

Practice Project: Conference Event Planner



Estimated time needed: 90 minutes

Understand the task

Alejandro manages a venue for business conferences. Her parent company, "BudgetEase" wants to hire you to develop a website so BudgetEase customers can price their conference events easily.

The application's requirements include allowing users to select and price the rooms in the conference center, add-on selections, like microphones and projectors, and meals for a given number of guests.

The *BudgetEase* conference expense planner features will include:

- A dynamic user interface that updates in real time based on user selections
- Components for venue selection, add-ons, and meal options
- Redux integration using Redux Toolkit to manage state changes
- Redux slices to manage different section states
- Display selected items and their costs with a table in a pop-up window
- Calculate and display subtotals and total costs based on user selections

Learning objectives

After completing this lab, you will be able to:

- **Create React components:** Create functional React components using component composition and nesting.
- **Manage states with hooks:** Implement React Hooks, specifically the `useState` and `useEffect` hooks. You will use hooks to manage component-level state and control the visibility of elements.
- **Integrate Redux:** Integrate Redux within an application using Redux concepts like actions, reducers, and the store.
- **Render dynamic data:** Dynamically render data fetched from an array of objects into the UI. You will map over arrays to generate lists of components.
- **Handle events using conditional rendering:** Handle user events such as button selection and trigger corresponding actions.

Project tasks

1. Setup the project environment
2. Review the structure of the `ConferenceEvent.jsx` component
3. Review the code for the venue module
4. Combine Redux with components to manage updates and state changes
5. Add logic to calculate subtotals and total cost
6. Make a dynamic table to show chosen products, displaying the item name, unit cost, quantity, and overall costs for that item
7. Create a web design for a comfortable user experience
8. Deploy your website to a public hosting service

Solutions

You will find the solution code at the end of this lab. If you need help completing any of the tasks, you can find a suggested version of the working code there. Also, be sure to save either your solution or the code at the end of this lab. It will help you in developing the code for the final project.

Prerequisites

- Basic HTML and CSS
- Intermediate JavaScript
- Familiarity with React function components, hooks, and Redux toolkit for state management
- Code management using GitHub

Review [these instructions](#) if you need guidance on how to work in GitHub.

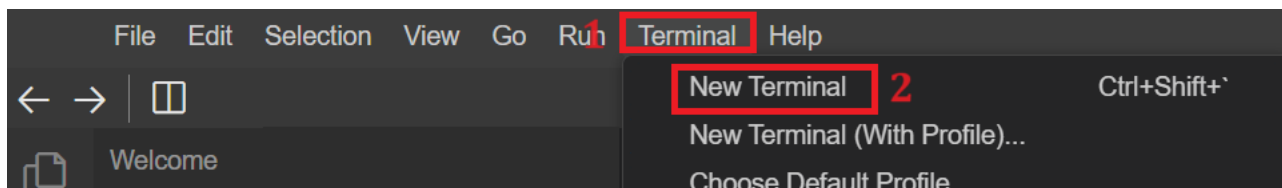
Important notice about this lab environment

Skills Network Cloud IDE (based on Theia and Docker) is an open-source IDE (Integrated Development Environment) that provides an environment for hands-on labs in course and project-related labs.

Please be aware that sessions for this lab environment are not persistent. Every time you connect to this lab, a new environment is created for you. You will lose data if you exit the environment without saving to GitHub or another external source. Plan to complete these labs in a single session to avoid losing your data.

Task 1: Setting up the environment

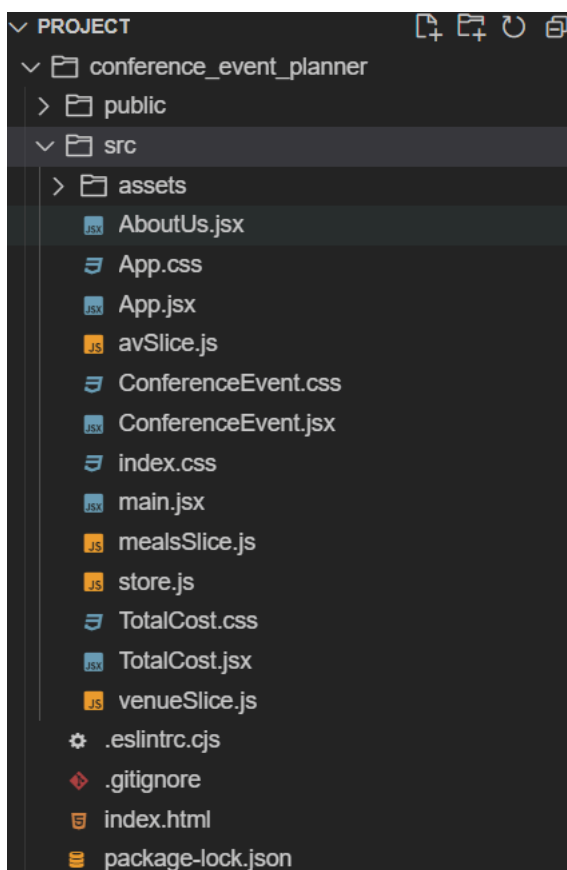
1. Create a public GitHub repository.
2. From the **Terminal** menu at the top of the IDE, select **New Terminal**.



3. In the terminal, write the following command to clone the boiler template for this React application. Press **Enter**.

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

The above command will create a folder, "conference_event_planner", under the "Project" folder. The screenshot displays the directory structure.



4. You can either find your own images for the project or use these suggested images from Pixabay, a site that supplies royalty-free images.

Conference room

<https://pixabay.com/photos/chairs-empty-office-room-table-2181916/>

Auditorium:

<https://pixabay.com/photos/event-venue-auditorium-meeting-1597531/>

Presentation room:

<https://pixabay.com/photos/convention-center-chair-seminar-3908238/>

Meeting room:

<https://pixabay.com/photos/chairs-empty-office-room-table-2181916/>

Small meeting room:

<https://pixabay.com/photos/laptops-meeting-businessmen-593296/>

Projector:

<https://pixabay.com/photos/business-computer-conference-20031/>

Speakers:

<https://pixabay.com/photos/speakers-bluetooth-tech-speaker-4109274/>

Microphone:

<https://pixabay.com/photos/public-speaking-mic-microphone-3926344/>

Whiteboard:

<https://pixabay.com/photos/whiteboard-dry-erase-marker-blank-2903269/>

Signs:

<https://pixabay.com/photos/signpost-waypoint-wood-grain-board-235079/>

5. Change your terminal path to the **conference_event_planner** folder using the following command.

```
cd conference_event_planner
```

6. Ensure the code you cloned works correctly.


Install the necessary packages to execute the application using npm.

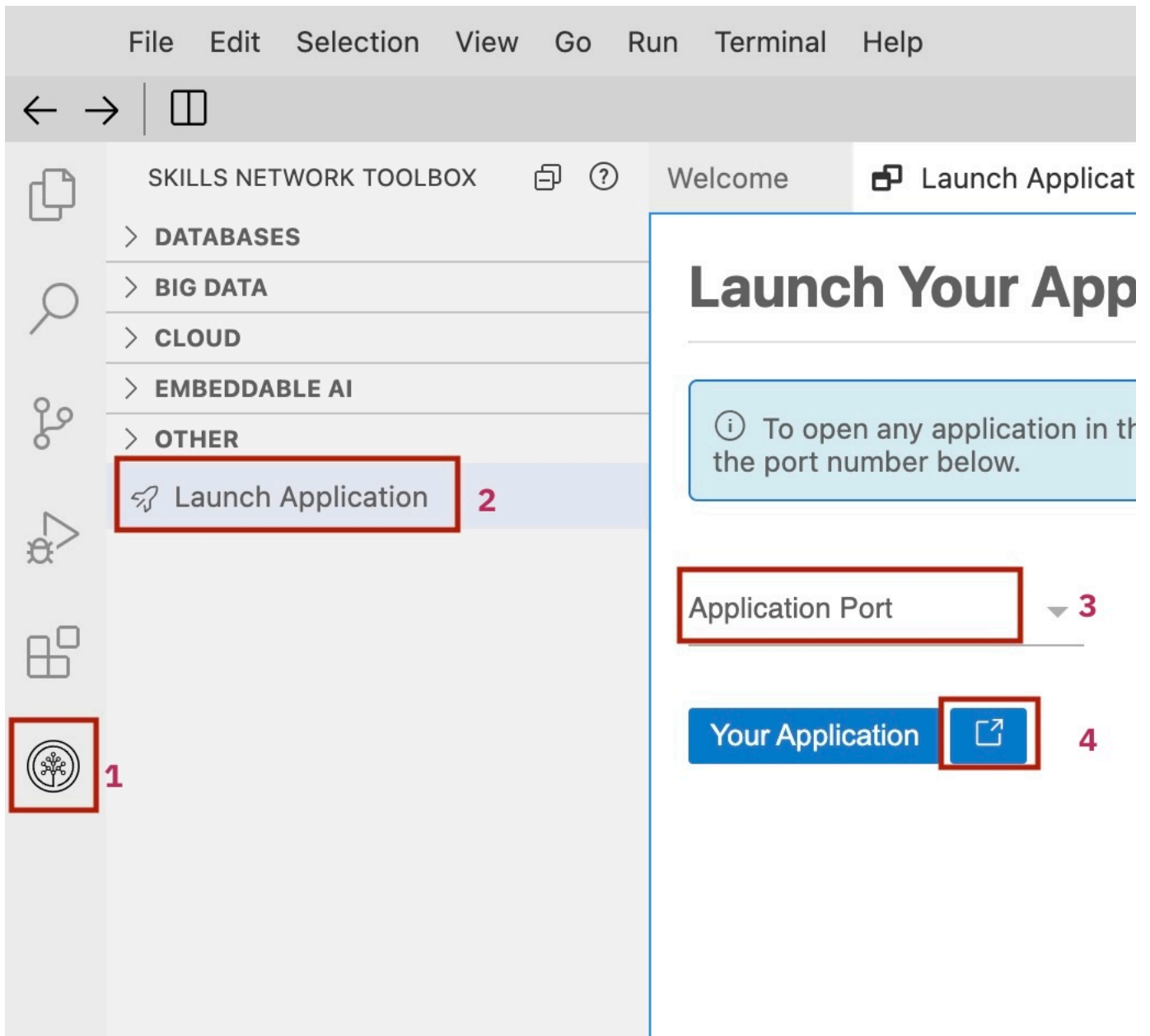
```
npm install
```

Execute this command to run the application, which will start the application server running on port number 4173.

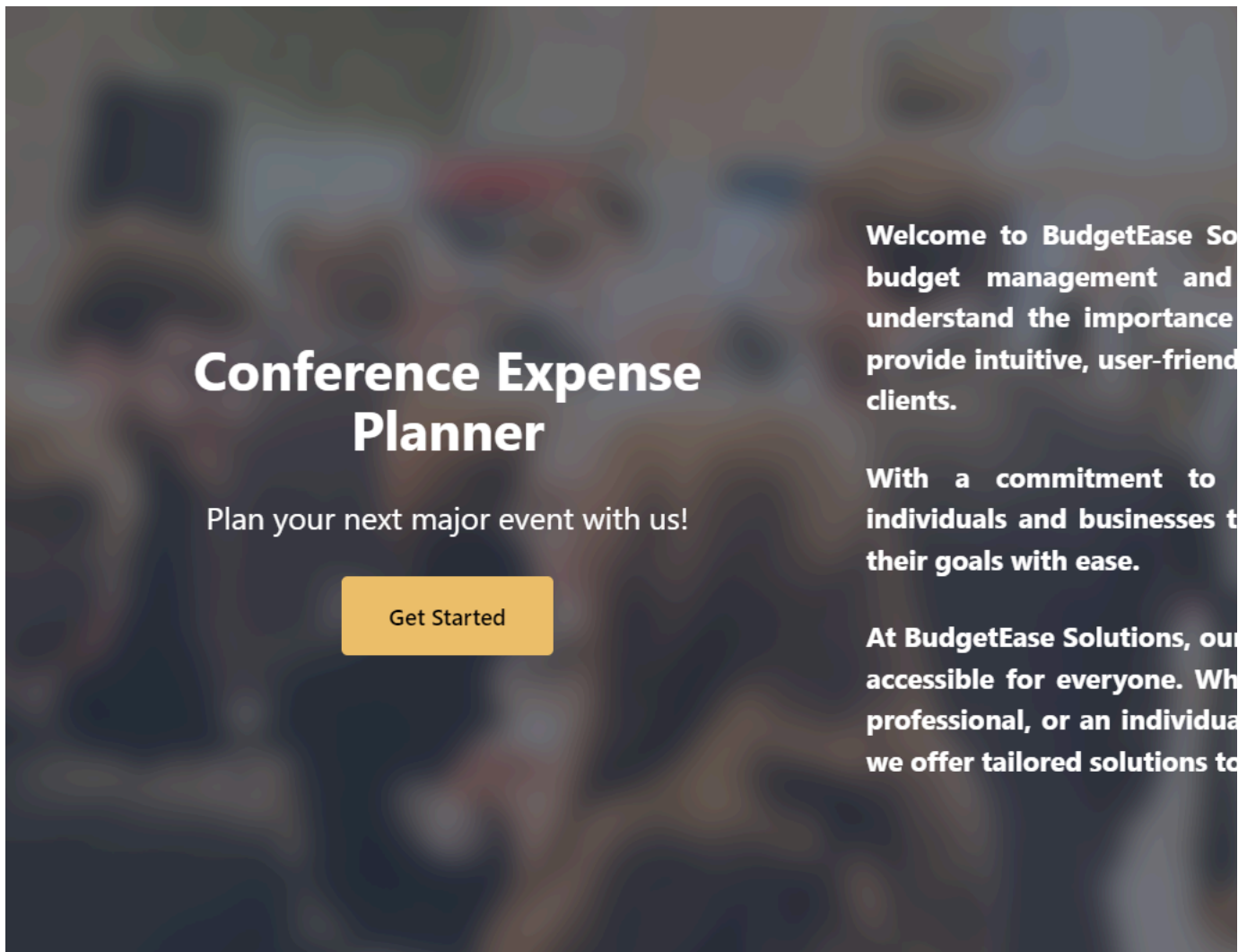
```
npm run preview
```

7. Now, you can view the application.

Click the **Skills Network** icon on the left panel (refer to number 1). This action will open the **Skills Network Toolbox**. Next, click **Launch Application** (refer to number 2). Enter port number **4173** in **Application Port** (refer to number 3) and click .

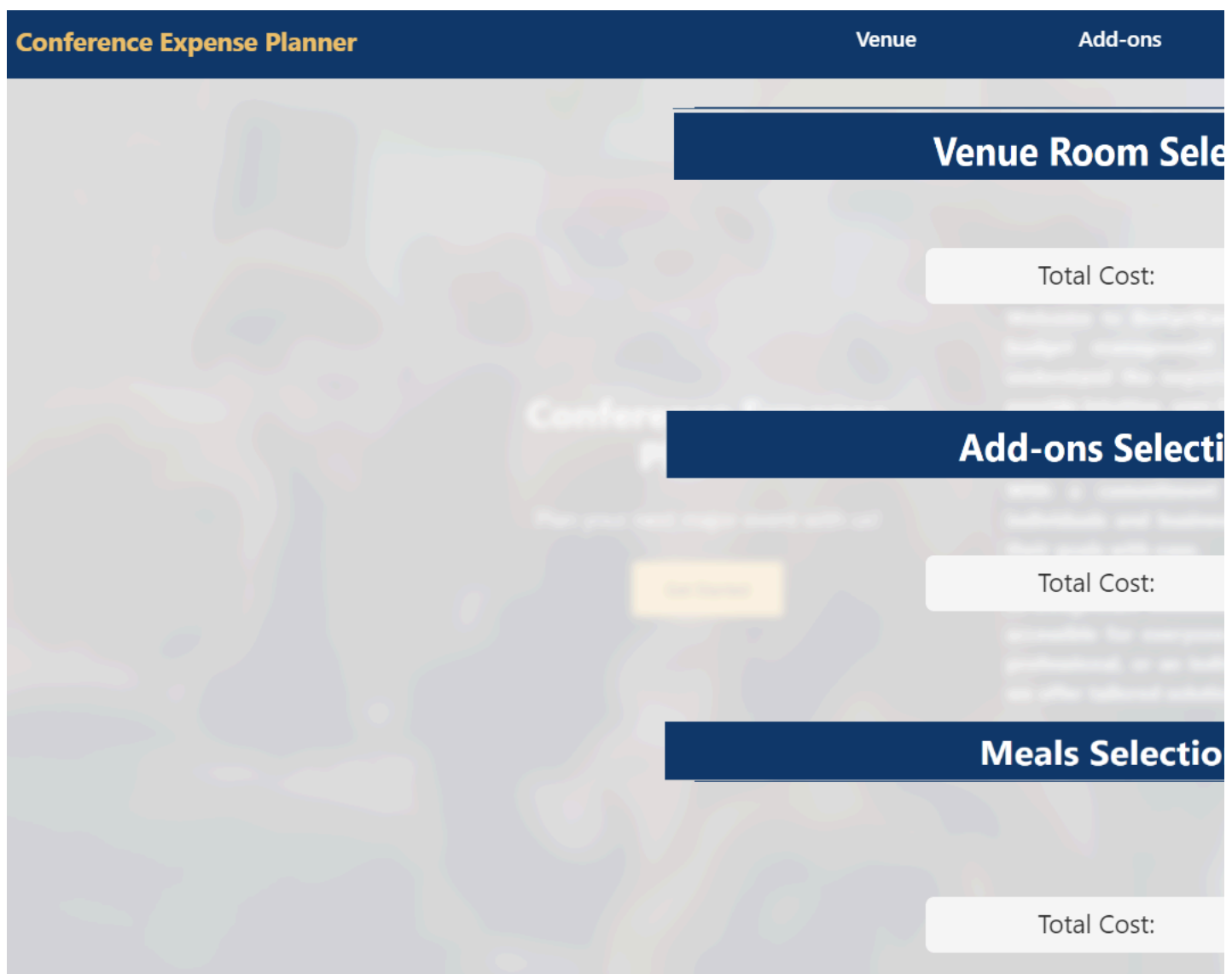


8. The output should look similar to the following screenshot. Clicking the **Get Started** button should take the user to the product selection page. The header and first section, "Room selection".



*Note: You have **been provided** with `backgroundColor` instead of `backgroundImage`. If you prefer an image rather than a color, you can incorporate your own image.

The application product page with the given code should look similar to the following:



10. Now save all the code. If you made any changes, perform `git add` and `git commit`. Then, perform the `git push` command to upload the committed changes to your remote GitHub repository, ensuring that you saved the latest version of your code.

Note: While pushing the code to your GitHub repository, it might ask you to provide a username and password. Review [these instructions](#) if you need more guidance on how to work in GitHub.

Task 2: Review ConferenceEvent.jsx structure

You have two working modules:

- The landing page with a button to get started and a company description.
- The "venue section" includes room selection in the venue and increment and decrement buttons.

Open the `ConferenceEvent.jsx` file in the `src` directory. This component includes functions and layouts for the product selection page.

Layouts

- A navbar bar element
- The `main_container` class in a `<div>` tag
- A `showItems` variable to toggle functionality using the conditional operator `?`
- The `items-information` class in a `<div>` tag
- Layouts for `venue`, `addons`, and `meals` in `<div>` tags. Each of these layouts has two common class names, `venue_container` and `container_main`
- A `total_amount_detail` class in a `<div>` tag to display the details of selected items

Task 3: Review venue module code

Next, you will walk through the code for the venue room selection functionality provided to you. Focus on the state flow and how the `ConferenceEvent` component works in conjunction with the `venueSlice`. The venue room selection functionality code is in three files: `venueSlice.js`, `ConferenceEvent.jsx`, and `store.js`. Let's discuss the code in these and the interactions among them.

The venue functionality uses **slices** from the Redux toolkit by importing the `createSlice()` function. A slice breaks down your application state into smaller features, helping organize your code, making it easier to read and simpler to maintain.

venueSlice.js:

Let's step through the code in the `venueSlice.js` file. It contains code to slice the Redux state related to venue selection using `createSlice` from `@reduxjs/toolkit`.

1. The initial state consists of an array of venue objects, each representing a rentable room in the venue. A venue object has properties such as the thumbnail image, name, cost, and quantity.
2. The `venueSlice.js` file includes reducer functions `incrementQuantity` and `decrementQuantity` to manage the number of venue items in the state.
 - Note: On the last page of this lab, we provide you with the links for images from Pixabay, with appropriate citations, or you may find your own.
3. `incrementQuantity()`:
 - This function handles incrementing the quantity of a venue item in the state. It receives an action containing the index of the item to **be incremented**.
 - It first checks if the item exists in the state at the provided index. If the item exists and it's an Auditorium Hall with a quantity greater than or equal to 3, it returns early without modifying the state.
 - Otherwise, it increments the quantity of the item by one.
4. `decrementQuantity()`:
 - This function handles decrementing the quantity of a venue item in the state. It receives an action containing the index of the item to **be decremented**.
 - It first checks if the item exists in the state at the provided index and if its quantity is greater than 0.
 - If both conditions **are met**, the quantity of the item will be decreased by one.

```
reducers: {
  incrementQuantity: (state, action) => {
    const { payload: index } = action;
    if (state[index]) {
      if (state[index].name === "Auditorium Hall (Capacity:200)" && state[index].quantity >= 3) {
        return;
      }
      state[index].quantity++;
    }
  },
  decrementQuantity: (state, action) => {
    const { payload: index } = action;
    if (state[index] && state[index].quantity > 0) {
      state[index].quantity--;
    }
  },
},
```

Redux Store Setup

Now we'll look at the `store.js` file.

1. Create the Redux store with the `configureStore()` function from `@reduxjs/toolkit`
2. The `store.js` file contains a reducer called `venue()`, imported from `venueSlice.js`

```
import { configureStore } from '@reduxjs/toolkit';
import venueReducer from './venueSlice';
export default configureStore({
  reducer: {
    venue: venueReducer,
  },
});
```

3. This code creates a global Redux store using the `@reduxjs/toolkit\configureStore()` function so all components in the application can access the state managed by the `venueReducer()`.

ConferenceEvent Component

Now let's walkthrough the relevant code from the `ConferenceEvent.jsx` file.

1. First, import the required dependencies.

```
import { useSelector, useDispatch } from "react-redux";
import { incrementQuantity, decrementQuantity } from "./venueSlice";
```

2. The `useSelector()` function retrieves venue items from the Redux store state.

```
const venueItems = useSelector((state) => state.venue);
```

3. It then defines event handlers like `handleAddToCart()`, and `handleRemoveFromCart()` to manage the increase and decrease quantities from the user interactions.

```
const handleAddToCart = (index) => {
  if (venueItems[index].name === "Auditorium Hall (Capacity:200)" && venueItems[index].quantity >= 3) {
    return; // Prevent further additions
  }
  dispatch(incrementQuantity(index));
};
const handleRemoveFromCart = (index) => {
  if (venueItems[index].quantity > 0) {
    dispatch(decrementQuantity(index));
  }
};
```

4. Next, it calculates the remaining number of available auditorium halls to three, so the user cannot request more than three.

```
const remainingAuditoriumQuantity = 3 - venueItems.find(item => item.name === "Auditorium Hall (Capacity:200)").quantity;
```

5. When the user selects the buttons to increase or decrease the number of rooms, the system should calculate the cost for all selected rooms.

6. For this, we define a `calculateTotalCost()` function and declare a `venueTotalCost`.

```
const calculateTotalCost = (section) => {
  let totalCost = 0;
  if (section === "venue") {
    venueItems.forEach((item) => {
      totalCost += item.cost * item.quantity;
    });
  }
  return totalCost;
};
const venueTotalCost = calculateTotalCost("venue");
```

7. Function declaration:

- The function is defined using arrow function syntax and assigned to the constant `calculateTotalCost`.
- It takes one string parameter, `section`, that indicates the section **is calculated**.

8. Initialization of `totalCost`:

- The `totalCost` variable is initialized to 0. This value will hold the cumulative total cost for the specified section.

9. Conditional check:

- The function checks if the section passed as an argument equals the string "venue".
- If true, **the total cost for the venue items will be calculated**.

10. Iteration over `venueItems`:

- The `venueItems` items array represents an item with properties `cost` and `quantity`.
- The `forEach` function iterates over each item in the `venueItems` array. For each item, it multiplies `item.cost` by `item.quantity` and adds the result to `totalCost`.

11. Return statement:

- After the loop is complete, the function returns the calculated `totalCost`.

12. Function call:

- The function `calculateTotalCost` is called with the "venue" argument, which calculates the total cost for the items in the "venue" section.
- The result of this calculation is stored in the constant `venueTotalCost`.

13. Lastly, the component displays the total cost of all selected venue items.

```
<div className="total_cost">Total Cost: ${venueTotalCost}</div>
```

Task 4: Write code for add-ons

Now, it's your turn to write some code. In this section, you will create the add-ons section. To begin, you need to first create logic within `avSlice.js` under the **src** folder.

1. Initialize the `initialState` array variable with objects to provide a data structure. Note that you need to add your own images and the appropriate paths to them in the data below. Links to sample images are provided in **Task 1: Setting up the environment**. Include the code below within `initialState` of `avSlice.js`.

```
{
  img: "https://pixabay.com/images/download/business-20031_640.jpg",
  name: "Projectors",
  cost: 200,
  quantity: 0,
},
{
  img: "https://pixabay.com/images/download/speakers-4109274_640.jpg",
  name: "Speaker",
  cost: 35,
```



```

    quantity: 0,
  },
  {
    img: "https://pixabay.com/images/download/public-speaking-3926344_640.jpg",
    name: "Microphones",
    cost: 45,
    quantity: 0,
  },
  {
    img: "https://pixabay.com/images/download/whiteboard-2903269_640.png",
    name: "Whiteboards",
    cost: 80,
    quantity: 0,
  },
  {
    img: "https://pixabay.com/images/download/signpost-235079_640.jpg",
    name: "Signage",
    cost: 80,
    quantity: 0,
  },
],

```

Increment and decrement

Now, you need to create the logic for `incrementAvQuantity()` and `decrementAvQuantity()` functions.

1. The `incrementAvQuantity()` reducer function increments the quantity of a specific item in the state.
2. It takes two parameters: `state` and `action`.
3. The `action.payload` object contains the identifier of the item to increment.
4. The reducer retrieves the item from the state using `state[action.payload]`.
5. If the item exists, it increments its quantity property by 1.

```

incrementAvQuantity: (state, action) => {
  const item = state[action.payload];
  if (item) {
    item.quantity++;
  }
},

```

6. The `decrementAvQuantity()` reducer function decrements the quantity of a specific item in the state.
7. Similar to `incrementAvQuantity()`, it takes two parameters: `state` and `action`.
8. The `action.payload` object contains the item identifier to decrement. - It's reducer retrieves the item from the state using `state[action.payload]`.
9. If the item exists and its quantity is greater than 0, it decrements its quantity property by 1, ensuring the quantity doesn't drop below 0 and indicates no more available items.

```

decrementAvQuantity: (state, action) => {
  const item = state[action.payload];
  if (item && item.quantity > 0) {
    item.quantity--;
  }
},

```

10. Export all reducer functions and actions in `avSlice.js`.

11. Provide the `avSlice` to the `store.js` file. Navigate to the `store.js` file and import `avSlice.js`. The file structure should look like the code below.

```

import { configureStore } from '@reduxjs/toolkit';
import venueReducer from './venueSlice';
import avReducer from './avSlice';
export default configureStore({
  reducer: {
    venue: venueReducer,
    av: avReducer,
  },
});

```

12. Now you need to display the details of the add-ons you initialized in `avSlice.js`.

13. Open the `ConferenceEvent.jsx` component. Import the add-on items into the `ConferenceEvent.jsx` component from `store.js` where `avSlice.js` sends the details using reducers.

```
const avItems = useSelector((state) => state.av);
```

Note: Include above code after `const venueItems = useSelector((state) => state.venue);` `ConferenceEvent.jsx` component.

14. Display items from the `avItems` variable using the `map()` method. Also include increase and decrease buttons to add or reduce the quantity by 1. Also, include the code inside a `<div>` tag with the class name `addons_selection` as shown in the following code. You can read the code explanation after the code snippet.

```

{avItems.map((item, index) => (
  <div className="av_data venue_main" key={index}>
    <div className="img">
      <img src={item.img} alt={item.name} />
    </div>
    <div className="text"> {item.name} </div>
    <div> ${item.cost} </div>
    <div className="addons_btn">
      <button className="btn-warning" onClick={() => handleDecrementAvQuantity(index)}> &ndash; </button>
      <span className="quantity-value">{item.quantity}</span>
    </div>
  </div>
)}

```

```

        <button className=" btn-success" onClick={() => handleIncrementAvQuantity(index)}> &#43; </button>
      </div>
    </div>
  )))

```

15. This code uses the `map()` function to iterate over an array called `avItems`, which contains information about audio-visual items. Let's review how the code works.

16. Each item in the array creates a `<div>` element with the class `av_data venue_main`.

17. Inside this `<div>`, it contains:

- An `` tag displays the item's image. The image source (`src`) **is obtained** from `item.img`, and the alt text is set to `item.name`.
- A `<div>` displaying the item's name (`item.name`).
- Another `<div>` displays the item's cost, `item.cost`, in dollars.
- A set of buttons wrapped in a `<div>` with class `addons_btn`. These buttons allow users to adjust the quantity of the item. There are two buttons:
 - The first button, with the class `btn-warning`, is labeled with an n-dash (-). It decrements the quantity when selected. Its click event is bound to the function `handleDecrementAvQuantity()` with the item's index as an argument.
 - The second button, with the class `btn-success`, is labeled with a plus sign (+). It increments the quantity when selected. Its click event is bound to the function `handleIncrementAvQuantity()` with the item's index as an argument.

18. The item's current quantity is displayed between the decrement and increment buttons. This value **is obtained** from the `item.quantity` object.

19. Next, you need to dispatch the actions. For the functions `handleDecrementAvQuantity()` and `handleIncrementAvQuantity()`, include the following code in their respective functions before returning the component.

```

const handleIncrementAvQuantity = (index) => {
  dispatch(incrementAvQuantity(index));
};
const handleDecrementAvQuantity = (index) => {
  dispatch(decrementAvQuantity(index));
};

```

20. To implement the above code, make sure to import `incrementAvQuantity` and `decrementAvQuantity` from `avSlice`. Include the above code at the top of the `ConferenceEvent.jsx` component.

```
import { incrementAvQuantity, decrementAvQuantity } from "../avSlice";
```

21. Now, create logic to calculate the total cost for selected AV items within the `calculateTotalCost` function, just as the cost is calculated for `venueSlice`.

```

const calculateTotalCost = (section) => {
  let totalCost = 0;
  if (section === "venue") {
    venueItems.forEach((item) => {
      totalCost += item.cost * item.quantity;
    });
  } else if (section === "av") {
    avItems.forEach((item) => {
      totalCost += item.cost * item.quantity;
    });
  }
  return totalCost;
};

```

22. Also, create one variable named `avTotalCost` and call this function by passing `av` as the parameter.

```
const avTotalCost = calculateTotalCost("av");
```

23. Include `avTotalCost` within the add-ons section to display the total cost of the selected av items.

```
<div className="total_cost">Total Cost: {avTotalCost}</div>
```

Task 5: Write code for meals selection

In this section, you will write the code for the meal selection functionality. To perform this functionality, you will include checkboxes to select the meals, allowing clients to click on a checkbox to select an item or uncheck it to deselect it.

Meal states

1. Navigate to the `mealsSlice.js` file in the `src` folder. It will include four meal items in an array. Include the code below within the `initialState` variable.

```

{ name: 'Breakfast', cost: 50, selected: false },
{ name: 'High Tea', cost: 25, selected: false },
{ name: 'Lunch', cost: 65, selected: false },
{ name: 'Dinner', cost: 70, selected: false },

```

2. Next, create the logic to select or de-select meal items to manage the state in the reducer.

```
toggleMealSelection: (state, action) => {
  state[action.payload].selected = !state[action.payload].selected;
},
```

3. The `toggleMealSelection` function toggles the selected property of a specific item in the state. It takes the current state and an action object, using `action.payload` to identify the item to update. It then switches the selected status of that item from true to false or vice versa.

► See the code for `mealsSlice.js`

Provide reducers to the store

4. Next, edit the `store.js` file to provide the reducer details for importing `mealsSlice.js`.

```
import { configureStore } from '@reduxjs/toolkit';
import venueReducer from './venueSlice';
import avReducer from './avSlice';
import mealsReducer from './mealsSlice';
export default configureStore({
  reducer: {
    venue: venueReducer,
    av: avReducer,
    meals: mealsReducer,
  },
});
```

Display meal items

5. Next, you will need to display the details of the meal items you initialized in `mealsSlice.js`. To do this, open the `ConferenceEvent.jsx` file.

6. Import the meal items into the `ConferenceEvent.jsx` component from `store.js` where `mealsSlice.js` sends the details using reducers.

```
const mealsItems = useSelector((state) => state.meals);
```

7. Next, include an input box to get the total number of people within the `<div>` tag with class name **input-container**

```
<div className="input-container venue_selection">
  <label htmlFor="numberOfPeople"><h3>Number of People:</h3></label>
  <input type="number" className="input_box5" id="numberOfPeople" value={numberOfPeople}
    onChange={(e) => setNumberOfPeople(parseInt(e.target.value))}
    min="1"
  />
</div>
```

8. The above code creates a labeled input field for specifying the number of people. It uses an `<input>` element of type number with a minimum value of 1 and updates the `numberOfPeople` state with the parsed integer value entered by the user.

9. To display the meal items, you must traverse the array using `map()` within `<div>` with class name **meal_selection**, as you did for the additions.

```
<div className="meal_selection">
  {mealsItems.map((item, index) => (
    <div className="meal_item" key={index} style={{ padding: 15 }}>
      <div className="inner">
        <input type="checkbox" id={`meal_${index}`}
          checked={item.selected}
          onChange={() => handleMealSelection(index)}
        />
        <label htmlFor={`meal_${index}`}> {item.name} </label>
      </div>
      <div className="meal_cost">${item.cost}</div>
    </div>
  ))}
</div>
```

10. Let's step through this code.

- `<div className="meal_selection">` This is a container for the list of meal items.
- `{mealsItems.map((item, index) => (...))}` This maps over an array of `mealsItems` and generates HTML for each item using the provided function.
- `<div className="meal_item" key={index} style={{ padding: 15 }}>` This is a container for each meal item. The `key` prop is necessary for React to keep track of each item in the list.
- `<input type="checkbox" id={meal_${index}} checked={item.selected} onChange={() => handleMealSelection(index)} />` This is a checkbox input element. The `selected` property of the current item controls its `checked` property. When the checkbox state changes, it triggers the `handleMealSelection()` function with the current item's `index`.
- `<label htmlFor={meal_${index}}>{item.name}</label>` This label is associated with the checkbox. Clicking on the label toggles the checkbox.
- `<div className="meal_cost">${item.cost}</div>` This displays the cost of each meal item.

11. Now you need to create logic in the `handleMealSelection()` function in the `ConferenceEvent.jsx` file to calculate the meal subtotal based on the number of people.

```
const handleMealSelection = (index) => {
```

```
const item = mealsItems[index];
if (item.selected && item.type === "mealForPeople") {
  // Ensure numberOfPeople is set before toggling selection
  const newNumberOfPeople = item.selected ? numberOfPeople : 0;
  dispatch(toggleMealSelection(index, newNumberOfPeople));
}
else {
  dispatch(toggleMealSelection(index));
}
};
```

12. Here's the explanation of the `handleMealSelection()` function.

- The function takes an index parameter of the meal item that triggered the selection
- It retrieves the meal item object from the `mealsItems` array using the provided index
- It checks if the retrieved item is both selected, `item.selected === true` and that its type is `mealForPeople`
- If these two conditions **are met**, it prepares to update the `numberOfPeople` state variable before toggling the selection
- If the item is of type `mealForPeople` and already selected, `item.selected` is true, it maintains the current `numberOfPeople`
- If not selected, it sets `numberOfPeople` to 0
- It dispatches the `toggleMealSelection` action with the index of the item and, if applicable, the new `numberOfPeople`
- If the item is not of type `mealForPeople` or is not selected, it dispatches an action to toggle the meal selection without any additional considerations

13. In the above `handleMealSelection()` function, you are dispatching the `toggleMealSelection` function from the `mealsSlice.jsx` file. For this, make sure that you have imported `toggleMealSelection` from `"./mealsSlice"` ;

```
import { toggleMealSelection } from "./mealsSlice";
```

14. Now, create logic to calculate the total cost for selected meal items based on the number of people within the `calculateTotalCost` function.

```
const calculateTotalCost = (section) => {
  let totalCost = 0;
  if (section === "venue") {
    venueItems.forEach((item) => {
      totalCost += item.cost * item.quantity;
    });
  } else if (section === "av") {
    avItems.forEach((item) => {
      totalCost += item.cost * item.quantity;
    });
  } else if (section === "meals") {
    mealsItems.forEach((item) => {
      if (item.selected) {
        totalCost += item.cost * numberOfPeople;
      }
    });
  }
  return totalCost;
};
```

Let us breakdown above code:

- Condition Check (`if (section === "meals")`): The code first checks if the variable `section` is equal to the string `"meals"` . If this condition is true, the code inside the `if` block will execute.
- Iterating Through `mealsItems` (`mealsItems.forEach((item) => { ... })`): `mealsItems` is assumed to be an array of objects. The `forEach` method is used to iterate over each element (referred to as `item`).
- Checking If Item is Selected (`if (item.selected) { ... }`): For each item in the `mealsItems` array, the code checks if the `selected` property of the item is true. If the item is selected, the code inside this `if` block will execute.
- Calculating Total Cost (`totalCost += item.cost * numberOfPeople;`): If the item is selected, its cost is multiplied by the `numberOfPeople`.
- The result of this multiplication is then added to `totalCost`. This implies that `totalCost` is accumulating the cost of all selected items, scaled by the number of people.

15. Create one variable named `mealsTotalCost` and call this function by passing **meals** as a parameter.

```
const mealsTotalCost = calculateTotalCost("meals");
```

16. Include `mealsTotalCost` within the meal section to display the total cost of selected meals.

```
<div className="total_cost">Total Cost: {mealsTotalCost}</div>
```

Task 6: Display selected items in a table

17. Next, you will write code to display the user-selected items and cost details in a table format. These details will appear when the user clicks on the **Show Details** button in the header.

18. The `ConferenceEvent.jsx` component contains the following code after the ternary operator .:

```
<div className="total_amount_detail">
  <TotalCost totalCosts={ totalCosts } ItemsDisplay={() => <ItemsDisplay items={ items } /> } />
```

</div>

19. Let's walk through this code snippet.

- The `TotalCost` component renders inside a `<div>` element with the class name `total_amount_detail`.
- The `TotalCost` component receives the props `totalCosts` and `ItemsDisplay`.
- The `totalCosts` prop contains cost data and the `ItemsDisplay()` component with `items` is passed as props to the `TotalCost` component.

20. Let's look at the `totalCosts` and `ItemsDisplay()` props next.

21. Now, you need to calculate the subtotal for each of the three selection types, including venue, add-ons, and meals.

22. Create one object named `totalCosts` which includes all three subtotals in the total cost.

```
const totalCosts = {
  venue: venueTotalCost,
  av: avTotalCost,
  meals: mealsTotalCost,
};
```

23. Include the above code before the return of the function component, `ConferenceEvents.jsx`

24. To get the selected items, you need to create logic within the `getItemsFromTotalCost()` function. You can review the code for that function below and read an explanation of the code after the snippet.

```
const getItemsFromTotalCost = () => {
  const items = [];
  venueItems.forEach((item) => {
    if (item.quantity > 0) {
      items.push({ ...item, type: "venue" });
    }
  });
  avItems.forEach((item) => {
    if (
      item.quantity > 0 &&
      !items.some((i) => i.name === item.name && i.type === "av")
    ) {
      items.push({ ...item, type: "av" });
    }
  });
  mealsItems.forEach((item) => {
    if (item.selected) {
      const itemForDisplay = { ...item, type: "meals" };
      if (item.numberOfPeople) {
        itemForDisplay.numberOfPeople = numberOfPeople;
      }
      items.push(itemForDisplay);
    }
  });
  return items;
};
```

25. The `getItemsFromTotalCost()` function will store all items the user selected in an array by creating an empty array named `items`.

- You can see each item type, venue, av, and meals, defines its own `forEach()` function. These functions look for and include only `items` in the array the user selects. These functions also label each item in the array, "venue", "av", or "meals".
- The function returns the `items` array, comprising items from each of the three categories: venue, AV, and meals.

26. The `getItemsFromTotalCost()` function is called in the `ConferenceEvent.jsx` component as `const items = getItemsFromTotalCost();` provided in the code you have **been given**.

27. Now, you need to create logic to display the `items` details as a table. In the `ConferenceEvent.jsx` component you will see an empty component, `ItemsDisplay` which has the `items` variable passed as props within `{}`.

28. The code for this component **is shown** below. After reviewing the code below, enter it into the `ItemsDisplay` component.

```
const ItemsDisplay = ({ items }) => {
  console.log(items);
  return <>
    <div className="display_box1">
      {items.length === 0 && <p>No items selected</p>}
      <table className="table_item_data">
        <thead>
          <tr>
            <th>Name</th>
            <th>Unit Cost</th>
            <th>Quantity</th>
            <th>Subtotal</th>
          </tr>
        </thead>
        <tbody>
          {items.map((item, index) => (
            <tr key={index}>
              <td>{item.name}</td>
              <td>${item.cost}</td>
              <td>
                {item.type === "meals" || item.numberOfPeople
                  ? `For ${numberOfPeople} people`
                  : item.quantity}
              </td>
              <td>{item.type === "meals" || item.numberOfPeople
                ? `${item.cost * numberOfPeople}`
                : item.cost}
              </td>
            </tr>
          ))}
        </tbody>
      </table>
    </div>
  </>
```

```

      : `${item.cost * item.quantity}`)
    </td>
  </tr>
)
  </tbody>
</table>
</div>
</>
};

```

29. Let's review how this code works.

First, the component writes the list of items to the console, which helps with testing.

The component returns a `<div>` element with the class name `display_box1`. Within this element:

- It displays the message "No items selected" if there are no items in the `items` array.
- If the array contains items, it displays a table with the class name "table_item_data".
- The table layout has four columns: "Name", "Unit Cost", "Quantity", and "Subtotal".
- It iterates over the `items` array using the `map()` function. Each item in the `items` array creates a table row, `<tr>` element.
- Each row of the table displays the following information, respectively
 - The name of the item
 - The item's unit price with a dollar sign in front of it
 - For rooms and add-ons, it displays the quantity of the item
 - For meals, it uses the `numberOfPeople` property to display "For x people" where x is the number of people.
 - It calculates the subtotal for each item type by multiplying the item unit cost by the quantity or the number of people.

Task 7: Write the TotalCost component

1. Now, you need to create the logic to display the items in the table when the user selects the Show Details button.

2. Navigate to the `TotalCost.jsx` component under the `src` folder. You will see the basic layout provided. It should look like the following:

```

import React, { useState, useEffect } from 'react';
import './TotalCost.css';
const TotalCost = ({ totalCosts, ItemsDisplay }) => {
  return (
    <div className="pricing-app">
      <div className="display_box">
        <div className="header">
          <p className="preheading"><h3>Total cost for the event</h3></p>
        </div>
        <div>
          <h2 id="pre_fee_cost_display" className="price"></h2>
          <div className="render_items"></div>
        </div>
      </div>
    </div>
  );
};
export default TotalCost;

```

3. Let's review this code.

- The import statements at the beginning of your file ensure access to `React`, `useState`, `useEffect`, and the stylesheet `./TotalCost.css`
- The `TotalCost()` function component takes the props `totalCosts` and `ItemsDisplay` as parameters.
- The `TotalCost` component structure includes multiple `<div>` elements along with their class names.
- In the `ConferenceEvent.jsx` component, one object named `totalCosts` aggregates the total costs for venue, audio-visual (AV), and meals.
- It assigns them to their respective properties `venue`, `av`, and `meals`.

4. Now, create a variable named `total_amount` to get the sum of the total costs for the venue, audio-visual (AV), and meals.

5. You can calculate this value by adding the values through the `totalCosts` prop parameter, `totalCosts.venue`, `totalCosts.av`, and `totalCosts.meals` as shown below:

```
const total_amount = totalCosts.venue + totalCosts.av + totalCosts.meals;
```

6. Include the above code before the return of the function component.

7. Now display the `total_amount` variable within an `<h2>` element with the id set to `pre_fee_cost_display`, using the code below:

```

<h2 id="pre_fee_cost_display" className="price">
  ${total_amount}
</h2>

```

8. Next, include the `ItemsDisplay` prop within a `<div>` tag with the class name `render_items` to render items details using the `ItemsDisplay`:

```

<div className="render_items">
  <ItemsDisplay />
</div>

```

Task 8: Check The output

Now, you should have all the code you need and can check the output of your table.

1. Save all the files and re-run the application. If you already have the application open in the browser, you can refresh the application page after running it again.
2. Click the **Get Started** button.
3. Add multiple items from the venue, add-ons, and meals sections. Also, enter the number of people in the meals section.
4. Select **Show Details** in the navbar. Your output will be displayed according to what you selected on the product selection page. Here is an example of what your output might look like.

TOTAL COST FOR THE EVENT

\$24310

Name	Unit Cost	Quantity
Auditorium Hall (Capacity:200)	\$5500	3
Presentation Room (Capacity:50)	\$700	3
Speaker	\$35	2
Microphones	\$45	2
Breakfast	\$50	For 30 people

Summary and Final Solutions

In this project, you:

- Created function components
- Review the structure of the ConferenceEvent.jsx component
- Used Redux Toolkit slices to manage different parts of your application's states
- Implemented Redux actions, reducers and the Redux store to increment and decrement stored values and dynamically display those values.
- Calculated and displayed costs using arrays and the `map()` function to iterate over the arrays
- Made a dynamic table to show selected products, displaying the item names, unit cost, quantity, and subtotal for that item

Now you are prepared to develop the shopping cart application for your final project!

Solutions

Solution for `avSlice.jsx`

► [Click here to see solution](#)

Solution for `ConferenceEvent.jsx`

► [Click here to see solution](#)

Solution for `mealsSlice.jsx`

► [Click here to see solution](#)

Solution for `store.js`

► [Click here to see solution](#)

Solution for `TotalCost.jsx`

► [Click here to see solution](#)

Solution for `VenueSlice.jsx`

► [Click here to see solution](#)

Author(s)

Richa Arora
Bethany Hudnutt

© IBM Corporation. All rights reserved.