

[Open in app](#)

Write



★ Get unlimited access to the best of Medium for less than \$1/week. [Become a member](#)



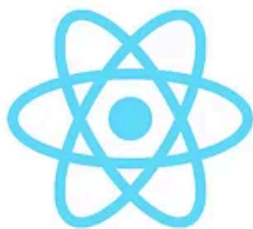
# Maximize Your React Skills: Build a To-Do List App from Start to Finish (with TypeScript + Vite)

Master the basics of building an app with React and TS. Learn how to use state, props, and data flow to create a fully functional app.



Eduardo Motta de Moraes · Follow

Published in Bits and Pieces · 20 min read · Jan 25, 2023



React + Vite + TypeScript

In this blog post, we're going to go through everything you need to **understand** and **build** a basic React application. Whether you're a beginner just getting started with React or a seasoned developer looking to brush up on your skills, this guide is for you.

This guide will take you through the entire process of building a fully-functional to-do list app, including design, layout, state management, and more. We will be using functional components and hooks. We will learn how to use state and props to pass data between components, and how to handle user input and update the state of your app.

By the end of this guide, we will have a solid understanding of how to build a React app from scratch, and you will be able to take your newfound knowledge to build your own React projects.

So, let's get started!

\*You can find the code of the app we're going to be building [here](#), and the live version [here](#).

## A brief intro

We'll be using TypeScript for writing the code and Vite for developing and building the app.

### TypeScript

TypeScript is a strongly typed programming language that builds on JavaScript. In practical terms, if you already know JavaScript, all you need to learn to use TypeScript is how to use types and interfaces.

Types and interfaces allow us to define the data types we're using in the code. With this, we can catch bugs early on and avoid problems down the line.

For instance, if a function takes a `number` but we pass it a `string`, TypeScript will complain immediately:

```
const someFunc = (parameter: number) => {...};  
  
someFunc('1') // Argument of type 'string' is not assignable to parameter of type
```

If we were using JavaScript we'd likely only catch the bug later on.

We don't always need to specify the type, as TypeScript can infer them automatically more often than not.

You can learn the basics of TypeScript [here](#). (Or just ignore the types.)

## Vite

The most common way of spinning up a React application is probably using [create-react-app](#). We'll be using [Vite](#) (pronounced like “veet”) instead. But fret not, it's just as simple — but more efficient.

With tools like [webpack](#) (used by create-react-app under the hood), your entire application needs to be bundled in a single file before it can be served to the browser. Vite, on the other hand, takes advantage of native ES modules in the browser to make bundling more efficient with [Rollup](#), serving parts of the source code as needed.

Vite can also greatly speed up development time with Hot Module Replacement — meaning whenever changes are made to the source code, only the changes are updated, rather than the entire application.

Besides that, Vite offers native support for Typescript, JSX and TSX, CSS and more.

Similarly to create-react-app, Vite offers a tool called create-vite, that allows us to quickly start a new project using basic templates, including options for Vanilla JS, or using libraries like React.

To be clear, we don't *need* a tool like Vite or create-react-app to build React applications, but they make our life easier by taking care of setting up the project, bundling the code, using transpilers and much more.

## Diving into React

### JSX / TSX

React allows us to add markup directly in the code which will later be compiled to plain JavaScript. This is called JSX. When we're using JSX we can save our files as .jsx for JavaScript or .tsx for TypeScript.

It looks like this:

```
const element = <h1>Hello, world!</h1>;
```

It's similar to HTML, but it's embedded in the JavaScript file, and it allows us to manipulate the markup with programming logic. We can also add JavaScript code inside the JSX, as long as it's inside curly brackets.

For instance, if we have an array of text we want to render as different paragraph elements, we could do this:

```
const paragraphs = ['First', 'Second', 'Third']  
  
paragraphs.map((paragraph) => <p>{paragraph}</p>)
```

And it would be compiled to something like this:

```
<p>First</p>
<p>Second</p>
<p>Third</p>
```

But if we try to do just that, it won't work. That's because React works with components, and JSX needs to be rendered inside these components.

## React components

React components can be written using JavaScript **classes** or just plain **functions**. We'll be focusing on function components, as they are the most up-to-date and the recommended way of writing React components today.

A component is defined by a function that will return the JSX which will be compiled and rendered by the browser. So to extend the example above, if we want to render the paragraph elements, it would look something like this:

```
// Define the component
const Component = () => {
  const paragraphs = ["First", "Second", "Third"];
  return (
    <>
      {paragraphs.map((paragraph) => (
        <p>{paragraph}</p>
      ))}
    </>
  );
}

// Use the component in the same way you use an HTML element in the JSX
const OtherComponent = () => {
  return (
    <Component />
  )
}
```

## Props

Now maybe we want to reuse this component with different information. We can do that by using props — which is just a JavaScript object holding some data.

In our example, instead of hardcoding the array, we could pass it to the component. The result will be the same, but now the component will be reusable.

If we're using TypeScript, we need to specify the types of the data inside the props object (there's no context for what they are, so TypeScript can't infer them), which in this case is an array of strings ( `string[]` ).

```
const Component = (props: {paragraphs: string[]}) => {  
  <>  
    {props.paragraphs.map((paragraph) => <p>{paragraph}</p>)}  
  </>  
}  
  
const OtherComponent = () => {  
  const paragraphs = ['First', 'Second', 'Third']  
  return (  
    <Component paragraphs={paragraphs}/>  
  )  
}
```

## State

If we want to make an interactive component, we're going to need to store information in the component's state, so it can “remember” it.

For instance, if we want to define a simple counter that shows the number of times a button is clicked, we need a way of storing and updating this value. React lets us do it with the `useState` hook (a hook is a function that lets you “hook” into the React state and lifecycle features).

We call the `useState` hook with the initial value, and it returns to us an array with the value itself and a function to update it.

```
import { useState } from 'react'

const Counter = () => {
  const [count, setCount] = useState(0);
  return (
    <>
      <span>{count}</span>
      <button onClick={() => setCount(count + 1)}>Increment count</button>
    </>
  )
}
```

With this knowledge, we're now ready to start building our React app.

## Creating the project

### Dependencies

To use Vite we're going to need **node** and a package manager.

To install node just choose one of the options [here](#) depending on your system and configurations. If you're using Linux or a Mac, you can also install it using [Homebrew](#).

The package manager can be [npm](#) or [yarn](#). In this post we're going to using **npm**.

### Creating the project

Next it's time to create the project. In the terminal, we navigate to the directory where the project will be created, then run the `create-vite` command.

```
$ npm create vite@latest
```

We may be prompted to install additional packages (like create-vite). Type `y` and press enter to continue.

```
Need to install the following packages:
  create-vite@4.0.0
Ok to proceed? (y)
```

Next we'll be prompted to enter the project information.

Enter the **name** of the project. I chose `my-react-project`.

```
? Project name: › my-react-project
```

Select **React** as the “framework”.

React is technically a library and not a framework, but don't worry about it.

```
? Select a framework: › - Use arrow-keys. Return to submit.
  Vanilla
  Vue
  ›  React
  Preact
  Lit
  Svelte
  Others
```

Select **TypeScript + SWC** as variant.



SWC (stands for Speedy Web Compiler ) is a super-fast TypeScript / JavaScript compiler written in Rust. They claim to be “20x faster than Babel on a single thread and 70x faster on four cores”.

```
? Select a variant: > - Use arrow-keys. Return to submit.  
  JavaScript  
  TypeScript  
  JavaScript + SWC  
> TypeScript + SWC
```

It's done, the project is created. To start it in development mode, we need to change to the project directory, install the dependencies and run the dev script command.

```
cd my-react-project  
npm install  
npm run dev
```

After a few seconds, we will see something similar to this:

```
VITE v4.0.4 ready in 486 ms  
  
→ Local:   http://localhost:5173/  
→ Network: use --host to expose  
→ press h to show help
```

If we open our browser and navigate to <http://localhost:5173/> we'll see the default Vite + React page:



This means that everything is as it should be and we can start working on our app.

## Building the app

### File structure and initial setup

If we open the project in our code editor or IDE of choice, we should see a file structure like this:

```
✓ MY-REACT-APP
  > node_modules
  ✓ public
    vite.svg
  ✓ src
    ✓ assets
      react.svg
    # App.css
    TS App.tsx
    # index.css
    TS main.tsx
    TS vite-env.d.ts
    .gitignore
    <> index.html
    {} package-lock.json
    {} package.json
    TS tsconfig.json
    {} tsconfig.node.json
    TS vite.config.ts
```

We can delete some of the boilerplate files, since we won't be using them (all .svg and .css files).

The code in the App function can be deleted to leave us with this:

```
function App() {  
  return (  
  
  )  
}  
  
export default App
```

We'll come back to this file later.

## Styling

Styling isn't the focus here, but we'll be using Tailwind CSS, which is a library that lets us style HTML elements by adding classes to them. Follow [these instructions](#) to see the styles reflected in your own project.

Otherwise you can just ignore the classes in the code.

## Thinking about the design: components layout

The design process is an integral part of the development of an app and shouldn't be overlooked.

To build our to-do list app, we need to first think of the components layout.

We start by mocking a basic UI and outlining a hierarchy of the components involved.

If you're not a designer, it doesn't need to be perfect or the final UI in terms of colors and exact placement yet — it's more important to think of the components structure.

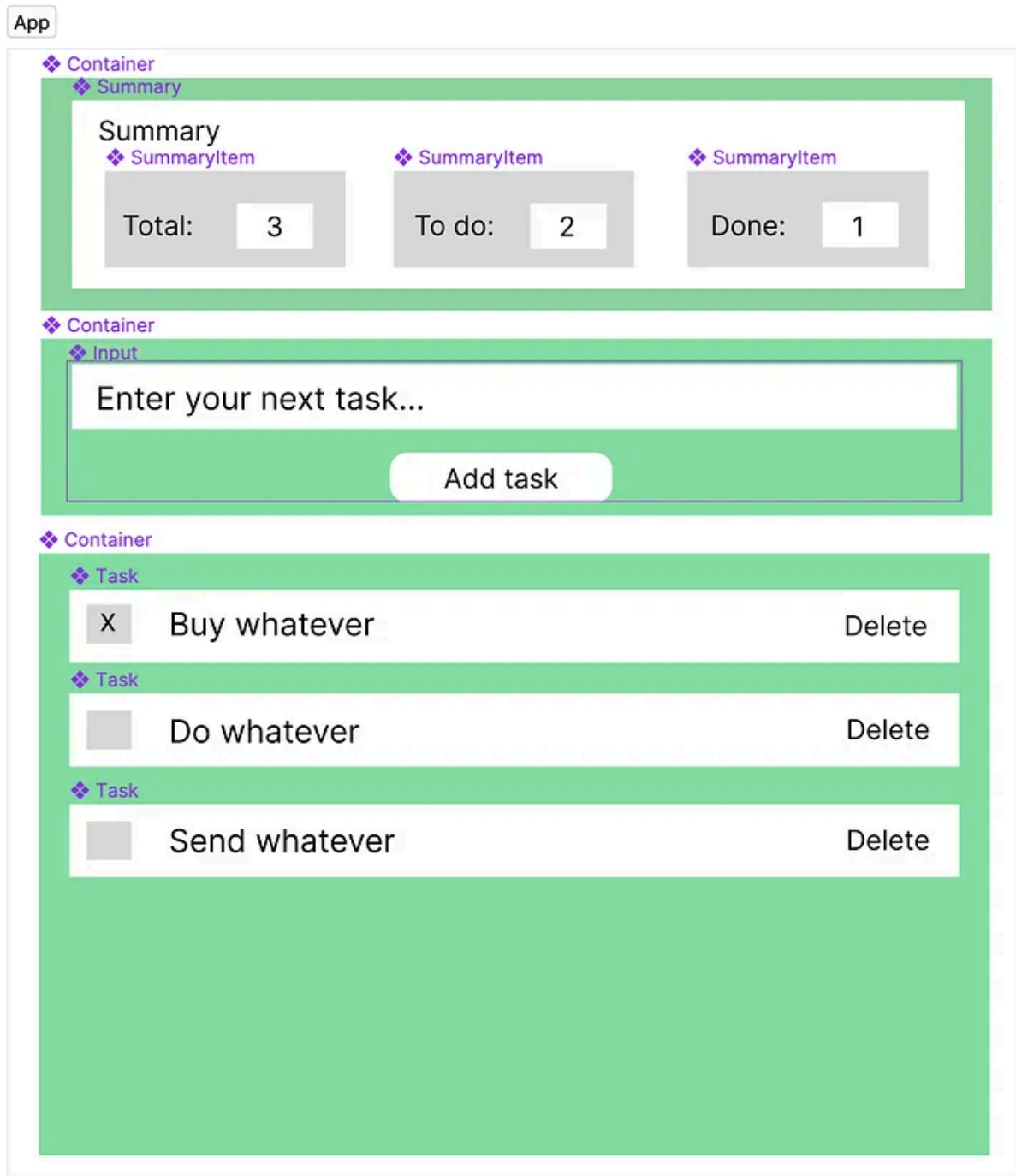
Ideally, our components should be responsible for only one thing, following the [single-responsibility principle](#). This is one of the fundamental principles of composable design and is promoted as a best practice in toolchains such as [Bit](#).

**How to reuse React components across your projects**

Finally, you completed the task of creating a fantastic input field for the form in your app. You are happy with the...

bit.dev

In the image below, the names in purple are the components we're going to be building — everything else are native HTML elements. If they're inside each other, it means there's likely going to be a parent-child relationship.



### Props: building a static version

After we have a sketch, we can start building a static version of the app. That is to say, just the UI elements, but with no interactivity yet. This part is pretty straightforward and involves a lot of typing and little thinking once you get the hang of it.

You can find the code for the static version in this [GitHub repository](https://github.com/maximilian-schneiders/build-a-to-do-list-app-from-start-to-finish-with-typescript-vite-b1b5e0faecbe), in the branch “static-version”. The code for the fully working app is the main

branch.

## Container

As outlined above, we're going to have a `Container` that is reused for every section of the app. This `Container` shows one of the ways of composing different elements: by passing them as children.

```
// src/components/Container.tsx
const Container = ({
  children,
  title,
}: {
  children: JSX.Element | JSX.Element[];
  title?: string;
}) => {
  return (
    <div className="bg-green-600 p-4 border shadow rounded-md">
      {title && <h2 className="text-xl pb-2 text-white">{title}</h2>}}
      <div>{children}</div>
    </div>
  );
};

export default Container;
```

It takes a props object with a `children` parameter of type `JSX.Element | JSX.Element[]`. This means we can compose it with any other HTML element or any other components we create. It can be rendered wherever we want inside the container — in this case inside the second `div`.

In our app, it's going to render each section (defined below) when we use them inside the `App` component.

The `Container` also takes an optional `string` prop named `title`, which will be rendered inside an `h2` whenever it exists.

```
// src/App.tsx
import Container from "../components/Container";
import Input from "../components/Input";
import Summary from "../components/Summary/Summary";
import Tasks from "../components/Tasks/Tasks";

function App() {
  return (
    <div className="flex justify-center m-5">
      <div className="flex flex-col items-center">
        <div className="sm:w-[640px] border shadow p-10 flex flex-col gap-10">
          <Container title={"Summary"}>
            <Summary />
          </Container>
          <Container>
            <Input />
          </Container>
          <Container title={"Tasks"}>
            <Tasks />
          </Container>
        </div>
      </div>
    </div>
  );
}

export default App;
```

## Summary

The first section is a summary ( `Summary` component) showing three items ( `SummaryItem` ): the total number of tasks, the number of pending tasks and the number of completed tasks. This is another way of composing components: just use them in the return statement of another component.

(It's important to never *define* a component inside another component, though, as that can lead to unnecessary renders and bugs.)

For now we can just use static data in the two components.



```
// src/components/Summary/SummaryItem.tsx
const SummaryItem = ({
  itemName,
  itemValue,
}: {
  itemName: string;
  itemValue: number;
}) => {
  return (
    <article className="bg-green-50 w-36 rounded-sm flex justify-between p-2">
      <h3 className="font-bold">{itemName}</h3>
      <span className="bg-green-900 text-white px-2 rounded-sm">
        {itemValue}
      </span>
    </article>
  );
};

export default SummaryItem;

// src/components/Summary/Summary.tsx
import SummaryItem from "../SummaryItem";

const Summary = () => {
  return (
    <>
      <div className="flex justify-between">
        <SummaryItem itemName={"Total"} itemValue={3} />
        <SummaryItem itemName={"To do"} itemValue={2} />
        <SummaryItem itemName={"Done"} itemValue={1} />
      </div>
    </>
  );
};

export default Summary;
```

You will notice `SummaryItem` takes two props: `itemName`, of type `string`, and `itemValue`, of type `number`. These props are passed when the `SummaryItem` component is used inside the `Summary` component, and then rendered in the `SummaryItem` JSX.

## Tasks

Similarly, for the tasks section (the last one) we have a `Tasks` component that renders the `TaskItem` components.

Also with static data for now. We will later need to pass a **task name** and a **status** down as props to the `TaskItem` component to make it reusable and dynamic.

```
// src/components/Tasks/TaskItem.tsx
const TaskItem = () => {
  return (
    <div className="flex justify-between bg-white p-1 px-3 rounded-sm">
      <div className="flex gap-2 items-center">
        <input type="checkbox" />
        Task name
      </div>
      <button className="bg-green-200 hover:bg-green-300 rounded-lg p-1 px-3">
        Delete
      </button>
    </div>
  );
};

export default TaskItem;

// src/components/Tasks/Tasks.tsx
import TaskItem from "../TaskItem";

const Tasks = () => {
  return (
    <div className="flex flex-col gap-2">
      <TaskItem />
    </div>
  );
};

export default Tasks;
```

## Input

Finally, the `Input` component is a form with a `label`, an `input` of type `text`, and a button to “Add task”. For now it doesn’t do anything, but we’ll soon change that.

```
// src/components/Input.tsx
const InputContainer = () => {
  return (
    <form action="" className="flex flex-col gap-4">
      <div className="flex flex-col">
        <label className="text-white">Enter your next task:</label>
        <input className="p-1 rounded-sm" />
      </div>
      <button
        type="button"
        className="bg-green-100 rounded-lg hover:bg-green-200 p-1"
      >
        Add task
      </button>
    </form>
  );
};

export default InputContainer;
```

## State: adding interactivity

To add interactivity in React, we need to store information in the component's state.

But before doing that, we need to think about how we want the data to change over time. We need to identify a **minimal representation** of this data, and identify **which components** we should use to store this state.

### A minimal representation of state

State should contain every bit of information necessary to make our app interactive — but nothing more. If we can compute a value from a different value, we should keep just one of them in state. This makes our code not only less verbose, but also less prone to bugs involving contradictory state values.

In our example we might think we need to track values for total tasks, pending tasks, and done tasks.

But to track the tasks, it is enough to have a single array with objects representing each task and its status (pending or done).

```
const tasks = [  
  {  
    name: "task one",  
    done: false,  
  },  
  {  
    name: "task two",  
    done: true,  
  },  
];
```

With this data we can always find all of the other information we need at render time using array methods. We also avoid the possibility of contradictions,— like having a total of 4 tasks, but only 1 pending and 1 done task, for example.

We also need state in our `form` (in the `Input` component) so we can make it interactive.

## Where the state should live

Think of it this way: which components need to access the data we're going to store in state? If it's a single component, the state can live in this component itself. If it's more than one component that need the data, then you should find the common parent to these components.

In our example, the state necessary to control the `Input` component only needs to be accessed there, so it can be local to this component.

```
// src/components/Input.tsx  
import { useState } from "react";
```

```

const InputContainer = () => {
  const [newTask, setNewTask] = useState(""); // Initialize newTask and setNewTa
  return (
    <form action="" className="flex flex-col gap-4">
      <div className="flex flex-col">
        <label className="text-white">Enter your next task:</label>
        <input
          className="p-1 rounded-sm"
          type="text"
          value={newTask} // Set the input value to newTask
          onChange={(e) => setNewTask(e.target.value)} // Set newTask to the inp
        />
      </div>
      <button
        type="submit"
        className="bg-green-100 rounded-lg hover:bg-green-200 p-1"
      >
        Add task
      </button>
    </form>
  );
};

export default InputContainer;

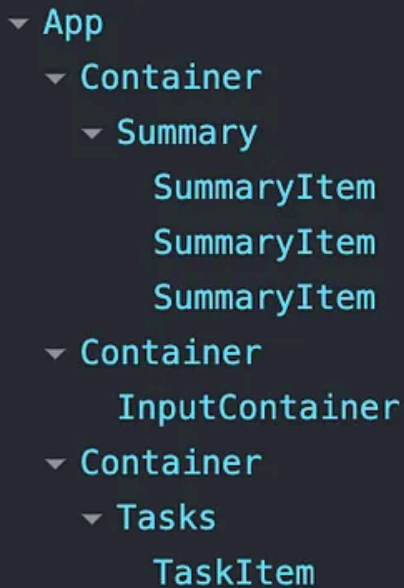
```

What this is doing is displaying our `newTask` value in the input, and calling the `setNewTask` function whenever the input changes (i.e., when the user types something).

We won't see any immediate changes in the UI, but this is necessary so we can control the input and have access to its value to use it later.

The state to track the tasks, however, has to be handled differently, as it needs to be accessed in the `SummaryItem` components (we need to show the number of total, pending and done tasks) as well as in the `TaskItem` components (we need to display the tasks themselves). And it needs to be the same state because this information must always be in sync.

Let's take a look at our component tree (you can use the [React dev tools](#) for this).



We can see that the first common parent component is `App`. So this is where our state for the tasks is going to live.

With the state in place, all that will be left will be to pass the data down as props to the components that need to use it.

(We're not yet worried about how to make and persist any changes to the parent state, that's the next part.)

```
// src/App.tsx
import { useState } from "react";
import { v4 as uuidv4 } from 'uuid'
import Container from "../components/Container";
import Input from "../components/Input";
import Summary from "../components/Summary/Summary";
import Tasks from "../components/Tasks/Tasks";

export interface Task {
  name: string;
  done: boolean;
  id: string;
}

const initialTasks = [
  {
    name: "task one",
    done: false,
```

```

      id: uuidv4(),
    },
    {
      name: "task two",
      done: true,
      id: uuidv4(),
    },
  ],
];

function App() {
  const [tasks, setTasks] = useState<Task[]>(initialTasks);
  return (
    <div className="flex justify-center m-5">
      <div className="flex flex-col items-center">
        <div className="border shadow p-10 flex flex-col gap-10 sm:w-[640px]">
          <Container title={"Summary"}>
            <Summary tasks={tasks} />
          </Container>
          <Container>
            <Input />
          </Container>
          <Container title={"Tasks"}>
            <Tasks tasks={tasks} />
          </Container>
        </div>
      </div>
    </div>
  );
}

export default App;

```

Here we're initializing the tasks value with dummy data ( `initialTasks` ), just so that we can visualize it before the app is finished. Later we can change it to an empty array, so a new user will not see any tasks when opening the app fresh.

Besides the `name` and `done` properties, we're also adding an `id` to our task objects, as it will be necessary shortly.

We're defining an `interface` with the types of the value in the task objects, and passing it to the `useState` function. This is necessary in this case, as TypeScript will not be able to infer it when we change the initial value of `tasks` to an empty array, or when we pass it as props. ▶

Finally, notice we're passing the tasks down as props to the `Summary` and `Tasks` components. These components will need to be changed to accommodate that.

```
// src/components/Summary/Summary.tsx
import { Task } from "../../App";
import SummaryItem from "../SummaryItem";

const Summary = ({ tasks }: { tasks: Task[] }) => {
  const total = tasks.length;
  const pending = tasks.filter((t) => t.done === false).length;
  const done = tasks.filter((t) => t.done === true).length;
  return (
    <>
      <div className="flex flex-col gap-1 sm:flex-row sm:justify-between">
        <SummaryItem itemName={"Total"} itemValue={total} />
        <SummaryItem itemName={"To do"} itemValue={pending} />
        <SummaryItem itemName={"Done"} itemValue={done} />
      </div>
    </>
  );
};

export default Summary;
```

We updated the `Summary` component so that it now accepts `tasks` as a prop. We also defined the value `total`, `pending` and `done`, which will be passed down as props to the `SummaryItem` components in place of the static `itemValue`'s we had before.

```
// src/components/Tasks/Tasks.tsx
import { Task } from "../../App";
import TaskItem from "../TaskItem";

const Tasks = ({ tasks }: { tasks: Task[] }) => {
  return (
    <div className="flex flex-col gap-2">
      {tasks.map((t) => (
        <TaskItem key={t.id} name={t.name} />
      ))}
    </div>
  );
};
```



```
export default Tasks;

// src/components/Tasks/TaskItem.tsx
import { useState } from "react";

const TaskItem = ({ name }: { name: string }) => {
  const [done, setDone] = useState(false);
  return (
    <div className="flex justify-between bg-white p-1 px-3 rounded-sm gap-4">
      <div className="flex gap-2 items-center">
        <input type="checkbox" checked={done} onChange={() => setDone(!done)} />
        {name}
      </div>
      <button className="bg-green-200 hover:bg-green-300 rounded-lg p-1 px-3">
        Delete
      </button>
    </div>
  );
};

export default TaskItem;
```

For the `Tasks` component, we also take `tasks` as a prop, and map its `name` property to `TaskItem` components. As a result, we get a `TaskItem` component for each object inside the `tasks` array. We also update the `TaskItem` component to accept `name` as a prop.

This is where the `id` comes in handy, as we need to pass a unique key every time we have a list of child components. If we don't add the key, this could lead to bugs on rerender. (In a production app, the `id` would most likely come from the backend.)

The result for now is this:

The screenshot displays a To-Do application interface with a green theme. It is divided into three main sections:

- Summary:** A green header bar containing three white boxes. The first box shows 'Total' with a value of '2'. The second box shows 'To do' with a value of '1'. The third box shows 'Done' with a value of '1'.
- Input:** A green box with the text 'Enter your next task:' followed by a white text input field and a light green 'Add task' button.
- Tasks:** A green box containing a list of tasks. Each task is in a white box with a checkbox, the task name, and a light green 'Delete' button. The tasks are 'task one' and 'task two'.

We can already see the summary numbers and the task names reflecting our dummy data. But we still lack a way to add or delete tasks.

### Adding inverse data flow

To finish our app, we need a way to change the `App` component state (where the tasks data is) from the `Input` and `TaskItem` child components.

To do that, we can use the functions generated by the `useState` hook to define event handlers, and pass them down as props. Once we do that, we simply call them during the appropriate user interaction from the child components.

Be sure to never mutate state whenever you're updating it, as this will lead to bugs. Always replace the state object with a new one when updating it.

Below is our final `App` component with the handlers declared and passed down as props to the `Input` and `Tasks` components.

`handleSubmit` returns a new array with the old tasks plus the new one.

`toggleDoneTask` returns a new array with the opposite `done` property, for the specified `id`. `handleDeleteTask` returns a new array without the task with the specified `id`.

```
// src/App.tsx
import { FormEvent, useState } from "react";
import { v4 as uuidv4 } from "uuid";
import Container from "../components/Container";
import Input from "../components/Input";
import Summary from "../components/Summary/Summary";
import Tasks from "../components/Tasks/Tasks";

export interface Task {
  name: string;
  done: boolean;
  id: string;
}

function App() {
  const [tasks, setTasks] = useState<Task[]>([]);

  const handleSubmit = (e: FormEvent<HTMLFormElement>, value: string) => {
    e.preventDefault();
    const newTask = {
      name: value,
      done: false,
      id: uuidv4(),
    };
    setTasks((tasks) => [...tasks, newTask]);
  };

  const toggleDoneTask = (id: string, done: boolean) => {
    setTasks((tasks) => {
      tasks.map((t) => {
        if (t.id === id) {
          t.done = !done;
        }
        return t;
      })
    })
  };
}
```

```

    };

    const handleDeleteTask = (id: string) => {
      setTasks((tasks) => tasks.filter((t) => t.id !== id));
    };

    return (
      <div className="flex justify-center m-5">
        <div className="flex flex-col items-center">
          <div className="border shadow p-10 flex flex-col gap-10 sm:w-[640px]">
            <Container title={"Summary"}>
              <Summary tasks={tasks} />
            </Container>
            <Container>
              <Input handleSubmit={handleSubmit} />
            </Container>
            <Container title={"Tasks"}>
              <Tasks
                tasks={tasks}
                toggleDone={toggleDoneTask}
                handleDelete={handleDeleteTask}
              />
            </Container>
          </div>
        </div>
      </div>
    );
  }

  export default App;

```

This is the final `Input` component using `handleSubmit` to update the `App` component state.

```

// src/components/Input.tsx
import { FormEvent, useState } from "react";

const InputContainer = ({
  handleSubmit,
}: {
  handleSubmit: (e: FormEvent<HTMLFormElement>, value: string) => void;
}) => {
  const [newTaskName, setNewTaskName] = useState("");
  return (
    <form
      action=""
      className="flex flex-col gap-4"
      onSubmit={(e) => {
        handleSubmit(e, newTaskName);
      }}
    >

```

```

        setNewTaskName("");
      }}
    >
    <div className="flex flex-col">
      <label className="text-white">Enter your next task:</label>
      <input
        className="p-1 rounded-sm"
        type="text"
        value={newTaskName}
        onChange={(e) => setNewTaskName(e.target.value)}
      />
    </div>
    <button
      type="submit"
      className="bg-green-100 rounded-lg hover:bg-green-200 p-1"
    >
      Add task
    </button>
  </form>
);
};

export default InputContainer;

```

This is the final `Tasks` component, which we updated to pass the props from `App` down to `TaskItem`. We also added a ternary operator to return “No tasks yet!” when there are no tasks.

```

// src/components/Tasks/Tasks.tsx
import { Task } from "../../App";
import TaskItem from "../TaskItem";

const Tasks = ({
  tasks,
  toggleDone,
  handleDelete,
}): {
  tasks: Task[];
  toggleDone: (id: string, done: boolean) => void;
  handleDelete: (id: string) => void;
} => {
  return (
    <div className="flex flex-col gap-2">
      {tasks.length ? (
        tasks.map((t) => (
          <TaskItem
            key={t.id}
            name={t.name}
            done={t.done}

```

```

        id={t.id}
        toggleDone={toggleDone}
        handleDelete={handleDelete}
      />
    ))
  ) : (
    <span className="text-green-100">No tasks yet!</span>
  )}
</div>
);
};

export default Tasks;

```

And this is the final `TaskItem` component, using `toggleDone` and `handleDelete` to update the `App` component state.

```

// src/components/Tasks/TaskItem.tsx
const TaskItem = ({
  name,
  done,
  id,
  toggleDone,
  handleDelete,
}: {
  name: string;
  done: boolean;
  id: string;
  toggleDone: (id: string, done: boolean) => void;
  handleDelete: (id: string) => void;
}) => {
  return (
    <div className="flex justify-between bg-white p-1 px-3 rounded-sm gap-4">
      <div className="flex gap-2 items-center">
        <input
          type="checkbox"
          checked={done}
          onChange={() => toggleDone(id, !done)}
        />
        {name}
      </div>
      <button
        className="bg-green-200 hover:bg-green-300 rounded-lg p-1 px-3"
        type="button"
        onClick={() => handleDelete(id)}
      >
        Delete
      </button>
    </div>
  )
}

```

```
    );  
  };  
  
  export default TaskItem;
```

And here's our final app after we add a few tasks!

The screenshot shows a To-Do application with a green theme. It features a 'Summary' section with three cards: 'Total' with a value of 3, 'To do' with a value of 2, and 'Done' with a value of 1. Below this is a section for adding a new task, consisting of a text input field and an 'Add task' button. At the bottom is a 'Tasks' section displaying a list of three tasks. The first task, 'This is my next task', is checked and has a 'Delete' button. The other two tasks, 'And another one' and 'And the last one', are unchecked and also have 'Delete' buttons.

Summary	
Total	3
To do	2
Done	1

Enter your next task:

Add task

Tasks	
<input checked="" type="checkbox"/> This is my next task	Delete
<input type="checkbox"/> And another one	Delete
<input type="checkbox"/> And the last one	Delete

If you're coding along, you can deploy your own app by following [these instructions](#).

You can find the repo with all of the code we went through [here](#), and the live version of the app [here](#).

## Final words

In conclusion, building a to-do list app can be a great way to learn and solidify our understanding of React and its principles. By breaking down the process into small steps and following best practices, we can create a functional app in a relatively short amount of time.

We've covered:

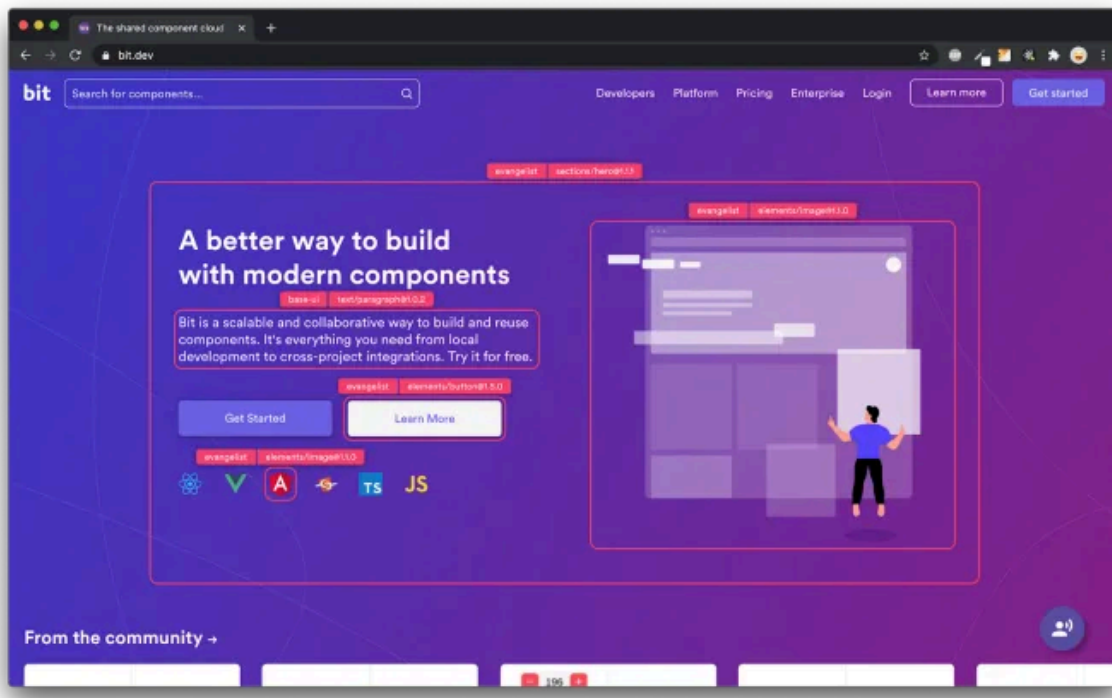
- the key concepts of components, state, and inverse data flow.
- the design and architecture of the app.
- best practices such as the single-responsibility principle

By following the steps outlined in this guide, you should now have a solid understanding of how to build a simple React app and be able to apply it to your own projects. You could also push your React project out to [Bit](#) for reusing in future or for sharing with teammates.

Happy coding!

**Build Apps with reusable components, just like Lego**





**Bit's** open-source tool help 250,000+ devs to build apps with components.

Turn any UI, feature, or page into a **reusable component** — and share it across your applications. It's easier to collaborate and build faster.

→ **Learn more**

Split apps into components to make app development easier, and enjoy the best experience for the workflows you want:

→ **Micro-Frontends**

→ **Design System**

→ **Code-Sharing and reuse**

→ **Monorepo**

**Learn more**

**How We Build Micro Frontends**

Building micro-frontends to speed up and scale our web development process.

[blog.bitsrc.io](https://blog.bitsrc.io)

**How we Build a Component Design System**

Building a design system with components to standardize and scale our UI development process.

[blog.bitsrc.io](https://blog.bitsrc.io)

**How to reuse React components across your projects**

Finally, you completed the task of creating a fantastic input field for the form in your app. You are happy with the...

[bit.dev](https://bit.dev)

**5 Ways to Build a React Monorepo**

Build a production-grade React monorepo: From fast builds to code-sharing and dependencies.

[blog.bitsrc.io](https://blog.bitsrc.io)

**How to Create a Composable React App with Bit**

In this guide, you'll learn how to build and deploy a full-blown composable React application with Bit. Building a...

[bit.dev](https://bit.dev)

[React](#)[Typescript](#)[JavaScript](#)[Software Development](#)[State Management](#)



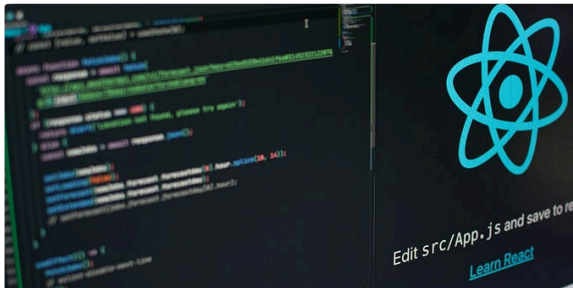
## Written by Eduardo Motta de Moraes

[Follow](#)

28 Followers · Writer for Bits and Pieces

Software developer

### More from Eduardo Motta de Moraes and Bits and Pieces



 Eduardo Motta de Moraes in Bits and Pieces

### How to use Firestore with Redux in a React application

I'll show you how to efficiently fetch data from Firestore and seamlessly add it to your Redu...

8 min read · Jan 9, 2023



233




4



...



 Danusha Navod in Bits and Pieces

### 7 Node.js Design Patterns Every Developer Should Know

Explore the Facade, Adapter, Singleton, Prototype, Builder, Proxy and Factory for...

10 min read · Feb 7, 2024



1.3K



7



...



Ashan Fernando in Bits and Pieces

## 5 Essentials for Modern Frontend Architecture

Quick insights to better design your frontend architecture

11 min read · Feb 22, 2024



1.1K



7



...



Eduardo Motta de Moraes in Bits and Pieces

## User Authentication with JavaScript and Express: A Practic...

Level up your web development skills: Explore secure user authentication in JavaScript with...

11 min read · Jul 18, 2023



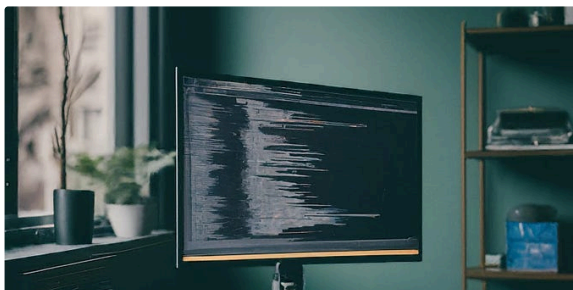
2



...

[See all from Eduardo Motta de Moraes](#)[See all from Bits and Pieces](#)

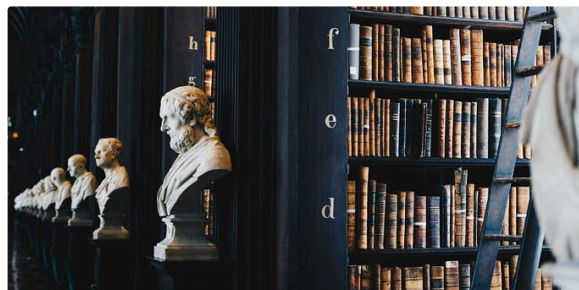
## Recommended from Medium



Joao Paulo C. Marra

## React Clean Architecture: A Guide for Scalable, Testable Apps

How to structure your React projects for long-term success



Mevlüt Can Tuna

## Building a Modern React Component Library: A Guide with...

Photo by Giammarco Boscaro on Unsplash

3 min read · Feb 22, 2024

3 min read · Jan 5, 2024

72

1

137

10

Lists



**General Coding Knowledge**  
20 stories · 1225 saves



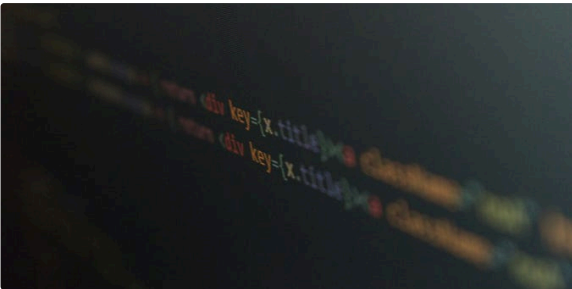
**Stories to Help You Grow as a Software Developer**  
19 stories · 1067 saves



**Coding & Development**  
11 stories · 614 saves



**Good Product Thinking**  
11 stories · 581 saves



Alex Nedopaka

**Setup a React Vite project with TypeScript, Prettier & Vitest [2024]**

Easy instructions for quick setup

7 min read · Feb 13, 2024

58

1



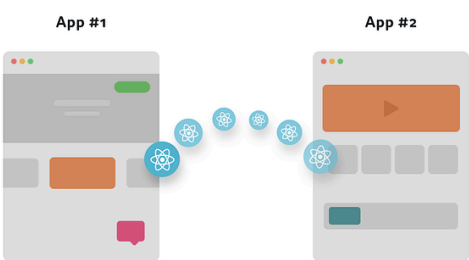
Dagang Wei

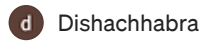
**Having Fun with TypeScript: Zod**

Schema Validation Made Easy

3 min read · Mar 12, 2024

3





Dishachhabra

## react

react anytime react updates the dom it is called a re render

1 min read · Feb 2, 2024



56



...



Suresh Bhatt

## Mastering React's Re-usability for Beginners: A Guide to Conditional...

While technically any imported component is being reused, BUT true re-usability goes...

6 min read · Apr 4, 2024



5



...

See more recommendations