1. Creating a React project with Vite that uses TypeScript

- Move your files into it from the React-REST Axios lab.

2. Setting Up TypeScript Types for Objects and Props

- TypeScript provides **static typing**, which prevents errors before running the app.
- It improves **developer experience** by offering better autocompletion.
- It makes your codebase **easier to maintain**.

Step 1.1: Define a `Product` Type

You can add this interface under the imports in Products.tsx, or create a `types/` directory for all types:

```
/src
  /types
    Product.ts
```

Create a TypeScript interface inside `src/Product.tsx` or `src/types/Product.ts`. If you put it in a separate file, you will import it with `import { Product } from '../types/Product';`

```
export interface Product {
  id: string; // Unique identifier
  name: string; // Product name
  description: string; // Short text about the product
  price: string; // Can store numbers as strings for easy
formatting
  stock: number; // Number of items available
}
```

Benefits:

- We define `id` as a `string` because it may come from databases like MongoDB (which uses string `_id`).

- `price` is a `string` to avoid floating point issues (e.g., `"10.99"` instead of `10.99`).
- `stock` is a `number` because it's an integer.

Step 1.2: Define Props for Each Component

Each component should **only accept properly typed props** to ensure consistency.

Inside `Product.tsx`

```
import { Product } from '../types/Product';

interface ProductProps {
  product: Product; // This component receives a single
product
  onBack: () => void; // Function prop with no arguments,
used for navigation
}
```

Inside `ProductForm.tsx`

```
import { Product } from '../types/Product';

interface ProductFormProps {
  onAddProduct: (product: Product) => void; // Function
that receives a new product
  onCancel: () => void; // Function to cancel adding a
product
}
```

Inside `ProductList.tsx`

```
import { Product } from '../types/Product';

interface ProductListProps {
  onSelectProduct: (product: Product) => void; // Function
to select a product
}
```

Benefits:

- Helps **document** each component's expected inputs.
- Prevents passing incorrect types, such as a `number` when a `string` is expected.
- Provides **auto-suggestions** in VSCode or other IDEs.

2. Converting `App.jsx` to `App.tsx`

Benefits:

- `.tsx` files support **TypeScript + JSX syntax**.
- Helps ensure all React components receive correctly typed props.

Step 2.1: Modify `App.tsx`

```
// src/App.tsx
import React, { useState } from 'react';
import ProductList from './components/ProductList';
import Product from './components/Product';
import ProductForm from './components/ProductForm';
import { Product as ProductType } from './types/
Product'; // Rename `Product` to avoid conflicts
import './App.scss';

const App: React.FC = () => {
  // State for selected product
  const [selectedProduct, setSelectedProduct] =
useState<ProductType | null>(null);

  // State to track if the form is open
  const [isAdding, setIsAdding] = useState<boolean>(false);

  // Function to handle selecting a product
  const handleSelectProduct = (product: ProductType) => {
    setSelectedProduct(product);
  };

  // Function to go back to the product list
  const handleBack = () => {
    setSelectedProduct(null);
    setIsAdding(false);
  };
```

```
  // Function to handle adding a new product
  const handleAddProduct = (product: ProductType) => {
    setIsAdding(false);
    setSelectedProduct(null);
  };

  return (
    <div className="App">
      {isAdding ? (
        <ProductForm onAddProduct={handleAddProduct}
onCancel={handleBack} />
      ) : selectedProduct ? (
        <Product product={selectedProduct}
onBack={handleBack} />
      ) : (
        <>
          <ProductList
onSelectProduct={handleSelectProduct} />
          <button onClick={() => setIsAdding(true)}>Add
Product</button>
        </>
      )}
    </div>
  );
};

export default App;
```

We changed:

Typed `useState<ProductType | null>`

- Ensures `selectedProduct` is either `null` or a `Product`.
- Prevents errors like `Cannot read properties of null`.

Function Prop Type Consistency

- `handleSelectProduct` accepts a `ProductType` to match `ProductListProps`.

Type-Safe `isAdding` State

- `useState<boolean>(false)` ensures it always holds a boolean.

3. Converting `Product.jsx` to `Product.tsx`

Benefits:

- Ensure the product editing component **only works with valid products**.

Step 3.1: Modify `Product.tsx`

```
// src/components/Product.tsx
import React, { useState } from 'react';
import axios from 'axios';
import { Product as ProductType } from '../types/Product';

interface ProductProps {
  product: ProductType;
  onBack: () => void;
}

const Product: React.FC<ProductProps> = ({ product,
onBack }) => {
  const [productData, setProductData] =
useState<ProductType>(product);

  // Handle input changes safely
  const handleChange = (e:
React.ChangeEvent<HTMLInputElement>) => {
    const { name, value } = e.target;
    setProductData((prevState) => ({
      ...prevState,
      [name]: value,
    }));
  };
```

```
  // Handle saving to API
  const handleSave = async () => {
    try {
      const response = await axios.put(`http://
localhost:5000/products/${productData.id}`, productData);
      console.log('Product updated:', response.data);
      onBack(); // Navigate back after saving
    } catch (error) {
      console.error('Error updating product:', error);
    }
  };

  return (
    <div className="form-container">
      <h2>Edit Product</h2>
      <form>
        <label>Name:</label>
        <input type="text" name="name"
value={productData.name} onChange={handleChange} />

        <label>Description:</label>
        <input type="text" name="description"
value={productData.description} onChange={handleChange} />

        <label>Price:</label>
        <input type="text" name="price"
value={productData.price} onChange={handleChange} />

        <label>Stock:</label>
        <input type="number" name="stock"
value={productData.stock} onChange={handleChange} />

        <button type="button" onClick={handleSave}>Save</
button>
        <button type="button" onClick={onBack}>Back to
list</button>
      </form>
    </div>
  );
};
```

```
export default Product;
```

We changed:

Typed `useState<ProductType>`

- Ensures `productData` follows the expected structure.
- Prevents issues like `undefined` properties.

Typed Event Handlers (`React.ChangeEvent<HTMLInputElement>`)

- Ensures `handleChange` only processes valid `input` elements.

Error Handling in `handleSave`

- Catches API failures so the app doesn't break.

Next:

- **Convert `ProductForm.tsx`** to ensure form inputs are type-safe.
- **Convert `ProductList.tsx`** to correctly fetch products with TypeScript.

Converting CSS to SCSS:

Let's convert `App.css` to `App.scss`

This file makes use of **SCSS features** like:

- **Variables (`$variable-name`)** – Store reusable values.
- **Nesting (`&` and children selectors)** – Make styling more readable.
- **Ampersand (`&`) for Parent Referencing** – Modify styles within the same block.
- **Transitions & Pseudo-classes (`:hover`, `:focus`, `:active`)** – Add animations.
- **Optional: SCSS Mixins & Extends** – Would help reduce repetition.

1. Variables in SCSS

```
$primary-color: #3498db;
```

```
$secondary-color: #2ecc71;
$font-family: 'Arial', sans-serif;
$border-radius: 8px;
```

- **What SCSS is doing:**
  - It defines **variables** that can be reused throughout the file.
  - Instead of hardcoding values like `#3498db` everywhere, we store them in variables (`$primary-color`).
  - This makes it **easy to change styles across the whole app** by modifying a single value.

2. Nesting in SCSS (`.App` example)

```
.App {
  display: flex;
  flex-direction: column;
  justify-content: center;
  align-items: center;
  height: 100vh;
  background-color: #f8f8f8;
  font-family: $font-family;
  color: #333;
  transition: background-color 0.3s ease;

  &.dark-mode {
    background-color: #2c3e50;
    color: white;
  }
}
```

- **What SCSS is doing:**
  - `.App` is the **parent selector**.
  - Inside `.App`, we use `&.dark-mode`:
    - The `&` **refers to `.App` itself**.

    This results in:

    ```
    .App.dark-mode {
      background-color: #2c3e50;
    ```

```
    color: white;
  }
```

- This makes it **easier to modify states** like dark mode inside `.App` without repeating selectors.

3. Using `&` for Pseudo-classes (`button` example)

```
button {
  padding: 10px 20px;
  font-size: 16px;
  background-color: $primary-color;
  color: white;
  border: none;
  border-radius: $border-radius;
  cursor: pointer;
  transition: all 0.3s ease;

  &:hover {
    background-color: $secondary-color;
    transform: translateY(-3px);
  }

  &:active {
    transform: translateY(1px);
  }

  &:focus {
    outline: none;
  }
}
```

- **What SCSS is doing:**
  - SCSS **nests pseudo-classes (`:hover`, `:active`, `:focus`) inside `button`**.

  The `&:hover` is the same as writing:

```
button:hover {
```

```
    background-color: #2ecc71;
    transform: translateY(-3px);
  }
```

- The `&` keeps it **scoped within `button`**, so we don't need to repeat `button` everywhere.
- The hover and active states **add animations and interactivity**.

4. Styling `.form-container` with SCSS Nesting

```
.form-container {
  background-color: white;
  padding: 20px;
  border-radius: $border-radius;
  box-shadow: 0 4px 12px rgba(0, 0, 0, 0.1);
  max-width: 400px;
  width: 100%;
  transition: transform 0.3s ease;

  &:hover {
    transform: scale(1.03);
  }

  label {
    display: block;
    margin: 10px 0 5px;
    font-weight: bold;
  }

  input[type='text'],
  input[type='number'] {
    width: 100%;
    padding: 10px;
    border: 1px solid #ddd;
    border-radius: $border-radius;
    margin-bottom: 15px;
    transition: all 0.3s ease;

    &:focus {
```

```
      border-color: $primary-color;
      box-shadow: 0 0 5px rgba(52, 152, 219, 0.5);
    }
  }
}
```

- **What SCSS is doing:**
  - `.form-container` has:
    - **A background color** (`white`).
    - **A box shadow** (for a 3D effect).
    - **A transform animation** when hovered (`scale(1.03)`).
  - Inside `.form-container`, SCSS **nests `label` and `input`**:
    - This means the `label` and `input` styles **only apply inside `.form-container`**.
    - The input fields change border color and glow when focused.

5. Button Styling Inside `.form-container`

```
button {
  width: 100%;
  margin-top: 15px;
  background-color: $secondary-color;
}
```

- **What SCSS is doing:**
  - Since this `button` is **inside `.form-container`**, it **only affects buttons within forms**.
  - It makes the button full-width (`width: 100%`) and uses the green color (`$secondary-color`).
  - This prevents accidental styling of other buttons.

SCSS Advantages Over Regular CSS

1. Better Readability with Nesting

Instead of writing:

```
.form-container label {
  display: block;
}
```

```
.form-container input[type="text"] {
  width: 100%;
}
```

- SCSS **nests these rules**, making it more readable.

2. Easy Theme Changes with Variables

- If we wanted to change the theme color, we just update:

```
$primary-color: red;
```

- This updates **all primary color elements** automatically.

3. Scoped Selectors Using `&`

- Instead of writing:

```
.App.dark-mode { background-color: black; }
```

- SCSS allows us to **keep `.dark-mode` inside `.App`** using `&`.

How SCSS Compiles to CSS

The SCSS:

```
/*scss*/

button {
  background-color: $primary-color;

  &:hover {
    background-color: $secondary-color;
  }
}
```

Compiles into standard CSS:

```
/* CSS*/
button {
  background-color: #3498db;
}

button:hover {
  background-color: #2ecc71;
}
```

- The SCSS **simplifies organization** while keeping CSS **lightweight**.

SCSS Features Used in `App.scss`

Feature	What It Does
Variables (`$var`)	Store reusable colors, fonts, and sizes.
Nesting (`selector { child {}`)	Keep related styles together.
Parent Referencing (`&`)	Modify styles based on state changes (e.g., `:hover`).
Pseudo-classes (`:hover`, `:focus`)	Add interactivity.
Transitions (`transition: all 0.3s ease;`)	Smooth animations.