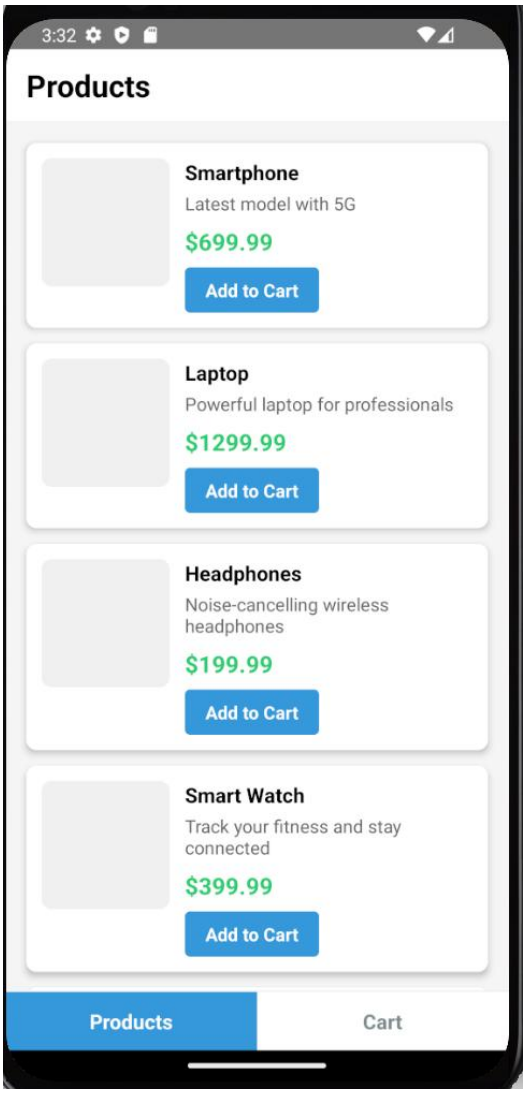
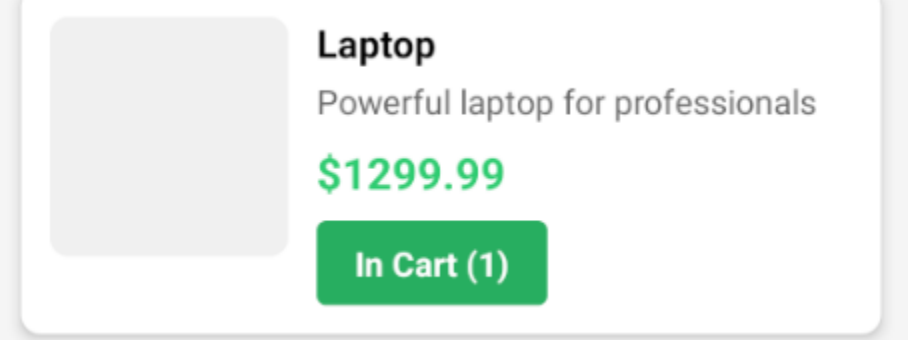
Lab7 Report

Date: 2025/11/11

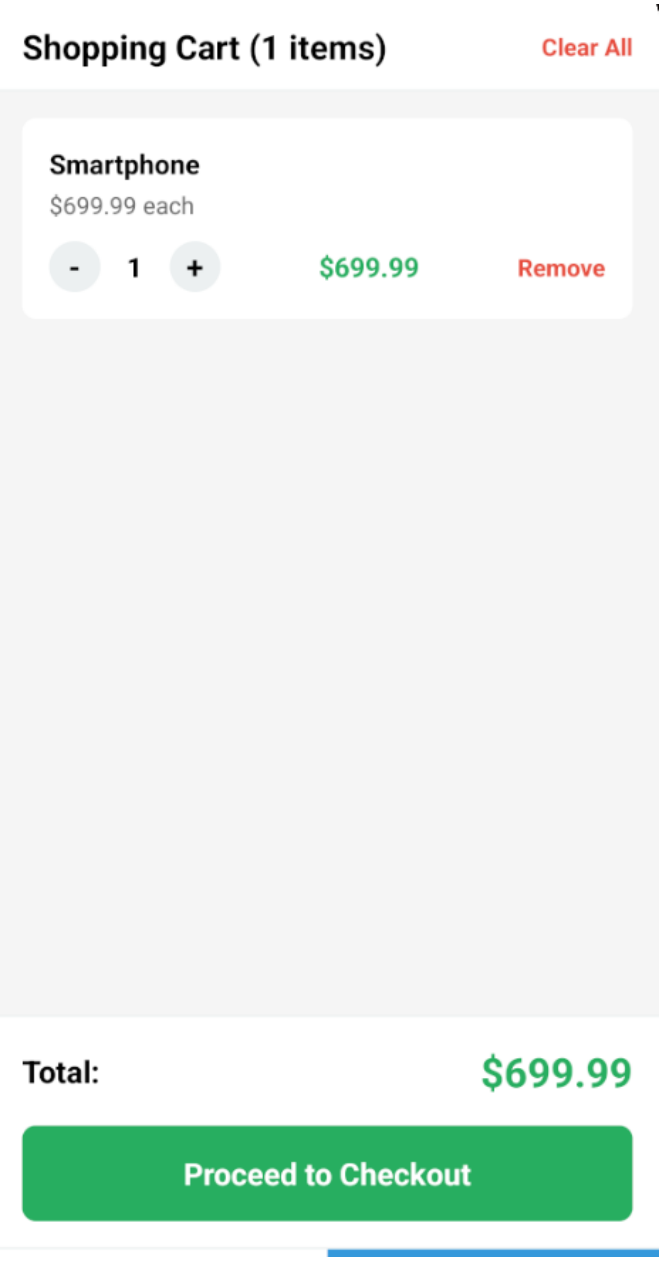
Products list



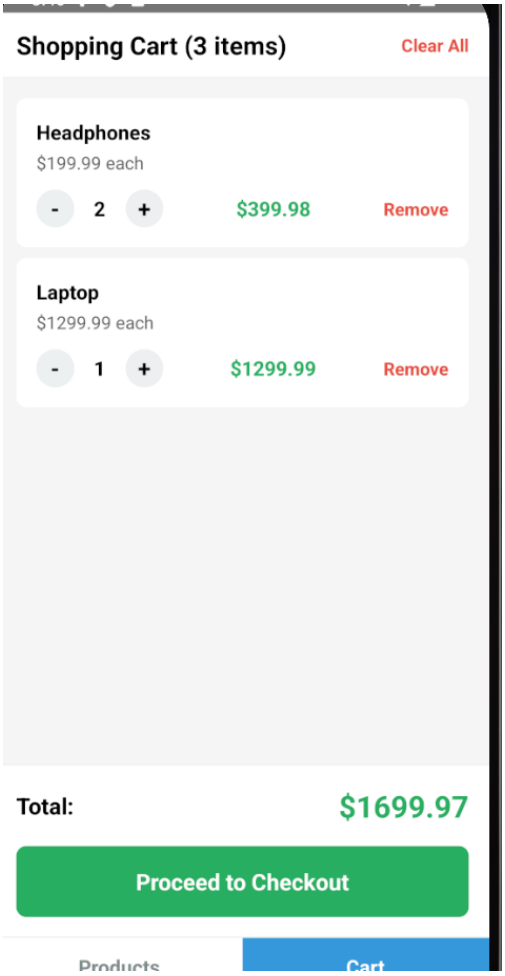
Product detail page



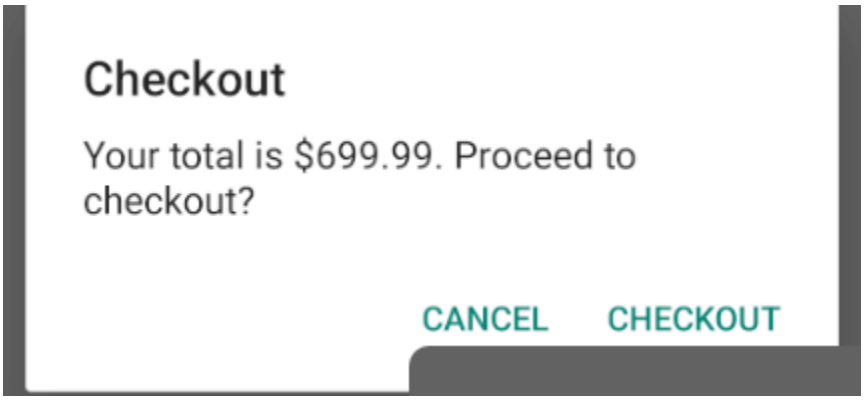
Cart with 1 item



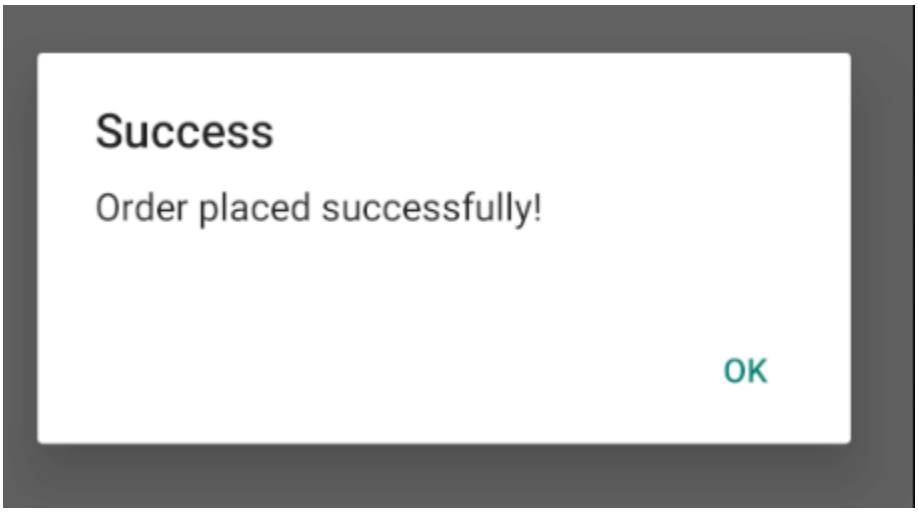
Cart with 3 items



Check out the confirmation dialogue



Order success message



Challenges and solutions that I faced during the lab

Challenge 1: State Management problems

Problem: Managing cart state across multiple components became I had trouble with prop drilling. The cart data needed to be accessible from ProductList, ProductDetail, Cart, and Checkout components.

Solution: Implemented Redux Toolkit to create a centralized store for cart state. This got rid of prop drilling and made the state accessible from any component.

Before: Prop drilling hell

<ProductList cart={cart} onAddToCart={handleAddToCart} />

<ProductDetail cart={cart} onAddToCart={handleAddToCart} />

After: Clean Redux approach

<ProductList />

<ProductDetail />

Challenge 2: Immutable State Updates

Problem: I tried to mutate the state directly, which caused rendering issues and bugs.

Solution: I had to learn how to use Redux Toolkit's createSlice with Immer.js, which allows writing "mutating" logic that actually creates immutable updates.

Incorrect: Direct mutation

```
state.items[0].quantity++
```

Correct: Immutable update

```
state.items[0] = {
  ...state.items[0],
  quantity: state.items[0].quantity + 1
}
```

Challenge 3: Complex Cart Operations

Problem: Handling multiple cart operations (add, remove, update quantity) while maintaining data consistency.

Solution: Created dedicated action creators for each operation and ensured the reducer handled edge cases.

```
addItem: (state, action) => {
  const existingItem = state.items.find(item => item.id === action.payload.id);
  if (existingItem) {
    existingItem.quantity += 1;
  } else {
    state.items.push({ ...action.payload, quantity: 1 });
  }
}
```

Challenge 4: Asynce State Updates with UI feedback

Problem: When users added items to the cart, there was no immediate visual feedback, making the app feel unresponsive. Users would click "Add to Cart" multiple times, unsure if their action was registered, which sometimes led to duplicate additions or confusing state.

Solution: Implemented a temporary UI state management pattern alongside Redux to provide instant feedback. I added a local "adding" state that shows a loading spinner or success checkmark immediately when the button is clicked, while the Redux state update processes in the background.

What you learned about Redux

1. Predictable State Management
    a. Single Source of Truth: Debugging is made easier because the entire app state is kept in a single store.
    b. State is Read-Only: Actions are the only way to change state.
    c. Pure Functions Are Used to Make Changes: Reducers define how state changes in reaction to actions.

2. Redux Toolkit Makes Conventional Redux Easier
    a. Less boilerplate: action creators and action types are automatically generated by createSlice.
    b. b. Integrated Immer: Enables the creation of more straightforward immutable update logic
    c. c. Integration of Developer Tools: Redux DevTools makes debugging simple.

3. Architecture of Components

a. Presentational versus Container Components: State management and UI logic are kept apart

b. Selectors for Derived Data: Total price and item counts were calculated using selectors.

c. Components that dispatch actions do not understand how they are handled.

Patterns in the Real World

4. Real-World Patterns

   Pattern 1: Normalized state structure

   ```
   {
    items: [{id, name, price, quantity}],
    total: 1699.97,
    itemCount: 3
   }
   ```

   /Pattern 2: Action payload standardization

   ```
   dispatch(addToCart({ id: 1, name: 'Laptop', price: 1299.99 }))
   ```

   // Pattern 3: Selector composition

   ```
   const total = useSelector(selectTotalPrice)
   const itemCount = useSelector(selectItemCount)
   ```

5. Benefits Realized
   a. Debugging: Time-travel debugging with Redux DevTools
   b. Testability: Pure reducers are easy to test
   c. Scalability: Easy to add new features without breaking existing ones
   d. Maintainability: Clear data flow makes the app easier to understand
   e.