Flutter Habit Tracker App

Prepared by:
Bwar Kamaran Kamal
IT23015

12-4-2024

Table of Contents

---

1. Introduction

The Flutter Habit Tracker app is a mobile application designed to help users build and maintain good habits. By providing a user-friendly interface and essential features such as habit creation, tracking, and analytics, the app aims to empower individuals to achieve their personal goals effectively.

---

2. Project Overview

Purpose

The primary objective of the Habit Tracker app is to assist users in tracking their daily habits, monitoring progress, and fostering consistency. The app provides functionalities to add, edit, delete, and visualize habits, ensuring users remain motivated and informed about their habit-forming journey.

Key Features

- **Habit Management:** Add, edit, delete habits.
- **Completion Tracking:** Mark habits as completed or incomplete.
- **Progress Visualization:** View statistics and progress over time.
- **Data Persistence:** Save habits and progress across app sessions.
- **Theming:** Toggle between light and dark modes.

- **Custom Navigation:** Utilize a centralized navigation drawer for easy access to different sections.

---

3. Technologies Used

- **Flutter:** An open-source UI software development toolkit by Google for building natively compiled applications.
- **Dart:** The programming language used to develop Flutter apps.
- **Provider:** A state management solution for Flutter, facilitating efficient and scalable state management.
- **Shared Preferences:** A Flutter plugin used for persisting simple data in key-value pairs.
- **Material Design:** Flutter's implementation of Google's Material Design for building visually appealing UIs.

---

4. Project Structure

The project is organized into several directories to maintain a clean and manageable codebase:

```
css
Copy code
lib/
├── main.dart
├── models/
│   ├── habit.dart
│   ├── note.dart
│   └── journal_entry.dart
├── providers/
│   ├── habits_provider.dart
│   ├── notes_provider.dart
│   ├── journal_provider.dart
│   └── theme_notifier.dart
├── screens/
│   ├── home_screen.dart
│   ├── add_habit_screen.dart
│   ├── progress_statistics_screen.dart
│   └── calendar_view_screen.dart
├── widgets/
│   ├── custom_app_bar.dart
│   └── app_drawer.dart
└── drawer_screens.dart
```

Directory Breakdown:

- **`models/`:** Contains data classes representing the core data structures (Habit, Note, JournalEntry).

- **`providers/`:** Holds classes that manage state and business logic, extending `ChangeNotifier`.
- **`screens/`:** Contains UI screens/pages of the app (HomeScreen, AddHabitScreen).
- **`widgets/`:** Contains reusable UI components/widgets (CustomAppBar, AppDrawer).
- **`drawer_screens.dart`:** Contains screen widgets accessible via the navigation drawer.

---

5. State Management with Provider

Provider Package

Definition:
Provider is a Flutter package that offers a simple and efficient way to manage and propagate state changes across the widget tree.

Advantages:

- **Simplicity:** Easy to understand and implement.
- **Performance:** Efficiently rebuilds only the widgets that depend on the changed state.
- **Flexibility:** Supports various state management patterns.

Key Components

1. **ChangeNotifier:**
   A class that provides change notification to its listeners. When a ChangeNotifier's state changes, it can call `notifyListeners()` to alert any listeners.
2. **ChangeNotifierProvider:**
   A widget that provides an instance of ChangeNotifier to its descendants.
3. **MultiProvider:**
   Allows providing multiple providers at once, reducing boilerplate.
4. **Consumer Widget:**
   Listens to changes in a provider and rebuilds when notified, enabling selective rebuilding.

Usage in Project

- **Providing State:**

  ```
  dart
  Copy code
  MultiProvider(
    providers: [
      ChangeNotifierProvider(create: (_) => ThemeNotifier()),
      ChangeNotifierProvider(create: (_) => HabitsProvider()),
      ChangeNotifierProvider(create: (_) => NotesProvider()),
      ChangeNotifierProvider(create: (_) => JournalProvider()),
    ],
  ```

```
  child: const MyApp(),
);
```

- **Consuming State:**

```
dart
Copy code
final habitsProvider = Provider.of<HabitsProvider>(context);
final habits = habitsProvider.habits;
```

---

6. Data Persistence with SharedPreferences

Definition

SharedPreferences is a Flutter plugin that provides persistent storage for simple data (key-value pairs). It's used to store user preferences, settings, or small amounts of app data.

Usage in Project

- **Saving Data:**

```
dart
Copy code
Future<void> _saveHabits() async {
  final prefs = await SharedPreferences.getInstance();
  final String encodedData = jsonEncode(_habits.map((e) =>
e.toMap()).toList());
  prefs.setString('habits', encodedData);
}
```

- **Loading Data:**

```
dart
Copy code
Future<void> _loadHabits() async {
  final prefs = await SharedPreferences.getInstance();
  final String? habitsData = prefs.getString('habits');
  if (habitsData != null) {
    final List<dynamic> decodedData = jsonDecode(habitsData);
    _habits = decodedData.map((item) => Habit.fromMap(item)).toList();
    notifyListeners();
  }
}
```

Key Points:

- **Serialization:** Convert complex objects into JSON strings before saving.
- **Deserialization:** Convert JSON strings back into objects when loading.
- **Persistence:** Data remains available across app launches.

7. Models

a. Habit

Definition:
Represents a habit that the user wants to track.

Implementation:

```
dart
Copy code
class Habit {
  String name;
  bool isCompleted;
  List<DateTime> completionDates;

  Habit(this.name, {this.isCompleted = false, List<DateTime>?
completionDates})
      : completionDates = completionDates ?? [];

  // Convert a Habit into a Map
  Map<String, dynamic> toMap() {
    return {
      'name': name,
      'isCompleted': isCompleted,
      'completionDates':
          completionDates.map((date) => date.toIso8601String()).toList(),
    };
  }

  // Create a Habit from a Map
  factory Habit.fromMap(Map<String, dynamic> map) {
    return Habit(
      map['name'],
      isCompleted: map['isCompleted'],
      completionDates: (map['completionDates'] as List<dynamic>)
          .map((dateStr) => DateTime.parse(dateStr))
          .toList(),
    );
  }
}
```

b. Note

Definition:
Represents a simple note that the user can create.

Implementation:

```
dart
Copy code
class Note {
  String title;
  String content;

  Note({required this.title, required this.content});

  // Convert a Note into a Map
  Map<String, dynamic> toMap() {
    return {
      'title': title,
      'content': content,
    };
  }

  // Create a Note from a Map
  factory Note.fromMap(Map<String, dynamic> map) {
    return Note(
      title: map['title'],
      content: map['content'],
    );
  }
}
```

c. JournalEntry

Definition:
Represents a journal entry with a title, content, and date.

Implementation:

```
dart
Copy code
class JournalEntry {
  String title;
  String content;
  DateTime date;

  JournalEntry({required this.title, required this.content, required
this.date});

  // Convert a JournalEntry into a Map
  Map<String, dynamic> toMap() {
    return {
      'title': title,
      'content': content,
      'date': date.toIso8601String(),
    };
  }

  // Create a JournalEntry from a Map
  factory JournalEntry.fromMap(Map<String, dynamic> map) {
    return JournalEntry(
      title: map['title'],
      content: map['content'],
      date: DateTime.parse(map['date']),
    );
  }
}
```

Key Points Across Models:

- **Serialization & Deserialization:**
  Each model includes `toMap` and `fromMap` methods to facilitate storing and retrieving complex objects using SharedPreferences.
- **Constructors:**
  The `fromMap` factory constructor allows creating instances of models directly from a Map, simplifying data retrieval.

---

8. Providers

a. ThemeNotifier

Purpose:
Manages the app's theme (light or dark mode).

Implementation:

```
dart
Copy code
class ThemeNotifier extends ChangeNotifier {
  ThemeMode _themeMode = ThemeMode.light;

  ThemeMode get themeMode => _themeMode;

  ThemeNotifier() {
    _loadTheme();
  }

  void toggleTheme(bool isDarkMode) {
    _themeMode = isDarkMode ? ThemeMode.dark : ThemeMode.light;
    _saveTheme(isDarkMode);
    notifyListeners();
  }

  Future<void> _loadTheme() async {
    final prefs = await SharedPreferences.getInstance();
    final isDark = prefs.getBool('isDarkMode') ?? false;
    _themeMode = isDark ? ThemeMode.dark : ThemeMode.light;
    notifyListeners();
  }

  Future<void> _saveTheme(bool isDarkMode) async {
    final prefs = await SharedPreferences.getInstance();
    prefs.setBool('isDarkMode', isDarkMode);
  }
}
```

b. HabitsProvider

Purpose:
Manages the list of habits, including adding, editing, deleting, and toggling completion status.

Implementation:

```
dart
Copy code
class HabitsProvider extends ChangeNotifier {
  List<Habit> _habits = [];

  List<Habit> get habits => _habits;

  HabitsProvider() {
    _loadHabits();
  }

  // Load habits from SharedPreferences
  Future<void> _loadHabits() async {
    final prefs = await SharedPreferences.getInstance();
    final String? habitsData = prefs.getString('habits');
    if (habitsData != null) {
      final List<dynamic> decodedData = jsonDecode(habitsData);
      _habits = decodedData.map((item) => Habit.fromMap(item)).toList();
      notifyListeners();
    }
  }

  // Save habits to SharedPreferences
  Future<void> _saveHabits() async {
    final prefs = await SharedPreferences.getInstance();
    final String encodedData =
        jsonEncode(_habits.map((e) => e.toMap()).toList());
    prefs.setString('habits', encodedData);
  }

  // Add a new habit
  void addHabit(String habitName) {
    _habits.insert(0, Habit(habitName));
    _saveHabits();
    notifyListeners();
  }

  // Edit an existing habit
  void editHabit(int index, String newName) {
    _habits[index].name = newName;
    _saveHabits();
    notifyListeners();
  }
  // Toggle habit completion
  void toggleHabitCompletion(int index) {
    _habits[index].isCompleted = !_habits[index].isCompleted;
    if (_habits[index].isCompleted) {
      DateTime today = DateTime.now();
      if (!_habits[index].completionDates.any((date) =>
          date.year == today.year &&
          date.month == today.month &&
          date.day == today.day)) {
        _habits[index].completionDates.add(today);
      }
    } else {
      _habits[index].completionDates.removeWhere((date) {
        DateTime today = DateTime.now();
        return date.year == today.year &&
            date.month == today.month &&
            date.day == today.day;
      });
    }
    _saveHabits();
    notifyListeners();
  }

  // Delete a habit
  void deleteHabit(int index) {
    _habits.removeAt(index);
    _saveHabits();
    notifyListeners();
  }

  // Delete all completed habits
  void deleteCompletedHabits() {
    _habits.removeWhere((habit) => habit.isCompleted);
    _saveHabits();
    notifyListeners();
  }
}
```

c. NotesProvider

Purpose:
Manages the list of notes, including adding and deleting notes.

Implementation:

```
dart
Copy code
class NotesProvider extends ChangeNotifier {
  List<Note> _notes = [];

  List<Note> get notes => _notes;

  NotesProvider() {
    _loadNotes();
  }

  // Load notes from SharedPreferences
  Future<void> _loadNotes() async {
    final prefs = await SharedPreferences.getInstance();
    final String? notesData = prefs.getString('notes');
    if (notesData != null) {
      final List<dynamic> decodedData = jsonDecode(notesData);
      _notes = decodedData.map((item) => Note.fromMap(item)).toList();
      notifyListeners();
    }
  }

  // Save notes to SharedPreferences
  Future<void> _saveNotes() async {
    final prefs = await SharedPreferences.getInstance();
    final String encodedData =
        jsonEncode(_notes.map((e) => e.toMap()).toList());
    prefs.setString('notes', encodedData);
  }

  // Add a new note
  void addNote(Note note) {
    _notes.insert(0, note);
    _saveNotes();
    notifyListeners();
  }

  // Delete a note
  void deleteNote(int index) {
    _notes.removeAt(index);
    _saveNotes();
    notifyListeners();
  }
}
```

d. JournalProvider

Purpose:
Manages the list of journal entries, including adding and deleting entries.

Implementation:

```
dart
Copy code
class JournalProvider extends ChangeNotifier {
  List<JournalEntry> _entries = [];

  List<JournalEntry> get entries => _entries;

  JournalProvider() {
    _loadEntries();
  }

  // Load journal entries from SharedPreferences
  Future<void> _loadEntries() async {
    final prefs = await SharedPreferences.getInstance();
    final String? entriesData = prefs.getString('journal_entries');
    if (entriesData != null) {
      final List<dynamic> decodedData = jsonDecode(entriesData);
      _entries = decodedData.map((item) =>
JournalEntry.fromMap(item)).toList();
      notifyListeners();
    }
  }

  // Save journal entries to SharedPreferences
  Future<void> _saveEntries() async {
    final prefs = await SharedPreferences.getInstance();
    final String encodedData =
        jsonEncode(_entries.map((e) => e.toMap()).toList());
    prefs.setString('journal_entries', encodedData);
  }

  // Add a new journal entry
  void addEntry(JournalEntry entry) {
    _entries.insert(0, entry);
    _saveEntries();
    notifyListeners();
  }

  // Delete a journal entry
  void deleteEntry(int index) {
    _entries.removeAt(index);
    _saveEntries();
    notifyListeners();
  }
}
```

Key Points Across Providers:

- **State Management:**
  Each provider manages a specific aspect of the app's state, ensuring a clear separation of concerns.
- **Data Persistence:**
  Providers handle loading and saving data to SharedPreferences, ensuring data persists across app sessions.
- **Notification:**
  Providers call `notifyListeners()` after any state change to update dependent UI components.

---

9. Custom Widgets

a. CustomAppBar

Purpose:
A reusable AppBar widget that maintains a consistent design across different screens, incorporating features like gradients, custom fonts, and theme toggling.

Implementation:

```
dart
Copy code
// lib/widgets/custom_app_bar.dart
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import '../providers/theme_notifier.dart'; // Adjust the path as necessary

class CustomAppBar extends StatelessWidget implements PreferredSizeWidget {
  final String title;
  final List<Widget>? actions;
  final bool showLeading;
  final bool isTransparent;
  final bool showThemeToggle;

  const CustomAppBar({
    Key? key,
    required this.title,
    this.actions,
    this.showLeading = true,
    this.isTransparent = false,
    this.showThemeToggle = true,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    final themeNotifier = Provider.of<ThemeNotifier>(context);

    return AppBar(
      backgroundColor: isTransparent ? Colors.transparent :
Colors.transparent,
      elevation: 0, // Removes default shadow
      leading: showLeading
          ? IconButton(
              icon: const Icon(Icons.menu, color: Colors.white),
              onPressed: () {
                Scaffold.of(context).openDrawer();
              },
              tooltip: 'Menu',
            )
          : null,
      title: Text(
        title,
        style: const TextStyle(
          fontFamily: 'Roboto',
          fontSize: 20.0,
          fontWeight: FontWeight.bold,
          color: Colors.white,
        ),
      ),
      actions: [
        if (showThemeToggle)
          IconButton(
            icon: Icon(
              themeNotifier.isDarkMode ? Icons.dark_mode : Icons.light_mode,
            ),
            tooltip: 'Toggle Theme',
            onPressed: () {
              themeNotifier.toggleTheme(!themeNotifier.isDarkMode);
            },
          ),
        if (actions != null) ...actions!,
      ],
      centerTitle: true,
      flexibleSpace: Container(
        decoration: BoxDecoration(
          gradient: LinearGradient(
            colors: [
              themeNotifier.isDarkMode ? Colors.teal[700]! : Colors.teal,
              themeNotifier.isDarkMode ? Colors.teal[400]! :
Colors.tealAccent,
            ],
            begin: Alignment.topLeft,
            end: Alignment.bottomRight,
          ),
        ),
      ),
    );
  }

  @override
  Size get preferredSize => const Size.fromHeight(kToolbarHeight);
}
```

Key Components:

- **Parameters:**
  - `title`: The text displayed in the center of the AppBar.
  - `actions`: A list of widgets (usually IconButtons) displayed on the right side.
  - `showLeading`: Determines whether to show the leading icon (menu button).
  - `isTransparent`: If `true`, makes the AppBar background transparent, allowing underlying content to be visible.
  - `showThemeToggle`: If `true`, includes a theme toggle icon in the AppBar.
- **Features:**
  - **Gradient Background:**
    Applied via `flexibleSpace` to give a modern look.
  - **Theme Toggle:**
    An icon button that toggles between light and dark themes using `ThemeNotifier`.
  - **Conditional Rendering:**
    Widgets like the theme toggle and additional actions are rendered based on the provided parameters.

Advantages:

- **Reusability:**
  Use the same AppBar design across multiple screens without duplicating code.
- **Consistency:**
  Ensures a uniform look and feel throughout the app.
- **Customization:**
  Easily customize the AppBar's appearance and behavior via parameters.

b. AppDrawer

Purpose:
A centralized navigation drawer that provides consistent navigation options across different screens.

Implementation:

```
dart
Copy code
// lib/widgets/app_drawer.dart
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import '../providers/habits_provider.dart';
import '../models/habit.dart';
import '../screens/home_screen.dart';
import '../screens/progress_statistics_screen.dart';
import '../screens/calendar_view_screen.dart';
import '../screens/notes_screen.dart';
import '../screens/journal_screen.dart';
import '../screens/data_backup_sync_screen.dart';
import '../screens/gamification_screen.dart';
import '../screens/language_support_screen.dart';
import '../screens/accessibility_screen.dart';
import '../screens/habit_scheduling_screen.dart';
import '../screens/tutorials_help_screen.dart';

class AppDrawer extends StatelessWidget {
  const AppDrawer({super.key});

  Widget _createDrawerItem({
    required BuildContext context,
    required IconData icon,
    required String text,
    required Widget screen,
  }) {
    return ListTile(
      leading: Icon(icon),
      title: Text(text),
      onTap: () {
        Navigator.pop(context); // Close the drawer
        Navigator.pushReplacement(
          context,
          MaterialPageRoute(builder: (context) => screen),
        );
      },
    );
  }

  @override
  Widget build(BuildContext context) {
    final habitsProvider = Provider.of<HabitsProvider>(context, listen:
false);
    final habits = habitsProvider.habits;

    return Drawer(
      child: ListView(
        padding: EdgeInsets.zero,
        children: <Widget>[
          const DrawerHeader(
            decoration: BoxDecoration(
              color: Colors.teal,
            ),
            child: Text(
              'Habit Tracker Menu',
              style: TextStyle(
                color: Colors.white,
                fontSize: 24,
              ),
            ),
          ),
          _createDrawerItem(
            context: context,
            icon: Icons.home,
            text: 'Home',
            screen: const HomeScreen(),
          ),
          _createDrawerItem(
            context: context,
            icon: Icons.show_chart,
            text: 'Progress & Statistics',
            screen: ProgressStatisticsScreen(habits: habits),
          ),
          _createDrawerItem(
            context: context,
            icon: Icons.calendar_today,
            text: 'Calendar View',
            screen: CalendarViewScreen(habits: habits),
          ),
          _createDrawerItem(
            context: context,
            icon: Icons.note,
            text: 'Notes',
            screen: const NotesScreen(),
          ),
          _createDrawerItem(
            context: context,
            icon: Icons.book,
            text: 'Journal',
            screen: const JournalScreen(),
          ),
          const Divider(),
          _createDrawerItem(
            context: context,
            icon: Icons.cloud_sync,
            text: 'Data Backup & Sync',
            screen: const DataBackupSyncScreen(),
          ),
          _createDrawerItem(
            context: context,
            icon: Icons.videogame_asset,
            text: 'Gamification',
            screen: const GamificationScreen(),
          ),
          _createDrawerItem(
            context: context,
            icon: Icons.language,
            text: 'Multi-language Support',
            screen: const LanguageSupportScreen(),
          ),
          _createDrawerItem(
            context: context,
            icon: Icons.accessibility,
            text: 'Accessibility',
            screen: const AccessibilityScreen(),
          ),
          _createDrawerItem(
            context: context,
            icon: Icons.schedule,
            text: 'Habit Scheduling',
            screen: const HabitSchedulingScreen(),
          ),
          _createDrawerItem(
            context: context,
            icon: Icons.help,
            text: 'Tutorials & Help',
            screen: const TutorialsHelpScreen(),
          ),
        ],
      ),
    );
  }
}
```

Key Components:

- **DrawerHeader:**
  A header section at the top of the drawer, typically containing branding or user information.
- **ListTile Items:**
  Each navigation option is represented as a `ListTile` with an icon and text. Tapping on a tile navigates to the corresponding screen.
- **Reusable Drawer Items:**
  The `_createDrawerItem` method simplifies the creation of consistent drawer items.

Advantages:

- **Consistency:**
  Provides the same navigation options across all screens, enhancing user experience.
- **Reusability:**
  Centralizes navigation logic, making it easier to manage and update.
- **Scalability:**
  Easily add or remove navigation items as your app evolves.

---

10. AnimatedList

Definition

`AnimatedList` is a Flutter widget that displays a scrolling list of widgets that animates items as they are inserted or removed.

Why Use AnimatedList?

- **Dynamic Animations:** Provides built-in animations for adding and removing items, enhancing user experience.
- **Customization:** Allows you to define custom animations for item insertion and removal.

Key Components

1. **GlobalKey<AnimatedListState>:**
   A key that uniquely identifies the `AnimatedList` and allows you to manipulate its state (e.g., inserting or removing items).
2. **initialItemCount:**
   Specifies the initial number of items in the list.
3. **itemBuilder:**
   A function that builds each item in the list, providing the context, index, and animation.

Usage in Project

- **Initialization:**

```
dart
Copy code
class HomeScreenState extends State<HomeScreen> {
  final GlobalKey<AnimatedListState> _listKey =
GlobalKey<AnimatedListState>();

  // ... rest of the code
}
```

- **Building the List:**

```
dart
Copy code
body: habits.isEmpty
    ? Center(
        child: Text(
          "No habits added yet!",
          style: TextStyle(
            fontSize: 18,
            color: Colors.grey[600],
          ),
        ),
      )
    : AnimatedList(
        key: _listKey,
        initialItemCount: habits.length,
        padding: const EdgeInsets.all(8.0),
        itemBuilder: (context, index, animation) {
          return _buildHabitItem(habits[index], index, animation);
        },
      ),
```

- **Adding Items:**

```
dart
Copy code
floatingActionButton: FloatingActionButton(
  onPressed: () async {
    final newHabit = await Navigator.push(
      context,
      MaterialPageRoute(builder: (context) => AddHabitScreen()),
    );
    if (newHabit != null && newHabit is String) {
      habitsProvider.addHabit(newHabit);
      _listKey.currentState?.insertItem(0);
    }
  },
  child: const Icon(Icons.add),
),
```

- **Removing Items:**

```
dart
Copy code
IconButton(
  icon: const Icon(Icons.delete),
  onPressed: () {
    final removedHabit = habit;
    habitsProvider.deleteHabit(index);
    _listKey.currentState?.removeItem(
      index,
      (context, animation) => _buildHabitItem(removedHabit, index,
animation),
    );
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(
        content: Text('Habit deleted.'),
      ),
    );
  },
  tooltip: 'Delete Habit',
),
```

- **Building Each Item with Animation:**

```
dart
Copy code
Widget _buildHabitItem(Habit habit, int index, Animation<double>
animation) {
  return SizeTransition(
    sizeFactor: animation,
    child: Card(
      margin: const EdgeInsets.symmetric(vertical: 6.0),
      elevation: 2,
      child: ListTile(
        leading: CircleAvatar(
          backgroundColor: habit.isCompleted ? Colors.green :
Colors.redAccent,
          child: Icon(
            habit.isCompleted ? Icons.check : Icons.close,
            color: Colors.white,
          ),
        ),
        title: Text(
          habit.name,
          style: TextStyle(
            decoration: habit.isCompleted
                ? TextDecoration.lineThrough
                : TextDecoration.none,
          ),
        ),
        trailing: Row(
          mainAxisSize: MainAxisSize.min,
          children: [
            Switch(
              value: habit.isCompleted,
              onChanged: (_) =>
                  Provider.of<HabitsProvider>(context, listen: false)
                      .toggleHabitCompletion(index),
              activeColor: Colors.green,
            ),
            IconButton(
              icon: const Icon(Icons.edit),
              onPressed: () async {
                final editedHabitName = await Navigator.push(
                  context,
                  MaterialPageRoute(
                    builder: (context) => AddHabitScreen(
                      habitName: habit.name, // Pass the current habit
name
```

```
                    ),
                  ),
                );
                if (editedHabitName != null && editedHabitName is
String) {
                  Provider.of<HabitsProvider>(context, listen: false)
                      .editHabit(index, editedHabitName);
                  ScaffoldMessenger.of(context).showSnackBar(
                    const SnackBar(
                      content: Text('Habit updated.'),
                    ),
                  );
                }
              },
              tooltip: 'Edit Habit',
            ),
            IconButton(
              icon: const Icon(Icons.delete),
              onPressed: () {
                final removedHabit = habit;
                habitsProvider.deleteHabit(index);
                _listKey.currentState?.removeItem(
                  index,
                  (context, animation) =>
                      _buildHabitItem(removedHabit, index, animation),
                );
                ScaffoldMessenger.of(context).showSnackBar(
                  const SnackBar(
                    content: Text('Habit deleted.'),
                  ),
                );
              },
              tooltip: 'Delete Habit',
            ),
          ],
        ),
      ),
    ),
  );
}
```

Key Points:

- **AnimatedListState:**
  Allows you to perform insertions and removals with animations.
- **Animations:**
  You can customize animations by using different transition widgets like `FadeTransition`, `SlideTransition`, etc.

---

11. Navigation

Definition

Navigation in Flutter refers to the ability to move between different screens or pages within an app. Flutter provides several mechanisms to handle navigation, with `Navigator` and `Route` being the primary tools.

Navigator

Definition:
`Navigator` is a widget that manages a stack of Route objects. It provides methods to navigate between screens by pushing and popping routes.

Key Methods:

1. **`Navigator.push`:**
   Adds a new route to the top of the stack, navigating to a new screen.

   Usage:

   ```
   dart
   Copy code
   Navigator.push(
     context,
     MaterialPageRoute(builder: (context) => NewScreen()),
   );
   ```

2. **`Navigator.pop`:**
   Removes the topmost route from the stack, returning to the previous screen.

   Usage:

   ```
   dart
   Copy code
   Navigator.pop(context);
   ```

3. **`Navigator.pushReplacement`:**
   Replaces the current route with a new route.

   Usage:

   ```
   dart
   Copy code
   Navigator.pushReplacement(
     context,
     MaterialPageRoute(builder: (context) => NewScreen()),
   );
   ```

Routes

Definition:
A Route is an abstraction for a screen or page in your app. `MaterialPageRoute` is a common implementation that transitions to the new screen using a platform-specific animation.

Usage in Project

- **Navigating to `AddHabitScreen`:**

  ```
  dart
  Copy code
  final newHabit = await Navigator.push(
    context,
    MaterialPageRoute(builder: (context) => AddHabitScreen()),
  );
  ```

- **Navigating with Replacement (Using Drawer):**

  ```
  dart
  Copy code
  Navigator.pushReplacement(
    context,
    MaterialPageRoute(builder: (context) =>
  ProgressStatisticsScreen(habits: habits)),
  );
  ```

Key Points:

- **Asynchronous Navigation:**
  Using `async` and `await` allows handling results returned from the pushed route.
- **Passing Data:**
  Data can be passed to new screens via constructor parameters during navigation.

---

12. Other Widgets and Concepts

a. Scaffold

Definition:
`Scaffold` provides a high-level structure for implementing the basic material design visual layout of the app. It includes properties like `appBar`, `body`, `drawer`, `floatingActionButton`, etc.

Usage:

```
dart
Copy code
Scaffold(
```

```
  appBar: AppBar(title: Text('Home')),
  body: Center(child: Text('Hello, Flutter!')),
  drawer: AppDrawer(),
  floatingActionButton: FloatingActionButton(
    onPressed: () {},
    child: Icon(Icons.add),
  ),
);
```

b. IconButton

Definition:
An IconButton is a clickable button that contains an icon. It's commonly used in AppBars and other interactive parts of the UI.

Usage:

```
dart
Copy code
IconButton(
  icon: Icon(Icons.settings),
  onPressed: () {
    // Handle button press
  },
  tooltip: 'Settings',
),
```

c. CircleAvatar

Definition:
`CircleAvatar` is a widget that displays a circular icon or image, often used to represent users or profile pictures.

Usage:

```
dart
Copy code
CircleAvatar(
  backgroundImage: AssetImage('assets/images/profile.jpg'),
  radius: 30,
),
```

d. ListTile

Definition:
`ListTile` is a widget that represents a single fixed-height row containing text and optional leading and trailing icons.

Usage:

```
dart
Copy code
ListTile(
  leading: Icon(Icons.account_circle),
  title: Text('John Doe'),
  subtitle: Text('Software Engineer'),
  trailing: Icon(Icons.arrow_forward),
  onTap: () {
    // Handle tap
  },
),
```

e. Switch

Definition:
`Switch` is a toggle button that allows users to switch between two states, typically on and off.

Usage:

```
dart
Copy code
Switch(
  value: _isSwitched,
  onChanged: (bool value) {
    setState(() {
      _isSwitched = value;
    });
  },
  activeColor: Colors.green,
),
```

f. SnackBar

Definition:
`SnackBar` is a lightweight message with an optional action, displayed at the bottom of the screen.

Usage:

```
dart
Copy code
ScaffoldMessenger.of(context).showSnackBar(
  SnackBar(
    content: Text('Habit deleted.'),
    duration: Duration(seconds: 2),
  ),
);
```

Key Points:

- **User Feedback:** Provides immediate feedback for user actions.
- **Non-Intrusive:** Doesn't block the UI or require user intervention.

13. Dart Programming Concepts

a. Classes and Constructors

Classes:
Classes are blueprints for creating objects (instances) that encapsulate data and behavior.

Constructors:
Constructors are special methods used to create and initialize objects.

Example:

```
dart
Copy code
class Person {
  String name;
  int age;

  // Constructor
  Person(this.name, this.age);
}
```

Named Constructors:
Allow creating multiple ways to initialize an object.

Example:

```
dart
Copy code
class Person {
  String name;
  int age;

  Person(this.name, this.age);

  // Named constructor for creating a person from a Map
  Person.fromMap(Map<String, dynamic> map)
      : name = map['name'],
        age = map['age'];
}
```

b. Methods

Definition:
Functions defined within a class. They can perform operations, manipulate data, and provide functionality.

Example:

```
dart
Copy code
class Calculator {
  int add(int a, int b) {
    return a + b;
  }
}
```

c. Asynchronous Programming (async/await)

Definition:
Asynchronous programming allows your app to perform tasks that take time (like fetching data) without freezing the UI.

Key Keywords:

- **`async`:**
  Marks a function as asynchronous, allowing the use of `await` inside it.
- **`await`:**
  Pauses the execution of an asynchronous function until the awaited future completes.

Example:

```
dart
Copy code
Future<void> fetchData() async {
  final response = await http.get(Uri.parse('https://example.com/data'));
  if (response.statusCode == 200) {
    // Process data
  } else {
    // Handle error
  }
}
```

Usage in Project:

- **Loading Data:**

  ```
  dart
  Copy code
  Future<void> _loadHabits() async {
    final prefs = await SharedPreferences.getInstance();
    // ...
  }
  ```

- **Saving Data:**

  ```
  dart
  Copy code
  ```

```
Future<void> _saveHabits() async {
  final prefs = await SharedPreferences.getInstance();
  // ...
}
```

d. Factory Constructors

Definition:
A factory constructor is a constructor that can return an instance of the class or even an instance of a subtype.

Usage in Models:

```
dart
Copy code
factory Habit.fromMap(Map<String, dynamic> map) {
  return Habit(
    map['name'],
    isCompleted: map['isCompleted'],
    completionDates: (map['completionDates'] as List<dynamic>)
        .map((dateStr) => DateTime.parse(dateStr))
        .toList(),
  );
}
```

Advantages:

- **Flexibility:** Can decide what instance to return, potentially returning cached instances or different subtypes.
- **Named Constructors:** Enhance readability by providing named constructors for different initialization scenarios.

e. List Manipulation

Adding Items:

```
dart
Copy code
_habits.insert(0, Habit('Exercise'));
```

Removing Items:

```
dart
Copy code
_habits.removeAt(index);
```

Filtering Items:

```
dart
Copy code
```

```
_habits.removeWhere((habit) => habit.isCompleted);
```

Mapping Items:

```
dart
Copy code
final habitNames = _habits.map((habit) => habit.name).toList();
```

Explanation:

- **List Operations:** Dart provides powerful list manipulation methods that make it easy to manage collections of data.

---

14. Conclusion

The Flutter Habit Tracker project is a well-structured application that integrates various Flutter and Dart concepts to deliver a functional and interactive user experience. Here's a recap of the key components and their roles within the project:

- **Flutter Basics:** Understanding widgets, stateful vs. stateless components, and the foundational building blocks of the UI.
- **Project Structure:** Organized directories for models, providers, screens, and widgets ensure maintainability and scalability.
- **State Management with Provider:** Efficiently manages and propagates state changes across the app using the Provider package.
- **Data Persistence with SharedPreferences:** Ensures user data like habits, notes, and journal entries persist across app sessions.
- **Models:** Defines data structures for habits, notes, and journal entries with serialization and deserialization methods.
- **Providers:** Manages different aspects of the app's state, including themes, habits, notes, and journal entries.
- **Custom Widgets:** Reusable components like `CustomAppBar` and `AppDrawer` maintain consistency and enhance the UI.
- **AnimatedList:** Implements dynamic and animated lists that respond to user interactions with smooth transitions.
- **Navigation:** Facilitates smooth transitions between screens using `Navigator` and `Route` concepts.
- **Other Widgets and Concepts:** Utilizes a variety of Flutter widgets like `Scaffold`, `IconButton`, `CircleAvatar`, `ListTile`, `Switch`, and `SnackBar` to build a rich UI.
- **Dart Programming Concepts:** Applies fundamental Dart programming concepts like classes, constructors, methods, asynchronous programming, and factory constructors to build robust app logic.

Next Steps:

1. **Deepen Understanding:**
   Explore each concept further through Flutter's official documentation and tutorials.
2. **Enhance Features:**
   Implement additional functionalities such as data backup, gamification elements, multi-language support, and accessibility features.
3. **Testing:**
   Write unit and widget tests to ensure the app behaves as expected.
4. **UI/UX Improvements:**
   Refine the app's design with advanced animations, responsive layouts, and improved color schemes.
5. **Performance Optimization:**
   Analyze and optimize the app for better performance and lower resource consumption.

Building and understanding this project has provided valuable insights into Flutter development, state management, and data persistence. Continue experimenting and expanding your app to further solidify your understanding and enhance your development skills.
