Experiment No. 3

AIM: Understanding State Management in Flutter

Theory:

State management is a fundamental concept in Flutter that helps maintain and update the state of widgets in a reactive way. Understanding state management is crucial for building efficient and scalable applications. There are multiple approaches to state management in Flutter:

1. **Understanding State in Flutter:**
   - Flutter applications are built using widgets that can either be stateless or stateful.
   - **Stateless Widgets**: These widgets do not change once built.
   - **Stateful Widgets**: These widgets maintain a mutable state that can change over time.
2. **Managing State Locally:**
   - The simplest way to manage state is by using setState() within a StatefulWidget.
   - This approach is useful for small applications but can be inefficient for complex applications.

   Example:

```
class Counter extends StatefulWidget {
  @override
  _CounterState createState() => _CounterState();
}

class _CounterState extends State<Counter> {
  int _count = 0;

  void _incrementCounter() {
    setState(() {
      _count++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Text('Count: $_count'),
        ElevatedButton(onPressed: _incrementCounter, child: Text('Increment')),
      ],
    );
  }
}
```

3. **Using InheritedWidget for Propagation:**
    - InheritedWidget helps pass data down the widget tree without requiring direct propagation.
    - It is useful for sharing app-wide state but can become complex when managing multiple state changes.
4. **Provider for State Management:**
    - Provider is a recommended approach for scalable state management in Flutter.
    - It helps separate business logic from UI and makes state changes more efficient.

    Example of Using Provider:

    ```
    class CounterModel extends ChangeNotifier {
      int _count = 0;
      int get count => _count;

      void increment() {
        _count++;
        notifyListeners();
      }
    }
    ```

5. **Other State Management Approaches:**
    - **Riverpod**: A more modern and simplified version of Provider.
    - **Bloc (Business Logic Component)**: Separates UI and business logic using streams.
    - **Redux**: Inspired by React, useful for large applications with complex state interactions.
6. **Best Practices for State Management:**
    - Choose the appropriate state management approach based on app complexity.
    - Use setState() for simple state management within a widget.
    - Use Provider or Riverpod for moderate complexity applications.
    - Use Bloc or Redux for large-scale applications needing clear separation of concerns.
7. **Performance Optimization in State Management:**
    - Minimize unnecessary widget rebuilds.
    - Use const constructors for stateless widgets.
    - Optimize large lists using ListView.builder.

Screenshots:

(Include relevant screenshots of state management implementation)

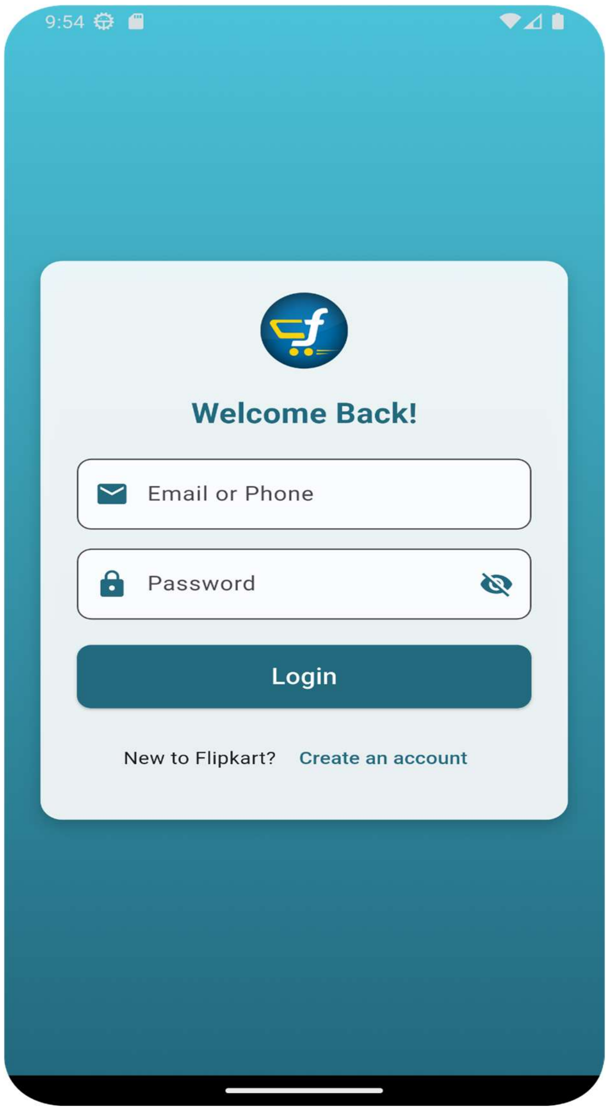
Code Snippets:

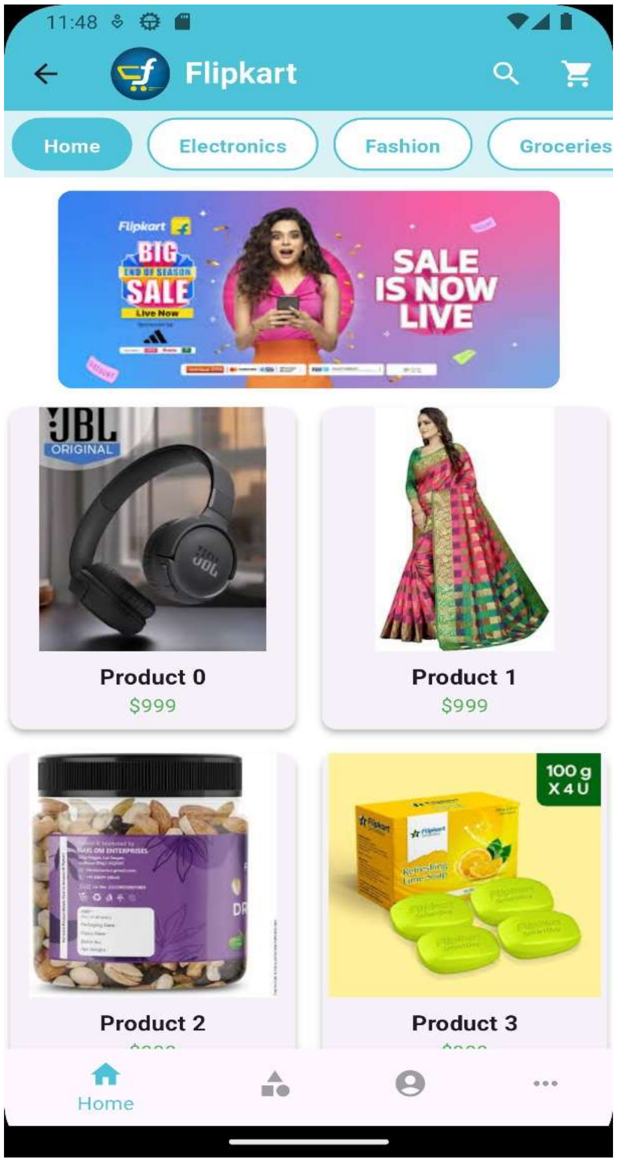
1. **Basic Provider Implementation:**

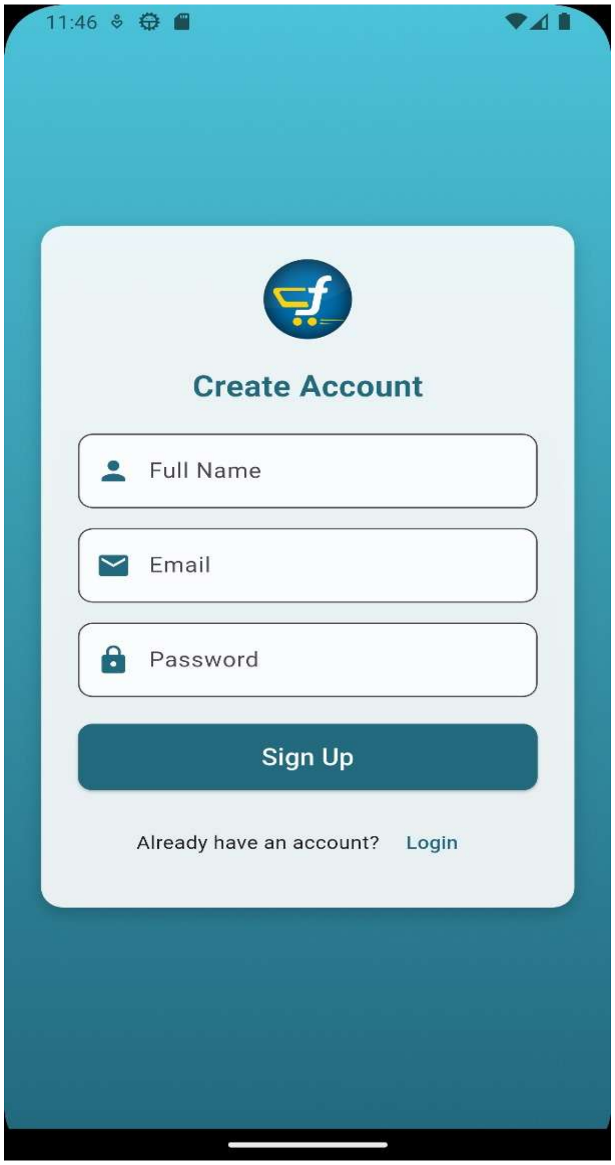
```
Consumer<CounterModel>(
  builder: (context, counter, child) {
    return Text('Count: ${counter.count}');
  },
)
```

6. **Bloc Implementation:**

BlocBuilder<CounterBloc, int>(
7. builder: (context, count) {
8. return Text('Count: $count');
9. },
10.)




11:48
Flipkart
Home
Electronics
Fashion
Groceries
SALE IS NOW LIVE
Product 0
$999
Product 1
$999
Product 2
Product 3
Home


11:46
Create Account
Full Name
Email
Password
Sign Up
Already have an account?
Login