PROJECT REPORT

MONEY MATTERS: A PERSONAL FINANCE MANAGEMENT APP

# INTRODUCTION

* 1. Project Overview
  2. Purpose

# LITERATURE SURVEY

* 1. Existing problem
  2. References
  3. Problem Statement Definition

# IDEATION & PROPOSED SOLUTION

* 1. Empathy Map Canvas
  2. Ideation & Brainstorming

# REQUIREMENT ANALYSIS

* 1. Functional requirement
  2. Non-Functional requirements

# PROJECT DESIGN

* 1. Data Flow Diagrams & User Stories
  2. Solution Architecture

# PROJECT PLANNING & SCHEDULING

* 1. Technical Architecture
  2. Sprint Planning & Estimation
  3. Sprint Delivery Schedule

# CODING & SOLUTIONING (Explain the features added in the project along with code)

* 1. Feature 1
  2. Feature 2
  3. Database Schema (if Applicable)

# PERFORMANCE TESTING

* 1. Performace Metrics

# RESULTS

* 1. Output Screenshots

# ADVANTAGES & DISADVANTAGES

1. CONCLUSION

# FUTURE SCOPE

1. APPENDIX

Source Code

GitHub & Project Demo Link

# Project Report on Financial Management App

## 1. INTRODUCTION

### 1.1 Project Overview

The Financial Management App is designed to provide users with a comprehensive tool for managing their finances efficiently. This app aims to simplify financial tracking, budgeting, and expense management.

### 1.2 Purpose

The purpose of this project is to create a user-friendly financial management application that enables individuals to monitor their income, expenses, and investments in real-time. The app will help users make informed financial decisions and achieve their financial goals.

## 2. LITERATURE SURVEY

### 2.1 Existing Problems

A review of existing financial management apps highlights the need for a more intuitive and customizable solution. Many apps lack certain features or have complex interfaces, making them less user-friendly.

### 2.2 References

- [Reference 1: Financial App Trends - Journal of Finance and Technology]

- [Reference 2: User-Centric Design in Finance Apps - UX Design Magazine]

### 2.3 Problem Statement Definition

The existing financial apps have limitations in terms of user experience and features. This project addresses these limitations by providing a more user-centric and feature-rich financial management solution.

## 3. IDEATION & PROPOSED SOLUTION

### 3.1 Empathy Map Canvas

An empathy map was created to understand the users' needs, pain points, and aspirations, leading to a more user-centered design.

### 3.2 Ideation & Brainstorming

Through brainstorming sessions, various features were identified to enhance the user experience, such as customizable budget categories, real-time expense tracking, and investment portfolio management.

## 4. REQUIREMENT ANALYSIS

### 4.1 Functional Requirement

The app must support features such as income tracking, expense categorization, budget setting, investment tracking, and financial goal setting.

### 4.2 Non-Functional Requirements

Non-functional requirements include user-friendly interfaces, responsive design, secure data storage, and real-time synchronization across devices.

## 5. PROJECT DESIGN

### 5.1 Data Flow Diagrams & User Stories

Data flow diagrams illustrate the flow of data within the app, while user stories define the interactions users will have with the application.

### 5.2 Solution Architecture

The solution architecture includes the backend server, database, and front-end components, ensuring a scalable and reliable system.

## 6. PROJECT PLANNING & SCHEDULING

### 6.1 Technical Architecture

The technical architecture outlines the technology stack, frameworks, and tools used to develop the financial management app.

### 6.2 Sprint Planning & Estimation

Sprint planning involves breaking down the project into manageable tasks, and estimation provides a timeline for each sprint.

### 6.3 Sprint Delivery Schedule

The sprint delivery schedule outlines when each sprint will be completed and delivered.

## 7. CODING & SOLUTIONING

### 7.1 Feature 1: Real-Time Expense Tracking

This feature allows users to input and track their expenses in real-time, providing an up-to-date overview of their financial status.

### 7.2 Feature 2: Investment Portfolio Management

Users can input and monitor their investments, track portfolio performance, and receive insights for informed decision-making.

### 7.3 Database Schema

The database schema includes tables for user information, income, expenses, budgets, and investments.

## 8. PERFORMANCE TESTING

### 8.1 Performance Metrics

Performance metrics include response time, system scalability, and data retrieval speed.

## 9. RESULTS

### 9.1 Output Screenshots

Provide screenshots of the app, showcasing its user interface and key features.

## 10. ADVANTAGES & DISADVANTAGES

Discuss the advantages, such as user-friendly design and comprehensive features, and potential disadvantages, such as dependency on internet connectivity.

## 11. CONCLUSION

Summarize the project's achievements, addressing the initial problem statement and highlighting the app's contributions to improving financial management.

## 12. FUTURE SCOPE

Discuss potential future enhancements, such as incorporating machine learning for financial predictions or integrating with emerging technologies.

## 13. APPENDIX

- Source Code

APP THEME:

import 'package:flutter/material.dart';

class AppTheme {

static ThemeData lightTheme = ThemeData(

primaryColor: Colors.blue,

// other theme configurations...

);

static ThemeData darkTheme = ThemeData(

primaryColor: Colors.indigo,

// other theme configurations...

);

}

DATABASE HELPER:

import 'dart:async';

import 'dart:io';

import 'package:path/path.dart';

import 'package:path\_provider/path\_provider.dart';

import 'package:sqflite/sqflite.dart';

class DatabaseHelper {

static final \_databaseName = "expense\_tracker.db";

static final \_databaseVersion = 1;

static final table = 'users';

static final columnId = '\_id';

static final columnName = 'name';

static final columnPassword = 'password';

// make this a singleton class

DatabaseHelper.\_privateConstructor();

static final DatabaseHelper instance = DatabaseHelper.\_privateConstructor();

// only have a single app-wide reference to the database

static Database? \_database;

Future<Database?> get database async {

if (\_database != null) return \_database;

\_database = await \_initDatabase();

return \_database;

}

// this opens the database (and creates it if it doesn't exist)

Future \_initDatabase() async {

Directory documentsDirectory = await getApplicationDocumentsDirectory();

String path = join(documentsDirectory.path, \_databaseName);

return await openDatabase(

path,

version: \_databaseVersion,

onCreate: \_onCreate,

onUpgrade: \_onUpgrade,

);

}

Future<void> \_onCreate(Database db, int version) async {

await db.execute('''

CREATE TABLE $table (

$columnId INTEGER PRIMARY KEY,

$columnName TEXT NOT NULL,

$columnPassword TEXT NOT NULL

)

''');

// Add code here to create the 'expenses' table if needed

await db.execute('''

CREATE TABLE expenses (

id INTEGER PRIMARY KEY,

name TEXT NOT NULL,

amount REAL NOT NULL,

date TEXT

)

''');

}

Future<void> \_onUpgrade(Database db, int oldVersion, int newVersion) async {

// Handle database upgrades if needed

}

// Helper methods

// Insert a user into the database

Future<int> insert(Map<String, dynamic> row) async {

final Database? db = await instance.database;

return await db?.insert(table, row) ?? 0;

}

// Query a specific user by username and password

Future<List<Map<String, dynamic>>> queryUser(String name,

String password) async {

Database? db = await instance.database;

return await db?.query(table,

where: '$columnName = ? AND $columnPassword = ?',

whereArgs: [name, password]) ?? [];

}

}

MAIN:

import 'package:flutter/material.dart';

import 'package:fl\_chart/fl\_chart.dart';

import 'package:intl/intl.dart';

import 'package:shared\_preferences/shared\_preferences.dart';

import 'settings\_page.dart';

import 'package:sqflite/sqflite.dart';

import 'package:provider/provider.dart';

import 'package:money\_management/database\_helper.dart';

import'package:money\_management/theme.dart';

void main() async {

WidgetsFlutterBinding.ensureInitialized();

// Initialize the database

await DatabaseHelper.instance.database;

runApp(

ChangeNotifierProvider(

create: (context) => ThemeProvider(),

child: const ExpenseTrackerApp(),

),

);

}

class AppData extends ChangeNotifier {

String \_selectedCurrency = 'USD'; // Add this line

String get selectedCurrency => \_selectedCurrency; // Add this line

void updateSelectedCurrency(String newCurrency) {

\_selectedCurrency = newCurrency;

notifyListeners();

}

}

class ExpenseTrackerApp extends StatefulWidget {

const ExpenseTrackerApp({super.key});

@override

\_ExpenseTrackerAppState createState() => \_ExpenseTrackerAppState();

}

class \_ExpenseTrackerAppState extends State<ExpenseTrackerApp> {

@override

Widget build(BuildContext context) {

final themeProvider = Provider.of<ThemeProvider>(context);

return MaterialApp(

title: 'Expense Tracker',

theme: themeProvider.currentTheme,

home: ChangeNotifierProvider(

create: (context) => AppData(),

child: Builder(

builder: (context) {

final appData = Provider.of<AppData>(context);

return MaterialApp(

initialRoute: '/',

theme: themeProvider.currentTheme,

routes: {

'/': (context) => const WelcomePage(),

'/settings': (context) => const SettingsPage(),

},

);

},

),

),

);

}

}

class WelcomePage extends StatelessWidget {

const WelcomePage({super.key});

@override

Widget build(BuildContext context) {

return Scaffold(

body: Center(

child: Column(

mainAxisAlignment: MainAxisAlignment.center,

children: [

const Text(

'Money Matters',

style: TextStyle(fontSize: 24, fontWeight: FontWeight.bold),

),

const SizedBox(height: 40),

ElevatedButton(

onPressed: () {

Navigator.push(

context,

MaterialPageRoute(builder: (context) => LoginPage()),

);

},

child: const Text('Login'),

),

const SizedBox(height: 20),

ElevatedButton(

onPressed: () {

Navigator.push(

context,

MaterialPageRoute(builder: (context) => RegistrationPage()),

);

},

child: const Text('New User Registration'),

),

],

),

),

);

}

}

class LoginPage extends StatefulWidget {

const LoginPage({Key? key}) : super(key: key);

@override

\_LoginPageState createState() => \_LoginPageState();

}

class \_LoginPageState extends State<LoginPage> {

final TextEditingController \_usernameController = TextEditingController();

final TextEditingController \_passwordController = TextEditingController();

bool \_isDarkMode = false;

bool \_isMounted = false; // Add this line

@override

void initState() {

super.initState();

\_isMounted = true; // Set \_isMounted to true when the widget is mounted

}

@override

void dispose() {

\_isMounted = false; // Set \_isMounted to false when the widget is disposed

super.dispose();

}

void \_toggleTheme() {

if (\_isMounted) {

setState(() {

\_isDarkMode = !\_isDarkMode;

});

}

}

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(

title: const Text('Login'),

),

body: Padding(

padding: const EdgeInsets.all(16.0),

child: Column(

mainAxisAlignment: MainAxisAlignment.center,

children: [

TextField(

controller: \_usernameController,

decoration: const InputDecoration(labelText: 'Username'),

),

TextField(

controller: \_passwordController,

decoration: const InputDecoration(labelText: 'Password'),

obscureText: true,

),

const SizedBox(height: 20),

ElevatedButton(

onPressed: () {

\_login(context);

},

child: const Text('Login'),

),

],

),

),

);

}

Future<void> \_login(BuildContext context) async {

final dbHelper = DatabaseHelper.instance;

final List<Map<String, dynamic>> user = await dbHelper.queryUser(

\_usernameController.text, \_passwordController.text);

// Now that the 'user' has been retrieved, you can work with it

if (user.isNotEmpty) {

// Successful login

Navigator.pushReplacement(

context,

MaterialPageRoute(

builder: (context) => ExpenseListScreen(

),

),

);

print('Login successful');

} else {

// Invalid credentials

print('Invalid credentials');

}

}

// Replace with your actual logic

}

class RegistrationPage extends StatelessWidget {

final TextEditingController \_newUsernameController = TextEditingController();

final TextEditingController \_newPasswordController = TextEditingController();

RegistrationPage({super.key});

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(

title: const Text('New User Registration'),

),

body: Padding(

padding: const EdgeInsets.all(16.0),

child: Column(

mainAxisAlignment: MainAxisAlignment.center,

children: [

TextField(

controller: \_newUsernameController,

decoration: const InputDecoration(labelText: 'New Username'),

),

TextField(

controller: \_newPasswordController,

decoration: const InputDecoration(labelText: 'New Password'),

obscureText: true,

),

const SizedBox(height: 20),

ElevatedButton(

onPressed: () {

\_register(context);

},

child: const Text('Register'),

),

],

),

),

);

}

Future<void> \_register(BuildContext context) async {

final dbHelper = DatabaseHelper.instance;

final newUsername = \_newUsernameController.text;

final newPassword = \_newPasswordController.text;

if (newUsername.isNotEmpty && newPassword.isNotEmpty) {

// Store new user registration data

final id = await dbHelper.insert({

DatabaseHelper.columnName: newUsername,

DatabaseHelper.columnPassword: newPassword,

});

if (id != null) {

// Navigate back to the welcome page after registration

Navigator.pop(context);

print('Registration successful');

} else {

print('Failed to register user');

}

} else {

// Handle invalid input

print('Invalid registration data');

}

}

}

class Expense {

final String name;

final double amount;

final DateTime? date;

Expense({required this.name, required this.amount, this.date});

}

class MonthlyExpenseChartScreen extends StatelessWidget {

final List<Expense> expenses;

const MonthlyExpenseChartScreen({super.key, required this.expenses});

@override

Widget build(BuildContext context) {

final List<BarChartGroupData> barGroups = generateBarGroups();

final double maxExpense = getMaxMonthlyExpense();

final bool hasExpenses = barGroups.isNotEmpty;

return Scaffold(

appBar: AppBar(

title: const Text('Monthly Expense Chart'),

),

body: Padding(

padding: const EdgeInsets.all(16.0),

child: hasExpenses

? BarChart(

BarChartData(

alignment: BarChartAlignment.spaceAround,

maxY: maxExpense \* 1.2,

titlesData: const FlTitlesData(

leftTitles:AxisTitles(sideTitles: SideTitles(reservedSize: 40, showTitles: true)),

bottomTitles:AxisTitles(sideTitles: SideTitles(reservedSize: 6, showTitles: true))

,

),

barGroups: barGroups,

),

)

: const Center(

child: Text(

'No data available',

style: TextStyle(fontSize: 18),

),

),

),

);

}

double getMaxMonthlyExpense() {

double maxExpense = 0.0;

for (var expense in expenses) {

if (expense.date != null) {

final month = DateTime(expense.date!.year, expense.date!.month);

final monthlyTotal = getMonthlyExpense(month);

if (monthlyTotal > maxExpense) {

maxExpense = monthlyTotal;

}

}

}

return maxExpense;

}

List<BarChartGroupData> generateBarGroups() {

final Map<String, double> monthlyExpenses = {};

for (var expense in expenses) {

if (expense.date != null) {

final month = DateTime(expense.date!.year, expense.date!.month);

final monthKey = '${month.year}-${month.month}';

monthlyExpenses.update(

monthKey,

(value) => value + expense.amount,

ifAbsent: () => expense.amount,

);

}

}

double maxExpense = getMaxMonthlyExpense();

final List<BarChartGroupData> barGroups = [];

monthlyExpenses.forEach((month, totalExpense) {

final isMaxMonth = totalExpense == maxExpense;

barGroups.add(

BarChartGroupData(

x: barGroups.length,

barsSpace: 12,

barRods: [

BarChartRodData(

toY: totalExpense,

color: isMaxMonth ? Colors.red : Colors.blue,

width: 16,

),

],

),

);

});

return barGroups;

}

double getMonthlyExpense(DateTime month) {

double monthlyTotal = 0.0;

for (var expense in expenses) {

if (expense.date != null) {

final expenseMonth = DateTime(expense.date!.year, expense.date!.month);

if (expenseMonth == month) {

monthlyTotal += expense.amount;

}

}

}

return monthlyTotal;

}

List<String> getUniqueMonths(List<Expense> expenses) {

Set<String> uniqueMonths = <String>{};

for (var expense in expenses) {

if (expense.date != null) {

final month = DateTime(expense.date!.year, expense.date!.month);

uniqueMonths.add('${month.year}-${month.month}');

}

}

return List<String>.from(uniqueMonths);

}

}

class ExpenseListScreen extends StatefulWidget {

const ExpenseListScreen({super.key, });

@override

\_ExpenseListScreenState createState() => \_ExpenseListScreenState();

}

class \_ExpenseListScreenState extends State<ExpenseListScreen> {

final List<Expense> expenses = [];

double budget = 0;

DateTime? selectedDate;

@override

void initState() {

super.initState();

\_loadExpenses();

\_loadBudgetAsync(); // Corrected method name

}

Future<void> \_loadBudget() async {

final prefs = await SharedPreferences.getInstance();

final storedBudget = prefs.getDouble('budget');

if (storedBudget != null) {

setState(() {

budget = storedBudget;

});

}

}

Future<void> \_loadExpenses() async {

final dbHelper = DatabaseHelper.instance;

final Database? db = await dbHelper.database;

if (db != null) {

final List<Map<String, dynamic>> queryResult = await db.query('expenses');

setState(() {

expenses.addAll(queryResult.map((e) => Expense(

name: e['name'],

amount: e['amount'],

date: e['date'] != null ? DateTime.parse(e['date']) : null,

)).toList());

});

} else {

// Handle the case where the database is null

print('Error: Database is null');

}

}

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(

title: const Text('Expense Tracker'),

actions: [

],

),

body: Column(

children: <Widget>[

Padding(

padding: const EdgeInsets.all(16.0),

child: Text(

'Budget: ${budget.toStringAsFixed(2)} ${Provider.of<AppData>(context).selectedCurrency}',

style: const TextStyle(fontSize: 18),

),

),

ElevatedButton(

onPressed: () {

\_setBudget(context);

},

style: ElevatedButton.styleFrom(

),

child: Text('Set Budget', style: TextStyle(color: Colors.white )),

),

ElevatedButton(

onPressed: () {

\_sortExpensesByMonth();

},

style: ElevatedButton.styleFrom(

),

child: Text('Sort by Month', style: TextStyle(color:Colors.white )),

),

ElevatedButton(

onPressed: () {

Navigator.pushNamed(context, '/settings');

},

child: const Text('Settings'),

),

ListView.builder(

shrinkWrap: true,

itemCount: expenses.length,

itemBuilder: (context, index) {

final expense = expenses[index];

final monthHeading = \_getMonthHeading(expense.date);

return Column(

crossAxisAlignment: CrossAxisAlignment.start,

children: <Widget>[

if (index == 0 || monthHeading != \_getMonthHeading(expenses[index - 1].date))

Padding(

padding: const EdgeInsets.symmetric(vertical: 8.0),

child: Text(monthHeading),

),

Card(

margin: const EdgeInsets.all(8.0),

child: ListTile(

title: Text(

expense.name,

style: TextStyle(

fontWeight: FontWeight.bold,

),

),

subtitle: Column(

crossAxisAlignment: CrossAxisAlignment.start,

children: <Widget>[

Text(

'\$${expense.amount.toStringAsFixed(2)}',

),

if (expense.date != null)

Text(

'Date: ${expense.date!.toLocal().toString().split(' ')[0] ?? "N/A"}',

),

],

),

trailing: IconButton(

icon: const Icon(Icons.delete),

color: Colors.red,

onPressed: () {

\_deleteExpense(expenses[index]);

},

),

),

)]);

},

),

],

),

floatingActionButton: FloatingActionButton(

onPressed: () {

\_addExpense(context);

},

child: const Icon(Icons.add),

),

bottomNavigationBar: ExpenseGraphNavBar(

onChartPressed: () {

Navigator.push(

context,

MaterialPageRoute(

builder: (context) => MonthlyExpenseChartScreen(expenses: expenses),

),

);

},

),

);

}

Future<void> \_loadBudgetAsync() async {

final prefs = await SharedPreferences.getInstance();

final storedBudget = prefs.getDouble('budget');

if (storedBudget != null) {

setState(() {

budget = storedBudget;

});

}

}

void \_sortExpensesByMonth() {

setState(() {

expenses.sort((a, b) {

if (a.date == null || b.date == null) {

return 0;

}

return a.date!.compareTo(b.date!);

});

});

}

String \_getMonthHeading(DateTime? date) {

if (date == null) {

return 'Unknown Month';

}

return DateFormat.yMMMM().format(date);

}

void \_deleteExpense(Expense expense) {

setState(() {

expenses.remove(expense);

});

}

// Replace your existing code for \_addExpense

// Replace your existing code for \_addExpense

Future<void> \_addExpense(BuildContext context) async {

TextEditingController nameController = TextEditingController();

TextEditingController amountController = TextEditingController();

selectedDate = DateTime.now();

Future<void> \_selectDate() async {

final DateTime? picked = await showDatePicker(

context: context,

initialDate: selectedDate!,

firstDate: DateTime(2000),

lastDate: DateTime(2101),

);

if (picked != null) {

setState(() {

selectedDate = picked;

});

}

}

return showDialog(

context: context,

builder: (context) {

return AlertDialog(

title: const Text('Add Expense'),

content: Column(

mainAxisSize: MainAxisSize.min,

children: [

TextField(

controller: nameController,

decoration: const InputDecoration(labelText: 'Expense Name'),

),

TextField(

controller: amountController,

decoration: const InputDecoration(labelText: 'Amount budget'),

keyboardType: TextInputType.number,

),

Row(

children: [

const Text('Date: '),

Text(selectedDate!.toLocal().toString().split(' ')[0] ?? "N/A"),

IconButton(

icon: const Icon(Icons.calendar\_today),

onPressed: \_selectDate,

),

],

),

],

),

actions: [

TextButton(

onPressed: () {

Navigator.of(context).pop();

},

child: const Text('Cancel'),

),

TextButton(

onPressed: () async {

final name = nameController.text;

final amount = double.tryParse(amountController.text) ?? 0.0;

if (name.isNotEmpty && amount > 0) {

if (getTotalExpenses() + amount <= budget) {

// Insert the expense into the database

final dbHelper = DatabaseHelper.instance;

final db = await dbHelper.database;

try {

final id = await db?.insert(

'expenses',

{

'name': name,

'amount': amount,

'date': selectedDate?.toIso8601String(),

},

);

if (id != null) {

setState(() {

expenses.add(Expense(name: name, amount: amount, date: selectedDate));

});

Navigator.of(context).pop();

print('Expense added successfully');

} else {

print('Failed to add expense. Insert returned null.');

}

} catch (e) {

print('Error during database insertion: $e');

}

} else {

showDialog(

context: context,

builder: (context) {

return AlertDialog(

title: const Text('Exceeded Budget'),

content: const Text('Your expenses exceed the budget.'),

actions: [

TextButton(

onPressed: () {

Navigator.of(context).pop();

},

child: const Text('OK'),

),

],

);

},

);

}

} else {

// Handle invalid input

}

},

child: const Text('Add'),

),

],

);

},

);

}

double getTotalExpenses() {

return expenses.isNotEmpty ? expenses.map((expense) => expense.amount).reduce((a, b) => a + b) : 0.0;

}

Future<void> \_setBudget(BuildContext context) async {

TextEditingController budgetController = TextEditingController();

budgetController.text = budget.toStringAsFixed(2);

return showDialog(

context: context,

builder: (context) {

return AlertDialog(

title: const Text('Set Budget'),

content: Column(

mainAxisSize: MainAxisSize.min,

children: [

TextField(

controller: budgetController,

decoration: const InputDecoration(labelText: 'Budget ()'),

keyboardType: TextInputType.number,

),

],

),

actions: [

TextButton(

onPressed: () {

Navigator.of(context).pop();

},

child: const Text('Cancel'),

),

TextButton(

onPressed: () async { // Add async keyword here

final newBudget = double.tryParse(budgetController.text) ?? 0.0;

if (newBudget >= 0) {

setState(() {

budget = newBudget;

});

// Save the budget to SharedPreferences

final prefs = await SharedPreferences.getInstance();

prefs.setDouble('budget', newBudget);

Navigator.of(context).pop();

} else {

// Handle invalid input

}

},

child: const Text('Set'),

),

],

);

},

);

}

}

class ExpenseGraphNavBar extends StatelessWidget {

final VoidCallback onChartPressed;

const ExpenseGraphNavBar({super.key, required this.onChartPressed});

@override

Widget build(BuildContext context) {

return BottomNavigationBar(

items: const <BottomNavigationBarItem>[

BottomNavigationBarItem(

icon: Icon(Icons.list),

label: 'List',

),

BottomNavigationBarItem(

icon: Icon(Icons.show\_chart),

label: 'Chart',

),

],

currentIndex: 0,

onTap: (index) {

if (index == 1) {

onChartPressed();

}

},

);

}

}

SETTINGS PAGE:

import 'package:flutter/material.dart';

import 'package:provider/provider.dart';

class AppData with ChangeNotifier {

String \_selectedCurrency = 'USD'; // default currency

List<String> currencies = ['USD', 'EUR', 'GBP', 'INR']; // add your currencies

String get selectedCurrency => \_selectedCurrency;

set selectedCurrency(String newCurrency) {

\_selectedCurrency = newCurrency;

notifyListeners(); // This notifies all listeners that the value has changed

}

}

class SettingsPage extends StatelessWidget {

const SettingsPage({super.key});

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(

title: const Text('Settings'),

),

body: Padding(

padding: const EdgeInsets.all(16.0),

child: Column(

crossAxisAlignment: CrossAxisAlignment.start,

children: [

const Text(

'Select Currency:',

style: TextStyle(fontSize: 18, fontWeight: FontWeight.bold),

),

const SizedBox(height: 10),

\_CurrencyDropdown(),

],

),

),

);

}

}

class \_CurrencyDropdown extends StatelessWidget {

@override

Widget build(BuildContext context) {

final appData = Provider.of<AppData>(context, listen: false);

return DropdownButton<String>(

value: appData.selectedCurrency,

onChanged: (newValue) {

appData.selectedCurrency = newValue!;

},

items: appData.currencies.map<DropdownMenuItem<String>>((String value) {

return DropdownMenuItem<String>(

value: value,

child: Text(value),

);

}).toList(),

);

}

}

THEME:

import 'package:flutter/material.dart';

import 'package:provider/provider.dart';

import 'package:money\_management/app\_theme.dart';

class ThemeProvider extends ChangeNotifier {

ThemeData \_currentTheme = AppTheme.lightTheme; // Set the initial theme

ThemeData get currentTheme => \_currentTheme;

void toggleTheme() {

\_currentTheme = (\_currentTheme == AppTheme.lightTheme)

? AppTheme.darkTheme

: AppTheme.lightTheme;

notifyListeners();

}

}

class YourWidget extends StatelessWidget {

@override

Widget build(BuildContext context) {

final themeProvider = Provider.of<ThemeProvider>(context);

return Scaffold(

appBar: AppBar(

title: const Text('Your Widget'),

),

body: Center(

child: Column(

mainAxisAlignment: MainAxisAlignment.center,

children: [

ElevatedButton(

onPressed: () {

themeProvider.toggleTheme(); // Toggle the theme

},

child: const Text('Toggle Theme'),

),

const SizedBox(height: 16),

Text(

'Current Theme: ${themeProvider.currentTheme == AppTheme.lightTheme ? 'Light' : 'Dark'}',

style: TextStyle(fontSize: 18),

),

],

),

),

);

}

}