Doctors Appointment',
   day: 'Feb 5th at 2:30pm',
  },
  {
   id: 2,
   text: 'Meeting at School',
   day: 'Feb 6th at 1:30pm',
  },
 ];

 return (
  <div className="App">
   <Header />
   <Tasks tasks={tasks} />
  </div>
 );
}

export default App;
```

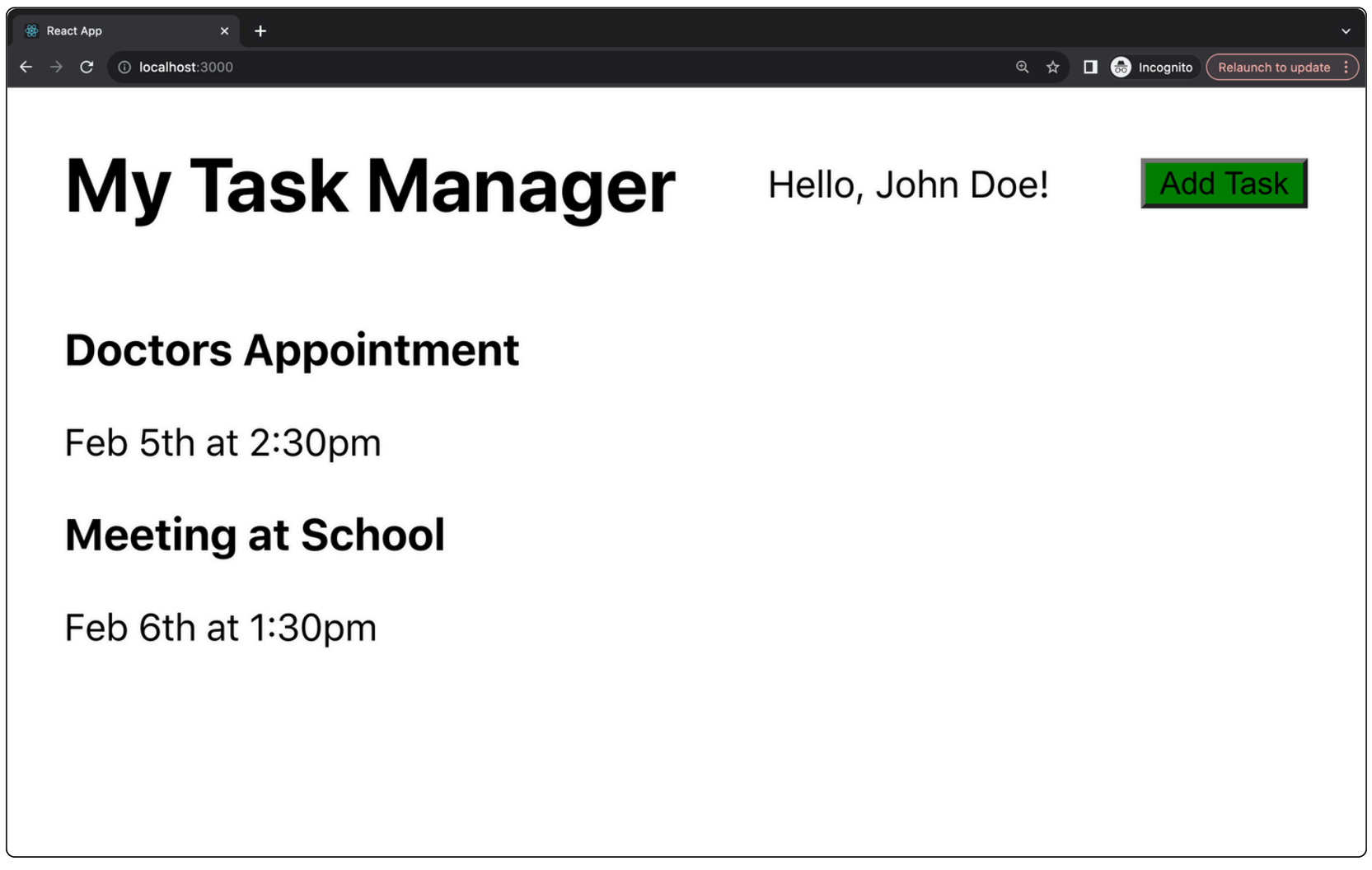
In this updated `App.js`:

- Import Statements: We import `React`, the `Header` component, and the newly created `Tasks` component.
- *Defining Tasks Data*: A constant array tasks is defined, containing objects that represent individual tasks. Each task has an id, text, and day, which correspond to the properties expected by the `Task` component.

- *Rendering Components*: Within the return statement, we render the `Header` component followed by the Tasks component. The tasks array is passed as a prop to the `Tasks` component, enabling it to render the list of tasks dynamically.
- *Export Statement*: The `App` component is exported for use in other parts of the application.

With these steps completed, the application now has a functional Task List UI. The App component serves as the central hub, orchestrating the rendering of the header and the list of tasks. The modular approach, utilizing separate components for individual tasks and the task list, ensures maintainability and scalability as your application grows.

At this point, you've successfully built a Task List UI using `React`. You've created reusable components that encapsulate specific functionalities, making your codebase organized and efficient. This modular design not only simplifies the development process but also facilitates easier testing and future enhancements. As you continue to develop the application, leveraging such component-based architectures will prove invaluable in managing complexity and fostering a seamless user experience.



2.6

Recap and What's Next

What We Covered

- *Components*: Learned about functional components and how to create them.
- *JSX*: Understood how JSX works and how to embed JavaScript expressions.
- *Props*: Passed data between components using props. Used default props and PropTypes for better code quality.
- *Building UI*: Started constructing the UI for the task manager with reusable components.

Sneak Peek into the Next Section

- *State Management*: We'll introduce state in React and how to manage it within components.
- *Hooks*: Learn about `useState` and `useEffect` hooks to add interactivity.
- *Enhancing the Task Manager*: Make the task list dynamic by adding functionality to add and remove tasks.

Closing Remarks

Great job! You've taken your first steps into building a `React` application by understanding components, props, and JSX. In the next chapter, we'll make our app interactive by introducing state and hooks. Don't forget to like, share, and subscribe to stay updated. See you in the next lesson!

Additional Resources:

- *React Documentation on Components*: React Components
- *JSX in Depth*: JSX Docs

Call to Action:

- *Subscribe*: Stay updated with the latest books in this series.
- *Comments*: If you have any questions or need clarifications, drop a comment Braun's e-mail `brn.gdsn@gmail.com`.
- *Share*: If you found this book helpful, share it with others who might benefit.

The End

Thanks for reading! Up next: "State Management and Hooks".

3.1

Understanding State in React

Difference Between Props and State

React, one of the most popular JavaScript libraries for building user interfaces, introduces important concepts that help developers create interactive and dynamic applications. One of these concepts is state. To understand state, it helps to first understand how it differs from props, another key concept in React.

- *Props*: Props, short for “properties,“ are pieces of data that are passed from a parent component down to its child components. They are read-only, which means that the child component cannot modify them. This makes props immutable in the child context, ensuring that the data stays consistent throughout the app.

```
<eg-component egProp=”I’m a prop.” />
```

- *State*: State, on the other hand, is data that is managed internally within a component. It is mutable, which means it can be updated over time, allowing components to track changes. When the state of a component changes, React automatically re-renders that component to reflect the updated data in the UI.

```
const [egState, egStateSetter] = useState(0);
```

Why State is Important

State is crucial for building interactive and dynamic user interfaces. Here are a few reasons why:

- Interactivity: State allows components to respond to user input and events, making your application interactive. For instance, when a

- *user types in a form or clicks a button, state can be updated to reflect those actions.*
- *Dynamic UI: By enabling changes to the state, the UI can be updated dynamically. This means that the app doesn't stay static; instead, it adjusts based on user actions, creating a more engaging user experience.*
- *Data Flow*: State also facilitates the flow of data within the application. Components can manage their own state and pass down important information to their children via props, keeping data consistent and easy to trace.

Understanding how props and state work together is fundamental to developing powerful React applications. Props are like the fixed details that you need to pass to child components, whereas state is the evolving part of your component, enabling the dynamic behavior that makes your apps feel alive.

3.2

Introduction to React Hooks

What Are Hooks?

React Hooks, introduced in React 16.8, are special functions that let you “hook into“ React features like state and lifecycle methods from functional components. Hooks allow developers to use state and other React features without needing to write class components.

Hooks simplify React development by allowing functional components to manage their own state and perform side effects. This means you can write cleaner, more reusable code, and keep related logic closely bundled.

Commonly Used Hooks

- *useState*: Manages state in functional components, enabling dynamic and interactive behavior.
- *useEffect*: Handles side effects, such as data fetching, subscriptions, or manually updating the DOM, in functional components.
- *useContext*: Allows components to consume context, making it easier to share values like themes or authentication status across the component tree.

React Hooks provide an elegant and efficient way to handle state, lifecycle, and context in functional components, making your React code more organized and easy to maintain.

MASTERING REACT 18

A FIVE-PART SERIES FROM BEGINNER TO ADVANCED

Throughout the five-part series, you'll be building a Personal Task Manager—a to-do list application

This series will guide you through the essentials of building a React application, starting from the basics and progressing to advanced topics. By the end of the series, you'll have a solid understanding of React and single-page app development, along with a deployable project to showcase your skills.

GITHUB REPOSITORY

Why This App?

- Beginner-Friendly: It's simple enough for newcomers to grasp basic concepts.
- Core React Features: Covers essential topics like components, props, state, and hooks.
- Advanced Concepts: Provides a foundation to introduce context API, routing, and performance optimization.
- Practical Use: Creates a functional tool that viewers can use and expand upon after the series.