实验：待办事项应用程序

估计所需时间： 40分钟

你将学到什么

在这个实验中，你将学习如何使用 React 函数组件和 useState 钩子来制作一个简单的待办事项应用程序。你将学习如何使用 useState 钩子管理状态。实验包括管理 React 状态的基础知识、获取用户输入以及显示动态列表的内容。

学习目标

完成这个实验后，你将能够：

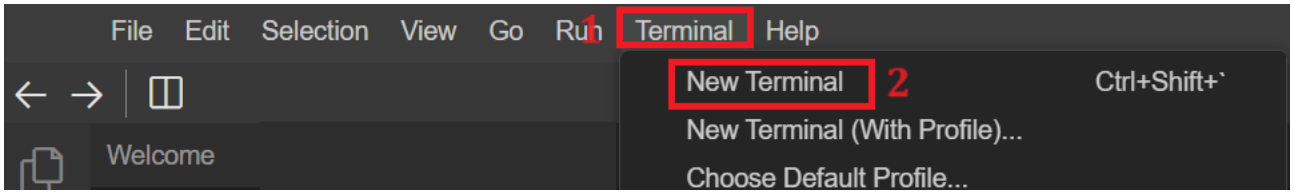
- 使用 React 创建一个简单的待办事项应用程序，并添加标题和相关列表以组织任务。
- 管理待办事项并删除特定的标题和列表，提供基本的 CRUD（创建、读取、更新、删除）功能。
- 使用 React 的 useState 钩子管理待办事项列表的状态，包括标题和相关列表。
- 还可以使用 React 的 useState 钩子动态渲染值。

先决条件

- 基本的 HTML 知识
- 中级的 JavaScript 知识
- 基本的 React 函数组件和使用 useState 钩子进行状态管理的知识

设置环境

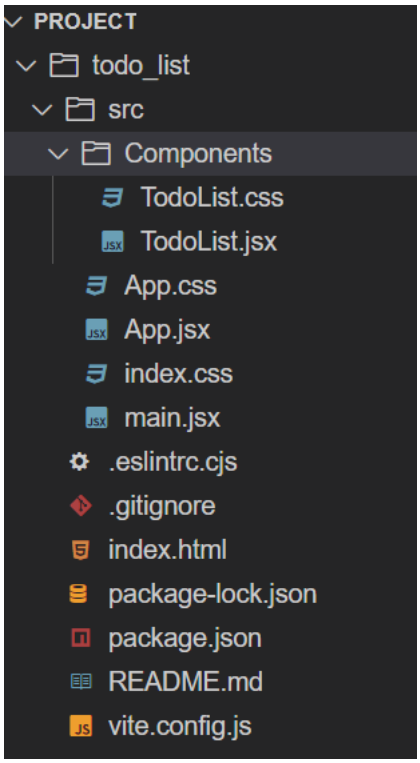
1. 从实验室顶部的菜单中，点击窗口右上角的 **Terminal** 选项卡，如给定截图中编号为 1 的位置所示，然后点击 **New Terminal**，如编号为 2 的位置所示。



2. 现在，在终端中输入以下命令以克隆此 React 应用程序的模板，并按 Enter 键。

```
git clone https://github.com/ibm-developer-skills-network/todo_list.git
```

3. 上述命令将在“Project”文件夹下创建一个名为“todo_list”的文件夹。您可以在截图中看到该结构。



4. 接下来，您需要通过在终端中输入文件夹名称来运行React应用程序，使用给定的命令。这将设置您的终端路径，以便在**todo_list**文件夹内运行React应用程序。

    ```
    cd todo_list
    ```

5. 为了确保您克隆的代码正常工作，您需要执行以下步骤：

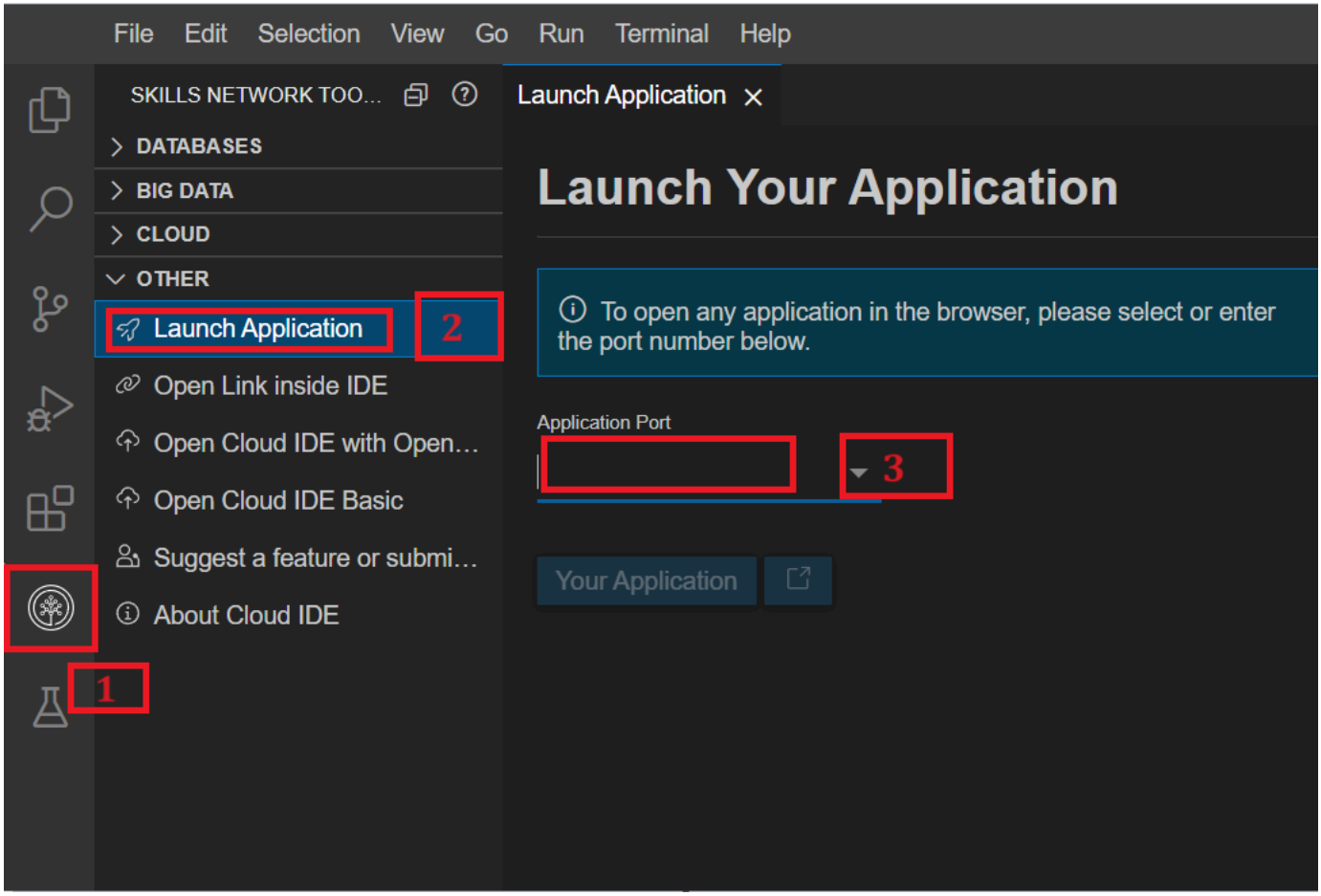
    - 在终端中输入给定的命令并按Enter。这条命令将安装执行应用程序所需的所有必要包。

        ```
        npm install
        ```

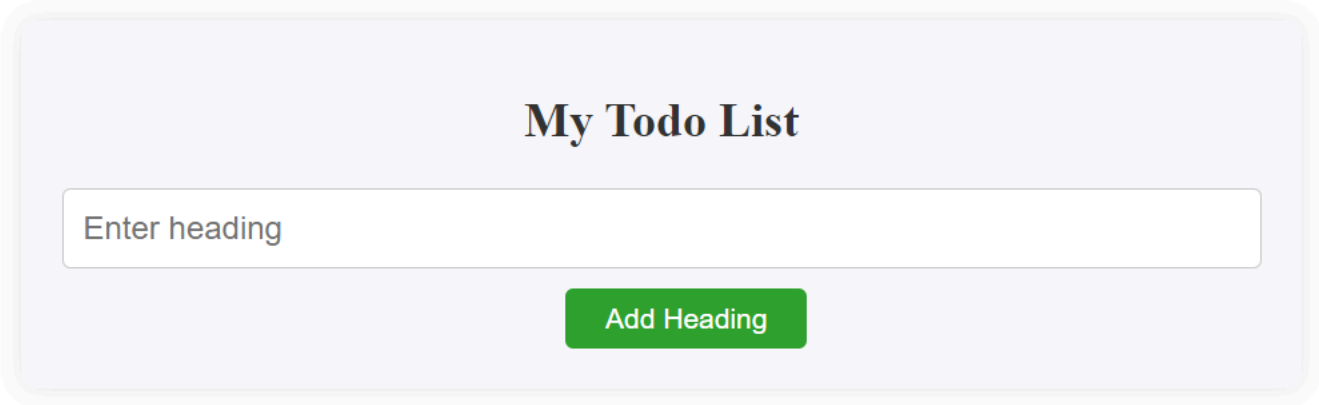
    - 然后执行以下命令以运行应用程序，这将为您提供端口号4173。

        ```
        npm run preview
        ```

6. 要查看您的React应用程序，请点击左侧的Skills Network图标（参见数字1）。此操作将打开**SKILLS NETWORK TOOLBOX**。接下来，点击**Launch Application**（参见数字2）。在**Application Port**中输入端口号**4173**（参见数字3），然后点击 。



7. 输出将显示在给定的截图中。



8. 您可以通过添加、提交并推送到您的GitHub存储库来保存您在此实验室的最新工作。这确保即使您没有持续进行任务，您的进度也会被保存，使您能够从中断的地方继续。

注意：步骤8是可选的。

设置初始状态

1. 接下来，导航到您克隆的 **todo_list** 文件夹中的 **src** 目录下的 **Components** 文件夹中的 **TodoList.jsx** 文件。
2. 该组件的基本结构如下所示。

```jsx
import React, { useState } from 'react';
import './TodoList.css';

const TodoList = () => {
  return (
    <>
      <div className="todo-container">
        <h1 className="title">My Todo List</h1>
        <div className="input-container">
          <input
            type="text"
            className="heading-input"
            placeholder="Enter heading"

          />
          <button className="add-list-button">Add Heading</button>
        </div>
      </div>
      <div className="todo_main">

      </div>
    </>
  );
};

export default TodoList;
```

- 上面的代码表示了 React 应用程序中待办事项组件的基本结构。它包含一个待办事项列表的容器 `<div className="todo-container">`，标题为“My Todo List”，以及一个输入容器 `<div className="input-container">`，其中包含一个用于输入待办事项标题的输入字段和一个标记为“Add Heading”的按钮。
- 还有一个主区域 `<div className="todo_main">`，用于显示待办事项。您需要实现处理用户输入、管理待办事项和动态渲染它们的功能，这将通过使用 React 状态和事件处理来实现。
- 您需要初始化以下三个状态：
  - `todos`：表示一个待办事项数组。将其初始化为空数组 `[]`，表示最初没有待办事项。
  - `headingInput`：表示用户在输入字段中输入的新待办事项标题的值。将其初始化为空字符串 `''`。
  - `listInputs`：将 listInputs 初始化为空对象 `{}`。该状态将单独保存每个待办事项输入字段的值。

```jsx
const [todos, setTodos] = useState([]);
const [headingInput, setHeadingInput] = useState('');
const [listInputs, setListInputs] = useState({});
```

注意：在组件的 `return` 之前包含上述代码。

实现添加标题功能：

Todo 列表的添加标题功能

- 首先，对于类似的待办任务，您将添加特定的标题，例如 `Grocery Items`。在这个标题下，待办事项将列出如牛奶、黄油和面包等项目。
- 对于这个，已经为您提供了一个 `input` 标签，类名为 `heading-input`。要从这个输入框中获取标题，您需要创建一个名为 **handleAddTodo** 的函数，该函数将在点击 **Add Heading** 按钮时触发。
- 检查 `headingInput` 是否为空。如果不为空，创建一个新的待办对象，标题来自 `headingInput`，并且列表为一个空数组。
- 然后将这个待办对象附加到待办数组中，并通过将其设置为空字符串来清除 `headingInput`。

```jsx
const handleAddTodo = () => {
  if (headingInput.trim() !== '') {
    setTodos([...todos, { heading: headingInput, lists: [] }]);
    setHeadingInput('');
  }
};
```

在上面的代码中：

- `const handleAddTodo = () => { ... }`：声明一个名为 `handleAddTodo` 的常量，并将其赋值为一个箭头函数。
- `if (headingInput.trim() !== '') { ... }`：检查 `headingInput` 变量（用户输入的文本）在去除前后空白后是否为空。这个条件确保用户在继续之前输入了一些内容。
- `setTodos([...todos, { heading: headingInput, lists: [] }]);`：如果 if 语句中的条件满足，这行代码更新状态变量 `todos`。它使用扩展语法（...todos）将现有的待办数组扩展到一个新数组中，并附加一个新对象。新对象包含一个 `heading` 属性，值为 `headingInput`，并且 `lists` 属性初始化为空数组。
- `setHeadingInput('');`：在添加新待办项后，这行代码清除 `headingInput` 状态变量，重置文本输入框，以便用户输入新的待办项标题。

注意：在初始化三个状态后包含上述代码。

- 要实现 "添加待办" 功能，需要在类名为 `input-container` 的 div 标签内进行更改，这里有输入字段和添加标题的按钮。
- 需要更新 div 标签中的输入字段和按钮元素，以实现点击按钮添加新待办项的功能。

```
  <div className="input-container">
  <input
    type="text"
    className="heading-input"
    placeholder="输入标题"
    value={headingInput}
    onChange={(e) => {setHeadingInput(e.target.value);}} // 添加 onChange 事件处理程序以更新 headingInput 状态
  />
  <button className="add-list-button" onClick={handleAddTodo}>添加标题</button>
</div>
```

在输入元素中：

- `value` 属性设置为 `{headingInput}`，将输入字段的值绑定到 `headingInput` 状态变量。
- onChange：这是一个在 React 中的事件处理程序，当输入元素的值发生变化时触发。
  - (e) => {...}：这是一个箭头函数，是在 JavaScript 中定义函数的简洁方式。它将事件 (e) 作为参数，该参数代表触发处理程序的事件。
  - {setHeadingInput(e.target.value);}：这是箭头函数的主体。在这里，setHeadingInput 可能是一个更新组件中 headingInput 状态变量的函数。e.target.value 检索触发事件的输入元素的当前值，并调用 setHeadingInput 来使用此值更新状态。

在按钮元素中：

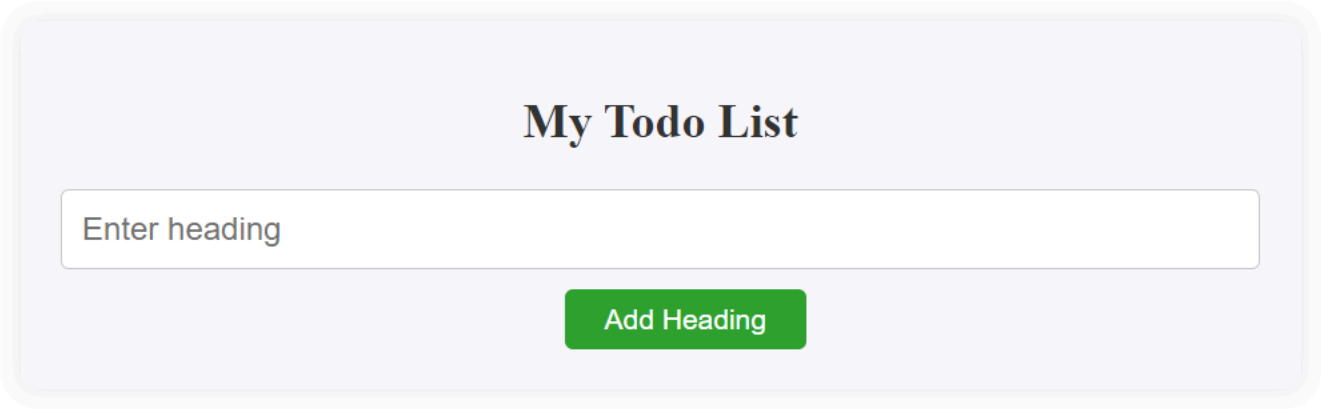
- 添加 `onClick` 事件处理程序，以在按钮点击时触发 `handleAddTodo` 函数。此函数在输入的标题下添加一个新的待办项。

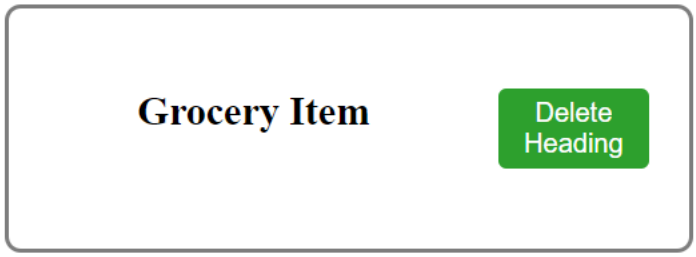
显示待办标题

- 要显示每个待办项的标题，您需要遍历待办数组并在 JSX 中渲染标题。
- 您需要更新 JSX 部分，在类名为 `todo-card` 的 div 内部渲染每个待办项以显示标题。
- 您需要将此代码包含在类名为 `todo_main` 的 `div` 标签内。

```
{todos.map((todo, index) => (
    <div key={index} className="todo-card">
      <div className="heading_todo">
        <h3>{todo.heading}</h3> {/* Display the heading here */}
        <button className="delete-button-heading" onClick={() => handleDeleteTodo(index)}>Delete Heading </button>
      </div>
    </div>
  ))}
```

- 在代码中：
  - `遍历 Todos 数组: todos.map((todo, index) => ...):` 遍历 todos 数组，该数组包含待办事项。map() 函数对数组中的每个待办事项执行指定的函数。
  - `渲染待办事项: <div key={index} className="todo-card"> ... </div>:` 每个待办事项都会渲染一个 class 为 `todo-card` 的 div 元素。key 属性设置为 index，以唯一标识列表中的每个待办事项。
  - `显示待办事项标题: <h3>{todo.heading}</h3>:` 在每个 todo-card div 内，当前待办事项的标题通过 `<h3>` 元素显示。标题文本从待办对象的 heading 属性中获取。
  - `删除待办事项: <button className="delete-button-heading" onClick={() => handleDeleteTodo(index)}>删除标题 </button>:` 每个待办事项旁边都有一个“删除标题”按钮。点击该按钮会触发 `handleDeleteTodo` 函数，并将当前待办事项的索引作为参数传递。这个索引使得函数能够识别并从 todos 数组中删除相应的待办事项。
- 现在保存 `TodoList.jsx` 组件并重新运行应用程序以检查代码的工作情况。如果您的应用程序已经打开，只需刷新网页即可。
- 现在在输入框中输入标题，例如 `Grocery Item`，然后点击 **添加标题** 按钮。您将看到类似于给定截图的输出。





实现添加列表功能

在这个任务中，您将把待办事项列表项包含在特定的标题下。为此，请按照给定的步骤操作。

在 JSX 中添加表单

- 要添加列表并显示列表，您需要包含一个输入框，供用户输入列表名称，以及一个按钮来添加列表。
- 列表应仅在包含添加部分后显示。因此，您需要包含一个输入框和一个按钮。
- 在类名为 `todo-card` 的 `div` 的子元素中，在类名为 `heading_todo` 的 `div` 标签之后包含以下代码，以便它仅在添加标题后出现。

```
<div className='add_list'>
          <input
            type="text"
            className="list-input"
            placeholder="添加列表"
            value={listInputs[index] || ''}
            onChange={(e) => handleListInputChange(index, e.target.value)}/>
          <button className="add-list-button" onClick={() => handleAddList(index)}>添加列表</button>
        </div>
```

- 上面的 JSX 代码片段表示：
  - 一个用于在待办事项中添加新列表项的表单元素。它包含一个输入字段 `<input>`，用于输入新列表项的文本，其值绑定到 `listInput` 状态变量。
  - 输入元素具有一个 `onChange` 事件处理程序。当输入的值发生变化时，事件 (e) 被捕获，并调用 `handleListInputChange` 函数，传入当前索引和从 e.target.value 获取的新值。这更新了 listInputs 状态对象，确保每个待办事项的列表输入被单独跟踪。
  - 此外，还有一个标记为“添加列表”的 `<button>`，在点击时触发 `handleAddList` 函数，并将当前索引值作为参数，以将输入的列表项添加到 todos 数组中指定索引的待办事项。

handleAddList() 函数以添加待办事项

要实现添加标题内列表的功能，您需要执行以下步骤：

- 创建一个名为 `handleAddList` 的函数，该函数以待办事项的索引作为参数。
- 检查 `listInputs[index]` 在去除前后空格后是否不为空。如果不为空，则创建 todos 数组的副本。
- 将来自 `listInputs[index]` 的新列表项推送到指定索引的待办事项的列表数组中。
- 使用修改后的 todos 数组更新状态，并通过将 `listInputs[index]` 设置为空字符串来清除它。

```
const handleAddList = (index) => {
    if (listInputs[index] && listInputs[index].trim() !== '') {
        const newTodos = [...todos];
```

```
            newTodos[index].lists.push(listInputs[index]);
            setTodos(newTodos);
            setListInputs({ ...listInputs, [index]: '' });
        }
    };
    const handleListInputChange = (index, value) => {
        setListInputs({ ...listInputs, [index]: value });
    };
```

- `const handleAddList = (index) => { ... }`: 声明一个名为 `handleAddList` 的常量，并将其分配给一个箭头函数，该函数接受一个索引参数。
- if (listInputs[index] && listInputs[index].trim() !== '') { ... }: 检查 listInputs[index] 变量，即用户输入的新列表项的文本，在去除前后空格后是否不为空。这个条件确保用户在继续之前输入了一些内容。
- `const newTodos = [...todos];`: 使用扩展语法 (`...todos`) 创建 `todos` 数组的浅拷贝。进行拷贝是为了避免直接修改状态，这是 React 中的最佳实践。
- `newTodos[index].lists.push(listInput);`: 访问 `newTodos` 数组中指定索引的待办事项，并将 `listInputs[index]` 的值推送到其 `lists` 数组中。此推送假设每个待办事项都有一个 `lists` 属性，该属性是一个包含该待办事项内项目的数组。
- `setTodos(newTodos);`: 在用新列表项更新 `newTodos` 数组后，使用 React 的 useState 钩子提供的状态更新函数 setTodos 更新状态变量 `todos`。
- `setListInput('');`: 最后，此函数重置 `listInput` 状态变量，清除用于添加新列表项的文本输入字段。

`handleListInputChange` 函数：

- `const handleListInputChange = (index, value) => { ... }`: 声明一个名为 handleListInputChange 的常量，并将其分配给一个箭头函数，该函数接受两个参数：待办事项的索引和列表输入的新值。
- `setListInputs({ ...listInputs, [index]: value });`: 此函数使用指定索引的新值更新 listInputs 状态对象，确保每个待办事项的列表输入被单独跟踪。

在 JSX 中显示待办事项列表

- 要显示列表，您需要在 `<ul>` 标签内迭代 `todo.lists` 并在 `<li>` 标签中包含用户输入。

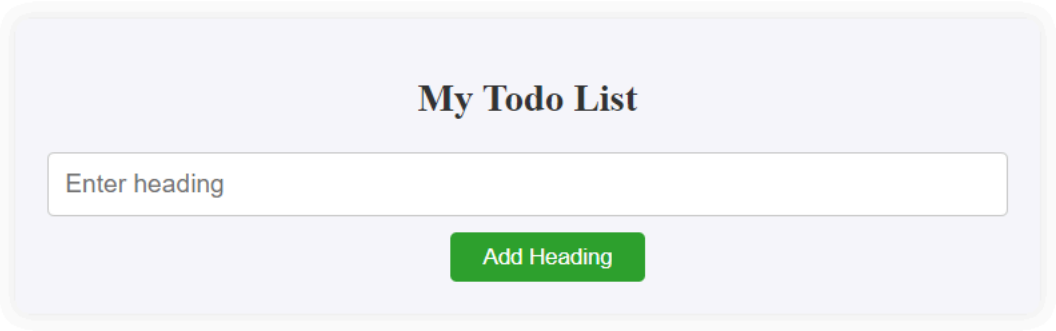
```
            <ul>
             {todo.lists.map((list, listIndex) => (
               <li key={listIndex} className='todo_inside_list'>
                <p>{list}</p>
               </li>
             ))}
            </ul>
```

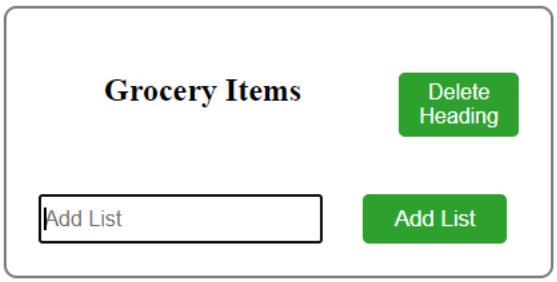
```

- This JSX code snippet renders a list of items within a todo item. It utilizes the map function to iterate over the lists array of the todo object (representing a todo item).
- For each item in the lists array, it generates a `&lt;li&gt;` element with a unique key attribute set to `listIndex` to ensure proper rendering and performance optimization.
- Inside each `&lt;li&gt;` element, it displays the list item' content wrapped in a `&lt;p&gt;` element.

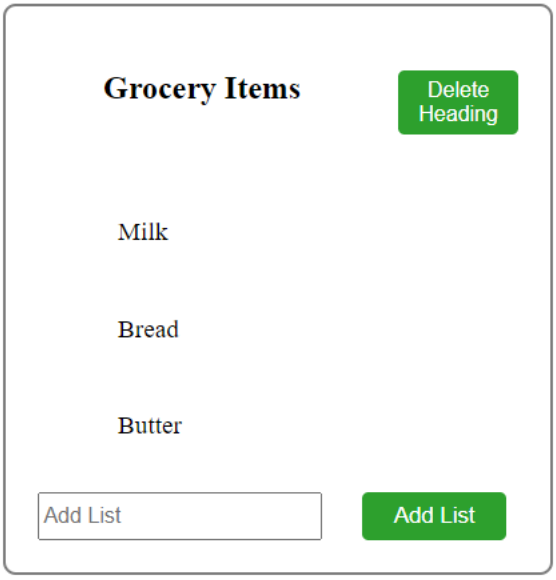
*Note:Include above code before `&lt;div&gt;` with class name **add_list***

- Check the output by re running the application again. At first you will see output according to given screenshot.
```





- Then write your grocery list one after another and click on **Add List** button. Then you items will display as below:



Delete Heading With Todo List

- To delete the heading section, the entire todo list will also be deleted. Suppose if a user has entered the heading as **Grocery List** and all items under this heading have been added, then the user should be able to delete tthe entire list from their application interface.
- To delete the list create a function with name `handleDeleteTodo` and apply logic to delete the element.

```
const handleDeleteTodo = (index) => {
   const newTodos = [...todos];
   newTodos.splice(index, 1);
   setTodos(newTodos);
 };
```

- The function `handleDeleteTodo` is designed to remove a todo item from the todos array at a specific index in the above code as follows:
  - `const handleDeleteTodo = (index) => { ... }`: Declares a constant named `handleDeleteTodo` that has an arrow function which takes an `index` parameter, indicating the index of the todo item to be deleted.
  - `const newTodos = [...todos];`: Creates a shallow copy of the todos array using the spread syntax (...todos). This step is crucial to avoid directly mutating the original state.
  - `newTodos.splice(index, 1);`: The splice method is called on the newTodos array to remove one element at the specified index.

- `setTodos(newTodos);`: Finally, the setTodos function, provided by React's useState hook, is called with the updated `newTodos` array as an argument. This updates the state variable `todos` with the modified array, removing the todo item specified by the index from the UI and re-rendering the component accordingly.

Note: Include above code before the `return`of component.

- Now you need to include `onClick` event in the button with class name **delete-button-heading**

```
<button className="delete-button-heading" onClick={handleDeleteTodo}>Delete Heading</button>
```

▼ Click here for the solution code of TodoList.jsx

```
import React, { useState } from 'react';
import './TodoList.css';
const TodoList = () => {
    const [todos, setTodos] = useState([]);
    const [headingInput, setHeadingInput] = useState('');
    const [listInputs, setListInputs] = useState({});
    const handleAddTodo = () => {
        if (headingInput.trim() !== '') {
            setTodos([...todos, { heading: headingInput, lists: [] }]);
            setHeadingInput('');
        }
    };
     const handleDeleteTodo = (index) => {
        const newTodos = [...todos];
        newTodos.splice(index, 1);
        setTodos(newTodos);
      };
    const handleAddList = (index) => {
        if (listInputs[index] && listInputs[index].trim() !== '') {
            const newTodos = [...todos];
            newTodos[index].lists.push(listInputs[index]);
            setTodos(newTodos);
            setListInputs({ ...listInputs, [index]: '' });
        }
    };
    const handleListInputChange = (index, value) => {
        setListInputs({ ...listInputs, [index]: value });
    };
    return (
        <>
            <div className="todo-container">
                <h1 className="title">My Todo List</h1>
                <div className="input-container">
                    <input
                        type="text"
                        className="heading-input"
                        placeholder="Enter heading"
                        value={headingInput}
                        onChange={(e) => setHeadingInput(e.target.value)}
                    />
                    <button className="add-list-button" onClick={handleAddTodo}>Add Heading</button>
                </div>
            </div>
            <div className="todo_main">
                {todos.map((todo, index) => (
                    <div key={index} className="todo-card">
                        <div className="heading_todo">
                            <h3>{todo.heading}</h3>
                            <button className="delete-button-heading" onClick={() => handleDeleteTodo(index)}>Delete Heading</buttor
                        </div>
                        <ul>
                            {todo.lists.map((list, listIndex) => (
                                <li key={listIndex} className='todo_inside_list'>
                                    <p>{list}</p>
                                </li>
                            ))}
                        </ul>
                        <div className='add_list'>
                            <input
                                type="text"
                                className="list-input"
                                placeholder="Add List"
                                value={listInputs[index] || ''}
                                onChange={(e) => handleListInputChange(index, e.target.value)}
                            />
                            <button className="add-list-button" onClick={() => handleAddList(index)}>Add List</button>
                        </div>
                    </div>
                ))}
            </div>
        </>
    );
};
export default TodoList;
```

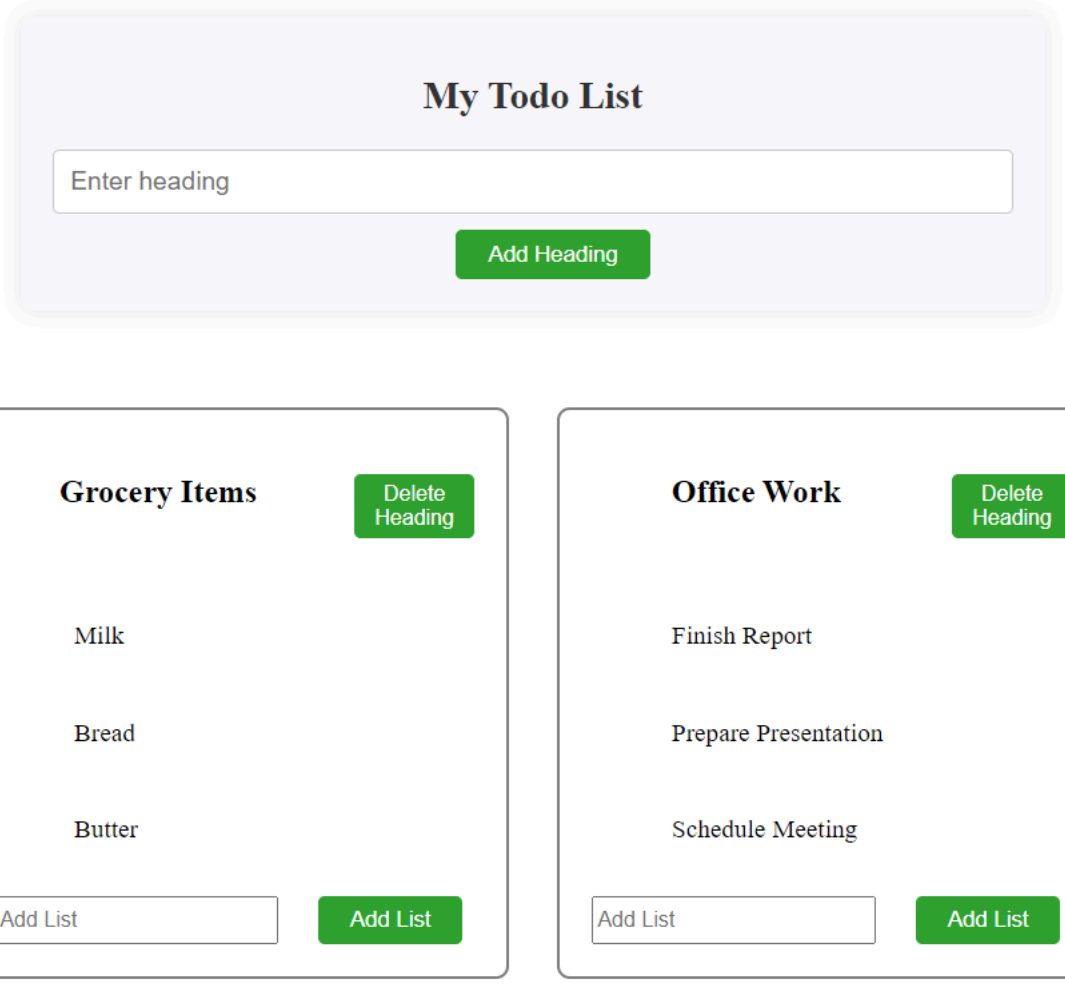
Check the output

1. Stop the execution of the React application in the terminal by performing `ctrl+c` to quit.
2. Then, write the given command in the terminal and hit Enter.

```
npm run preview
```

3. To view your React application, refresh the already opened webpage for the React application on your browser. If it is not open, then click the Skills Network icon on the left panel. This action will open the "SKILLS NETWORK TOOLBOX." Next, select "Launch Application". Enter the port number **4173** in "Application Port" and click .

4. The output will display as per the given screenshot after adding multiple headings and multiple todo lists inside heading.



Note– To see the latest changes, you need to execute `npm run preview` again in the terminal.

Congratulations! You have created a ToDo list React application!

Conclusion

- In this lab, you have learned how to manage state in a React functional component using the useState hook. State variables such as todos, headingInput, and listInput are used to maintain the state of the Todo List and user input fields.
- You have learned to render headings and associated lists based on the state maintained by React's useState hook.
- You have used code that implements event handling functions to add new headings and lists, and to handle changes in input fields. These functions update the state of the Todo List, triggering re-renders to reflect the changes in the UI.
- You have created a Todo List component that is well-structured, with separate sections for input fields, heading display, list display, and add list functionality. This modular structure enhances code readability and maintainability.

Author(s)

Richa Arora

© IBM Corporation. All rights reserved.