TypeScript

[**Step 1 — Starting the TypeScript Project**](https://www.digitalocean.com/community/tutorials/typescript-new-project#step-1-starting-the-typescript-project)

To begin your TypeScript project, you will need to create a directory for your project:

mkdir typescript-project

Now change into your project directory:

$ cd typescript-project

$ npm init

Install TypeScript:

npm i -D typescript

It is important to include the --save-dev flag because it saves TypeScript as a development dependency. This means that TypeScript is required for the development of your project.

Initialize your TypeScript project

npx tsc --init

Install Node JS types

npx i -D @types/node

npm also includes a tool called npx, which will run executable packages. npx allows us to run packages without having to install them globally.

The tsc command is used here because it is the built-in TypeScript compiler. When you write code in TypeScript, running tsc will transform or compile your code into JavaScript.

Using the --init flag in the above command will initialize your project by creating a tsconfig.json file in your typescript-project project directory. This tsconfig.json file will allow you to configure further and customize how TypeScript and the tsc compiler interact. You can remove, add, and change configurations in this file to best meet your needs.

There will be many options, most of which are commented out:

typescript-project/tsconfig.json

{

"compilerOptions": {

"target": "es2020",

"module": "commonjs", // or "Node16"

"moduleResolution": "node", // or "Node16"

"rootDir": "./src",

"outDir": "./dist",

"esModuleInterop": true,

"forceConsistentCasingInFileNames": true,

"strict": true,

"skipLibCheck": true,

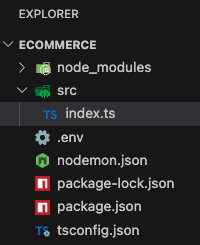
},

"include": [

"src/\*\*/\*"

]

}



You can customize your TypeScript configuration through the tsconfig.json file. For instance, you might consider uncommenting the outDir entry and setting it to "./build", which will put all of your compiled TypeScript files into that directory.

With TypeScript installed and your tsconfig.json file in place, you can now move on to coding your TypeScript app and compiling it.

**Note:** Step 3 below will replace many of your configurations with sensible defaults, but these changes will get you started right away.

[**Step 2 — Compiling the TypeScript Project**](https://www.digitalocean.com/community/tutorials/typescript-new-project#step-2-compiling-the-typescript-project)

You can now begin coding your TypeScript project. Open a new file named index.ts in your editor. Write the following TypeScript code in index.ts:

typescript-project/index.ts

const world = 'world';

export function hello(who: string = world): string {

return `Hello ${who}! `;

}

With this TypeScript code in place, your project is ready to be compiled. Run tsc from your project’s directory:

npx tsc

Copy

You will notice that the compiled JavaScript index.js file and the index.js.map sourcemap file have both been added to the build folder if you specified that in the tsconfig.js file.

Open index.js and you will find the following compiled JavaScript code:

typescript-project/build/index.js

"use strict";

Object.defineProperty(exports, "\_\_esModule", { value: true });

exports.hello = void 0;

const world = 'world';

function hello(who = world) {

return `Hello ${who}! `;

}

exports.hello = hello;

Running the TypeScript compiler every time you make a change can be tedious. To fix this, you can put the compiler in watch mode which will recompile your code every time changes are made.

You can activate watch mode using the following command:

npx tsc -w

You’ve learned how the TypeScript compiler works, and you are now able to successfully compile your TypeScript files. You can take your TypeScript projects to the next level by introducing a linter into your workflow.

### Compilation

tsx and ts-node are both tools for directly executing TypeScript files in Node.js without a prior compilation step, but they differ in their approach and features:

#### ts-node

* **Type Checking:**

By default, ts-node performs type checking during execution. This means it leverages the TypeScript compiler to ensure type safety, which can be beneficial for catching errors early in development.

* **Configuration:**

It often requires a tsconfig.json file for configuration, allowing for fine-grained control over TypeScript compilation options.

* **Performance:**

While it offers type safety, the type checking process can introduce a performance overhead, making it slower for rapid development or scripts where type checking is handled elsewhere (e.g., by an IDE or separate build step).

npm i -D ts-node

#### tsx

* **Type Checking:**

tsx prioritizes speed by skipping type checking during execution. It focuses solely on transpilation, converting TypeScript syntax into valid JavaScript that Node.js can run.

* **Performance:**

It utilizes esbuild for fast transpilation, resulting in significantly quicker execution times, particularly for larger projects or frequently run scripts.

* **Configuration:**

tsx aims for a more "out-of-the-box" experience, often working without explicit configuration files, though it does support tsconfig.json paths.

* **Use Cases:**

Ideal for scenarios where rapid iteration and execution speed are critical, and type checking is either handled by an IDE, a separate build process, or not deemed necessary for the specific script.

Install TSX to compile and run the TS files

npm i -D tsx

"scripts": {

"dev": "tsx src/index.ts"

},

**In Summary:**

**Choose ts-node if:**

* You require on-the-fly type checking during execution.
* You need extensive control over TypeScript compilation through tsconfig.json.

**Choose tsx if:**

* You prioritize fast execution and rapid development cycles.
* Type checking is handled by other tools in your workflow (IDE, linter, build step).

You prefer a simpler setup and less configuration.

#### Hot Reloading

$ npm i -D nodemon

nodemon.json

{

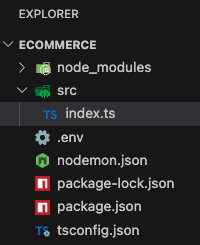
"watch": ["src" ],

"ext": "js,ts",

"ignore": [ "node\_modules", "dist"],

"exec": "npx ts-node ./src/index.ts"

}



package.json

"scripts": {

"start": "npx nodemon",

### Environment Variables

$ npm i -D dotenv

import { configDotenv } from "dotenv";

configDotenv()

src/index.ts

import { configDotenv } from "dotenv";

import express from "express";

configDotenv()

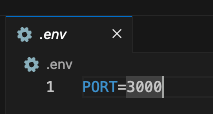
const app = express();

app.use(express.json())

app.listen(process.env.PORT, () => {

console.log(`Server is running on port ${process.env.PORT}`);

})



## Aliasing

type Props = {

icon: LucideIcon,

label: string,

href: string,

}

export default function NavButton({

icon: Icon,

label,

href

}: Props) {

return (

<Button>

{href? (<Link href={href}><Icon /></Link>):<Icon />}

</Button>

)

}

In the destructuring assignment here, icon: Icon means that the icon prop passed to NavButton will be renamed to Icon. This is a TypeScript feature known as **aliasing**.

## ShadCn – Button “asChild” attribute

### The Purpose of asChild:

The asChild prop is used to allow the Button component to **render as a different element** rather than the default <button> HTML element. This is part of a pattern commonly referred to as **polymorphism** in React components. When you use asChild, you're telling the Button component to render the children (in this case, the Link and Icon) as if they were the root element, rather than wrapping them in a <button>.

In other words, asChild lets you **customize the wrapper** around the child components. This is particularly useful if you want to turn a component that typically renders as a <button> into another element, like a <Link>, while still preserving the button's styling and behavior.

### How asChild Works:

* Without asChild, a Button component would render as a <button> element.

Example:

<Button>Click me</Button>

// This will render as:

// <button class="..." ...>Click me</button>

* With asChild, the Button component can be used to wrap any other element (like a Link, div, a, etc.), while still applying the button's styles and logic.

Example:

<Button asChild>

<Link href="/home">Go to Home</Link>

</Button>

// This will render as:

// <Link href="/home" class="...">Go to Home</Link>

### In Your Case:

<Button variant={'ghost'} size={'icon'} aria-label={label} title={label} className="rounded-full" asChild>

{href ? (<Link href={href}><Icon /></Link>) : <Icon />}

</Button>

Here, the Button is acting as a wrapper for either:

* A Link (when href is provided), or
* Just the Icon (when no href is provided).

When asChild is applied, the Button doesn't render as a <button>, but instead renders whatever element is passed as its child. In your case, it's either:

* <Link href={href}><Icon /></Link>, or
* <Icon />.

This ensures that the Button component's styling and behavior (like size, variant, etc.) apply to the child component (the Link or Icon), but without wrapping it in an actual <button> element.

### Why Use asChild?

The asChild prop is useful in scenarios where you need to:

1. **Use a different semantic element**: For example, if you want to use a Link component (which renders an <a> tag) instead of a <button>, but still apply button-like styling and behavior.
2. **Maintain button behavior**: You may still want to keep the accessibility and interaction behavior associated with a button (like keyboard navigation, focus states, etc.) but render a different HTML element.

In summary, asChild allows the Button component to be flexible, letting it render other elements (like Link or Icon) while still applying the button's styles and functionality. It effectively turns the Button into a higher-order component that doesn't dictate the HTML element type it renders.