EXPENSE TRACKER - REACT - VITE - TYPESCRIPT

main.tsx

```tsx
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App";
import "bootstrap/dist/css/bootstrap.css";
import "./index.css";

ReactDOM.createRoot(document.getElementById("root") as HTMLElement).render(
 <React.StrictMode>
   <App />
 </React.StrictMode>
);
```

index.html

```html
<!DOCTYPE html>
<html lang="en">
 <head>
   <meta charset="UTF-8" />
   <link rel="icon" type="image/svg+xml" href="/currency-dollar.svg" />
   <link rel="stylesheet" href="./src/index.css">
   <meta name="viewport" content="width=device-width, initial-scale=1.0" />
   <title>Expense Tracker</title>
 </head>
 <body>
   <div id="root"></div>
   <script type="module" src="/src/main.tsx"></script>
 </body>
</html>
```

index.css

```
html,
body {
 justify-content: center;
 align-items: center;
 margin: 0 auto;
 background-color: #333333 !important;
 font-family: monospace;
 font-size: 13px;
}

.container {
 width: 650px !important;
 flex-direction: column;
 justify-content: center;
 align-items: center;
 background-color: #222222;
 margin: 5px auto;
 border-radius: 8px;
 box-shadow: 2px 2px 5px 5px deeppink;
 padding: 10px;
 text-align: center;
 box-sizing: border-box;
}

table {
 width: 95%;
 border-collapse: collapse;
 border-spacing: 0px;
 border-top: none;
 border-bottom: 1px solid white;
 border-top: 1px solid #222222;
 margin: 0 auto;
```

```
}

table td {
 padding: 8px;
 font-size: 15px;
 color: white;
 border: 1px #222222;
}

table th {
 text-align: left;
 font-size: 17px;
 color: cyan;
 font-weight: normal;
 border: 1px #222222;
}

.form_box {
 background-color: cyan;
 border-radius: 8px;
 padding: 10px 10px;
 color: #222222;
 font-size: 16px;
 font-weight: bold;
 flex-direction: column;
 width: 350px;
 margin: 0 auto;
 margin-bottom: 20px;
 border: white;
 font-family: monospace;
 display: flex;
 justify-content: center;
 align-items: center;
}
```

```
form {
 justify-content: center;
 font-size: 16px;
}

.main_title {
 font-size: 33px;
 font-family: monospace;
 text-shadow: 2px 2px 5px deeppink;
 text-align: center;
 color: yellow;
 margin-bottom: 20px;
}

.heading_name {
 font-size: 23px;
 color: yellow;
 font-family: monospace;
 margin-bottom: 5px;
 text-align: center;
}

.btn:hover {
     background-color: yellow !important;
     cursor: pointer;
}


/* Styles for mobile screens */
@media (max-width: 650px) {
 html,
   body {
     display: flex;
```

```
    font-size: 10px;
  }

.container {
  width: 400px !important;
  margin: 10px 10px;
  border-radius: 5px;
  box-shadow: 2px 2px 5px 5px deeppink;
  padding: 5px;
  }

table {
  min-width: 350px;
  margin: 0 auto;
  }

table th {
  font-size: 13px;
}

table td {
  font-size: 11px;
}

.heading_name {
 font-size: 19px;
  margin-bottom: 10px;
  text-align: center;
}
}
```

App.tsx

```
import React, { useState } from "react";
import ExpenseList from "./components/ExpenseList";
import ExpenseFilter from "./components/ExpenseFilter";
import ExpenseForm from "./components/ExpenseForm";
import "./index.css";
import categories from "../categories";
import styled from "styled-components";

function App() {
 const [selectedCategory, setSelectedCategory] = useState("");

 const [expenses, setExpenses] = useState([
   {
     id: 1,
     description: "Electricity Bill",
     cost: 23,
     category: "Utilities",
   },
   {
     id: 2,
     description: "Birthday dinner items",
     cost: 14,
     category: "Groceries",
   },
   {
     id: 3,
     description: "Movie tickets",
     cost: 40,
     category: "Entertainment",
   },
   {
     id: 4,
```

Here, the App component is defined as a function. It has two state variables - selectedCategory and expenses.

selectedCategory is initially set to an empty string, which means no category is selected by default. It will be updated by the ExpenseFilter component.

expenses is an array of objects that represent the user's expenses. This is also initialized with some sample data.

```
    description: "Mother's Day gift",
    cost: 120,
    category: "Other",
  },
]);

const visibleExpenses = selectedCategory
  ? // if user has selected a category, return that category, otherwise return all
    expenses.filter((e) => e.category === selectedCategory)
  : expenses;

return (
  <>
    <div className="main_title">Expense Tracker</div>
    <div className="container">
      <div>
        <div className="heading_name">Add an Expense</div>
        <ExpenseForm
          onSubmit={(newExpense) =>
            setExpenses([
              ...expenses,
              { ...newExpense, id: expenses.length + 1 },
            ])
          }
        />
        <div className="heading_name">Current Expenses</div>
        <ExpenseFilter
          onSelectCategory={(category) => setSelectedCategory(category)}
        />
        <ExpenseList
          // filter and show only expenses that have not been deleted
          expenses={visibleExpenses}
          onDelete={(id) => setExpenses(expenses.filter((e) => e.id !== id))}
        />
```

visibleExpenses is a new variable that filters the expenses array based on whether the user has selected a specific category or not. If a category is selected, only expenses belonging to that category will be shown, otherwise all expenses will be shown.

The function returns JSX that renders the Expense Tracker app, which consists of a title ( ), a container div ( ) that wraps the components related to adding and filtering expenses, and a set of nested components to handle input and display of expense data.

```
        </div>
        ;
      </div>
    </>
  );
}

export default App;
```

ExpenseList.tsx

```
import styled from "styled-components";
import "../index.css";

const Box = styled.div`
 margin: 0 auto;
 border: white;
 font-family: monospace;
 max-width: 600px;
`;

const RemoveCol = styled.td`
 text-align: center;
 vertical-align: middle;
`;

const RemoveButton = styled.button`
 color: white;
 display: inline-block !important;
 margin: 4px 4px;
`;

const Total = styled.tr`
```

The component defines a set of styled components that define how different elements of the expense list should look.

```
 border-bottom: none;
 background-color: #444444;
`;

const Nothing = styled.div`
 color: white;
 text-align: center;
 font-family: monospace;
 font-size: 18px;
`;

const DescCol = styled.td`
 text-align: left;
 margin-right: 20px;
`;

const CostCol = styled.td`
 text-align: right;
 margin-right: 30px;
`;

const CatCol = styled.td`
 text-align: left;
 margin-left: 20px;
`;

// definition of an Expense object
interface Expense {
 id: number;
 description: string;
 cost: number;
 category: string;
}
```

The component defines an interface for an Expense object, which includes an id, description, cost and category.

```
// expenses will be an array of Expense objects
interface Props {
 expenses: Expense[];
 onDelete: (id: number) => void;
}

const ExpenseList = ({ expenses, onDelete }: Props) => {
 // if there are no expenses to be shown, show no list and a message
 if (expenses.length === 0)
   return (
     <Nothing>
       There are no expenses to show. <br /> Have a great day!
     </Nothing>
   );
 return (
   <Box>
     <table className="table-bordered">
       <thead>
         <tr>
           <th>Description</th>
           <th>Cost</th>
           <th>Category</th>
           <th>Remove</th>
         </tr>
       </thead>
       <tbody>
         {/* mapping expenses to their own rows in the expenses list */}
         {expenses.map((expense) => (
           <tr key={expense.id}>
             <DescCol>{expense.description}</DescCol>
             <CostCol>${expense.cost.toFixed(2)}</CostCol>
             <CatCol>{expense.category}</CatCol>
             <RemoveCol>
               <RemoveButton
```

The component defines another interface for props, which includes an array of Expense objects and a function for deleting an expense by id.

This code defines a React component named ExpenseList which receives two props expenses and onDelete.

The component then uses these props to render a table of expenses. If there are no expenses to render, it instead displays a message.

ExpenseList renders a table of expenses with the following columns: Description, Cost, Category, and Remove. It first checks if the expenses array is empty, in which case it displays a message indicating there are no expenses to show.

The expenses array is then mapped over to display each expense as a row in the table. Each expense object has an id, description, cost, and category property, which are displayed in the corresponding columns of the table.

The cost property is formatted as a dollar amount with two decimal places using the

```
                  className="btn btn-outline-light"
                  onClick={() => onDelete(expense.id)}
                >
                  X
                </RemoveButton>
              </RemoveCol>
            </tr>
          ))}
        </tbody>
        <tfoot>
          <Total>
            <td>Total</td>
            <td>
              $
              {expenses
                // acc keeps track of the running total, as costs are iteratively added
                .reduce((acc, expense) => expense.cost + acc, 0)
                .toFixed(2)}
            </td>
            <td>{}</td>
            <td>{}</td>
          </Total>
        </tfoot>
      </table>
    </Box>
  );
};

export default ExpenseList;
```

toFixed() method. The onDelete function is called when the remove button is clicked to remove the corresponding expense from the list.

Finally, the total expense cost is calculated and displayed in a row at the bottom of the table with the Total label. The styled-components library is used to apply CSS styling rules to various HTML elements, including the table cells and the table itself.

To calculate the total cost, it uses the reduce() method on the expenses array. reduce() takes two arguments: a callback function and an initial value. The callback function is executed on each element in the array and takes two arguments: an accumulator and the current element. The initial value of the accumulator is provided as the second argument to reduce(). In this case, the initial value is 0.

The callback function adds the cost of the current expense to the accumulator and returns the new value of the accumulator. The toFixed() method is called on the final value to format it as a string with two decimal places.

ExpenseFilter.tsx

```
import categories from "../../categories";
import "../index.css";
import styled from "styled-components";

interface Props {
 onSelectCategory: (category: string) => void;
}

const Filter = styled.div`
 display: flex !important;
 justify-content: center !important;
 align-items: center !important;
 margin-bottom: 15px;
 font-family: monospace;
 width: 100%;
`;

const FilterSelect = styled.select`
 width: 250px;
 font-size: 15px;
`;

const ExpenseFilter = ({ onSelectCategory }: Props) => {
 return (
   <Filter>
     <FilterSelect
       className="form-select"
       onChange={(event) => onSelectCategory(event.target.value)}
     >
       <option value="">View All</option>
       {categories.map((category) => (
         <option key={category} value={category}>
```

Define the Props interface with one property, onSelectCategory, which is a function that takes a string argument and returns nothing.

Define the Filter styled component that wraps the FilterSelect component. This component uses the flex display property to center and align the child elements.

Define the FilterSelect styled component which is a select element with a width of 250px and a font size of 15px.

Define the ExpenseFilter component which takes the onSelectCategory prop and returns a Filter component wrapping a FilterSelect component.

The ExpenseFilter component returns the Filter and FilterSelect components wrapped around the categories array. The onChange event listener is added to the FilterSelect component to call the

```
            {category}
          </option>
        ))}
      </FilterSelect>
    </Filter>
  );
};

export default ExpenseFilter;
```

onSelectCategory function when the user selects a category from the dropdown list.

That's it! The ExpenseFilter component returns a styled filter dropdown menu with the list of categories, allowing the user to filter the expenses by category.

ExpenseForm.tsx

```
import categories from "../../categories";
import { z } from "zod";
import { useForm } from "react-hook-form";
import { zodResolver } from "@hookform/resolvers/zod";
import styled from "styled-components";
import "../index.css";

const AddButton = styled.button`
 color: yellow;
 background-color: deeppink;
 font-size: 16px;
 font-weight: bold;
 display: inline-block !important;
 margin: 4px 8px;
 width: 300px;
 margin: 0 auto;
`;

const ExpenseInput = styled.input`
 width: 300px;
 margin: 0 auto;
```

```
 font-size: 13px;
`;

const SelectCategory = styled.select`
 width: 300px;
 margin: 0 auto;
`;

const ErrorMessage = styled.p`
 color: deeppink;
`;

// setting up Zod validation
const schema = z.object({
 description: z
   .string()
   .min(3, {
     message: "Must contain 3+ characters.",
   })
   .max(50),
 cost: z
   .number({
     invalid_type_error: "Cost required.",
   })
   .min(0.01)
   .max(100_000),
 category: z.enum(categories, {
   errorMap: () => ({ message: "Category required." }),
 }),
});

type ExpenseFormData = z.infer<typeof schema>;
```

The code defines a schema for validating the form data using the Zod library. The schema defines three fields: description, cost, and category. The description field is a string that must be between 3 and 50 characters in length. The cost field is a number that must be greater than 0.01 and less than 100,000. The category field is an enum that must match one of the categories imported from "../../categories". If any of the fields fail validation, an error message will be displayed.

The code defines an interface for the form data called ExpenseFormData, which is inferred from the Zod schema.

```
interface Props {
 onSubmit: (data: ExpenseFormData) => void;
}

const ExpenseForm = ({ onSubmit }: Props) => {
 const {
   register,
   handleSubmit,
   reset,
   formState: { errors },
 } = useForm<ExpenseFormData>({ resolver: zodResolver(schema) });

 return (
   <div className="form_box">
     <form
       onSubmit={handleSubmit((data) => {
         onSubmit(data);
         reset();
       })}
     >
       <div className="mb-3">
         <label htmlFor="description" className="form-label">
           Description:
         </label>
         <ExpenseInput
           {...register("description")}
           id="description"
           type="text"
           className="form-control"
         />
         {errors.description && (
           <ErrorMessage>{errors.description.message}</ErrorMessage>
         )}
       </div>
```

The code defines an interface for the props called Props, which contains an onSubmit function that takes in the form data.

The code defines a functional component called ExpenseForm that takes in the Props interface and returns a form. The form uses useForm from "react-hook-form" to manage form state and validation. The form is wrapped in a styled div with class "form_box".

The form has three input fields: description, cost, and category. The description and cost fields use the ExpenseInput styled component, while the category field uses the SelectCategory styled component. Each input field is registered with the useForm hook using the {...register} syntax.

The useForm hook is used to register the form fields so that their values can be captured and validated when the form is submitted. The register function returned by the hook is used to register each field with the form, along with any necessary validation rules.

The register function is then passed to each component as props using the spread operator ({...register}). This registers the form field with the useForm hook and

```
<div className="mb-3">
  <label htmlFor="cost" className="form-label">
    Cost:
  </label>
  <ExpenseInput
    {...register("cost", { valueAsNumber: true })}
    id="cost"
    type="number"
    className="form-control"
  />
  {errors.cost && <ErrorMessage>{errors.cost.message}</ErrorMessage>}
</div>
<div className="mb-3">
  <label htmlFor="category" className="form-label">
    Category:
  </label>
  <SelectCategory
    {...register("category")}
    id="category"
    className="form-select"
  >
    <option value=""></option>
    <option value="">All</option>
    {categories.map((category) => (
      <option key={category} value={category}>
        {category}
      </option>
    ))}
  </SelectCategory>
  {errors.category && (
    <ErrorMessage>{errors.category.message}</ErrorMessage>
  )}
</div>
<AddButton className="btn btn-light">Add Expense</AddButton>
```

allows the hook to handle validation and other functionality related to the form field.

The form has an "Add Expense" button that uses the AddButton styled component. When clicked, the form data is submitted using handleSubmit from useForm. If the form data passes validation, the onSubmit function from the Props interface is called with the form data.

If the form data fails validation, an error message is displayed using the ErrorMessage styled component.

The form uses the reset function from useForm to clear the form input fields after submission.

The form contains an hr element to separate it visually from other form elements on the page.

```
      </form>
      <hr></hr>
    </div>
  );
};

export default ExpenseForm;
```

categories.ts

```
/*
- list of categories that all components will utilize
- must be set "as const" so that Zod will accept it for validating input categories
- by creating a unique file for categories, it can be imported into each component
   and does not cause complications by needing to dictate the order in which items
   are called or imported.
*/

const categories = [
  "Groceries",
  "Utilities",
  "Entertainment",
  "Other",
] as const;

export default categories;
```

This code is a simple TypeScript array of string literals named `categories`. It is exported as a default export. The `as const` syntax is used to declare a tuple of string literals rather than a regular array. This means that each string in the array is treated as a unique type, rather than all being treated as the generic `string` type.

This array of categories is then imported and used in the `ExpenseForm` component that we previously discussed, specifically in the validation schema for the `category` field. By using `z.enum(categories)`, we ensure that only the valid categories in the `categories` array can be submitted as a value for the `category` field in the form.