Members: 07/11/25

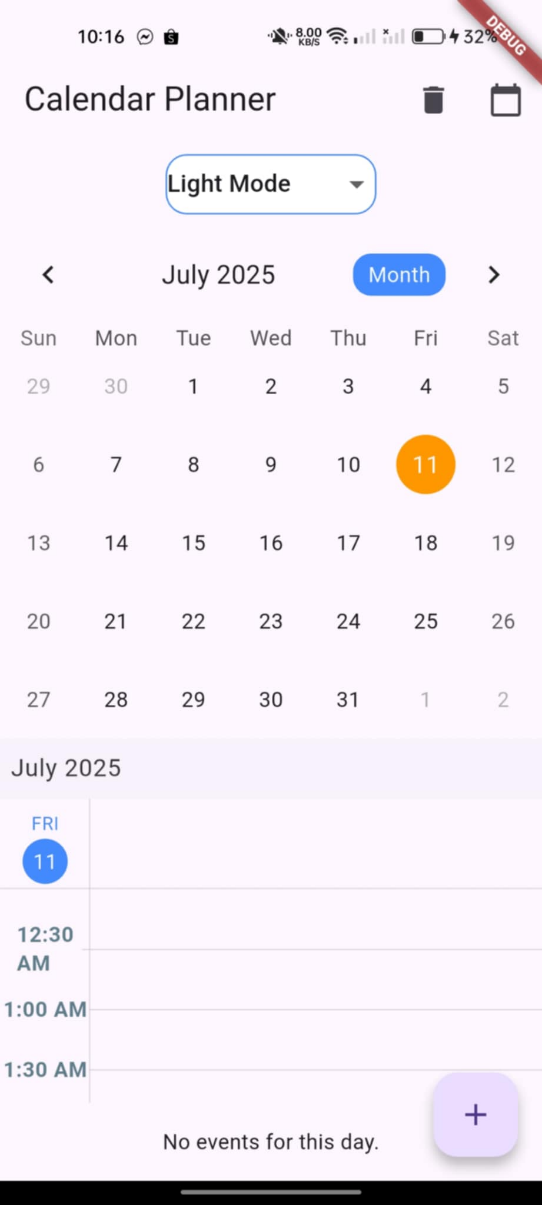
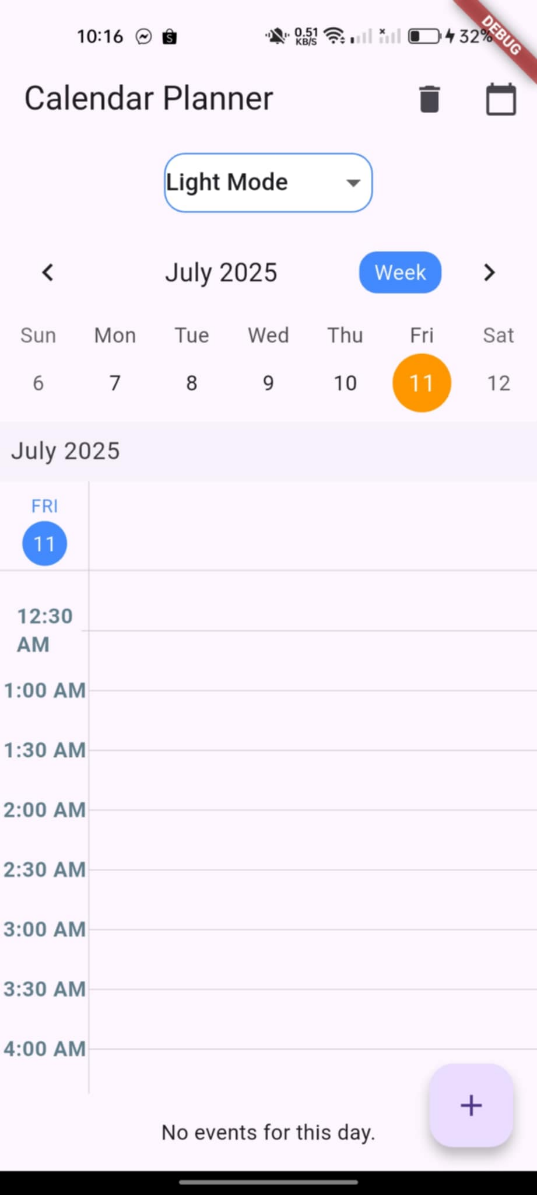
Alo, Emmanuel

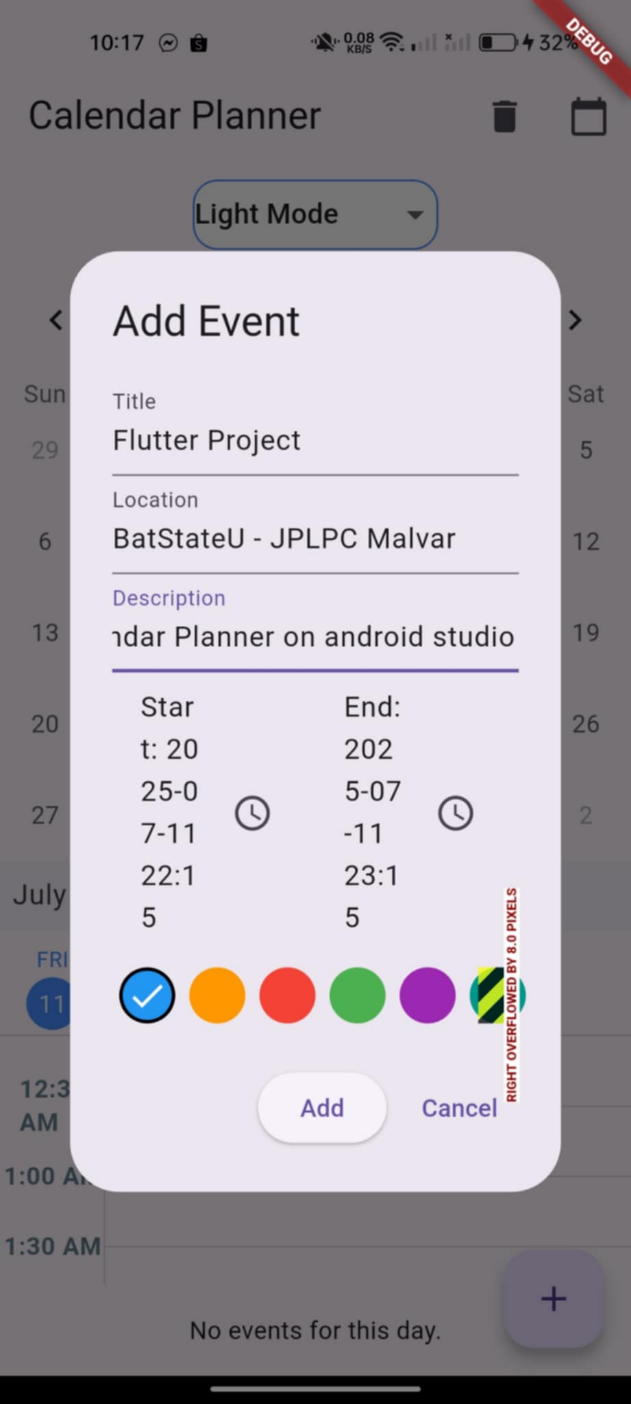
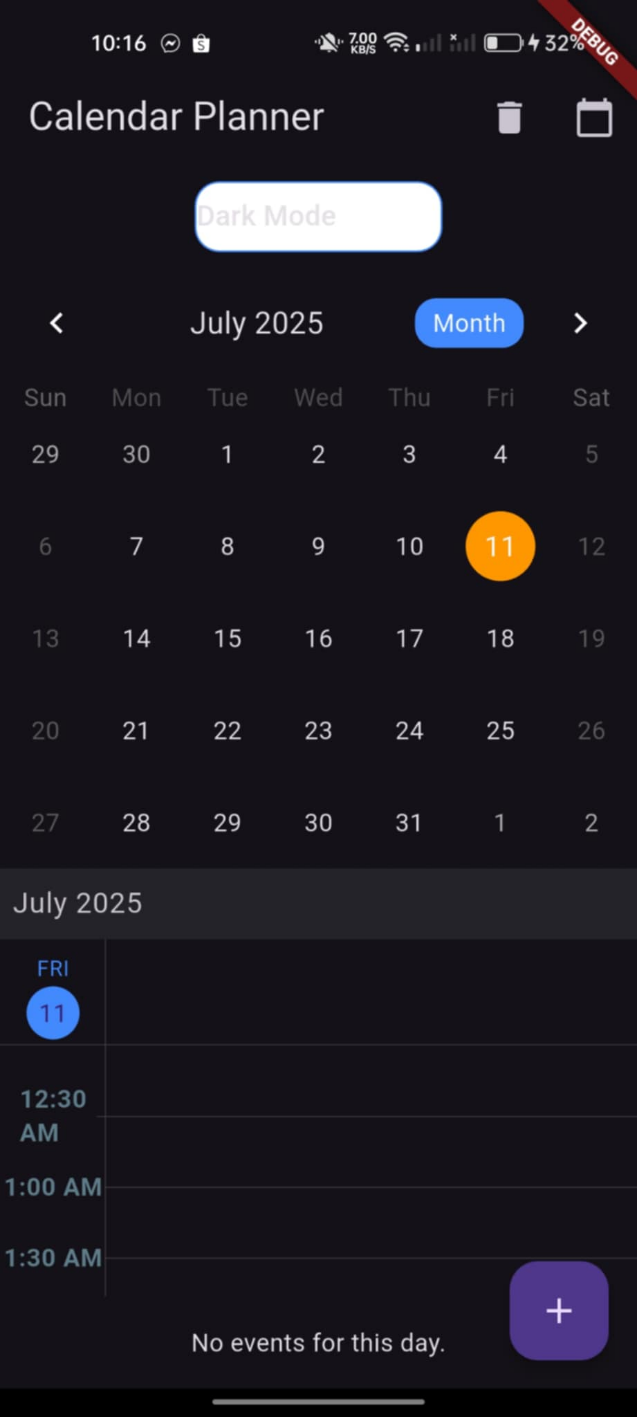
Doton, John Ansel

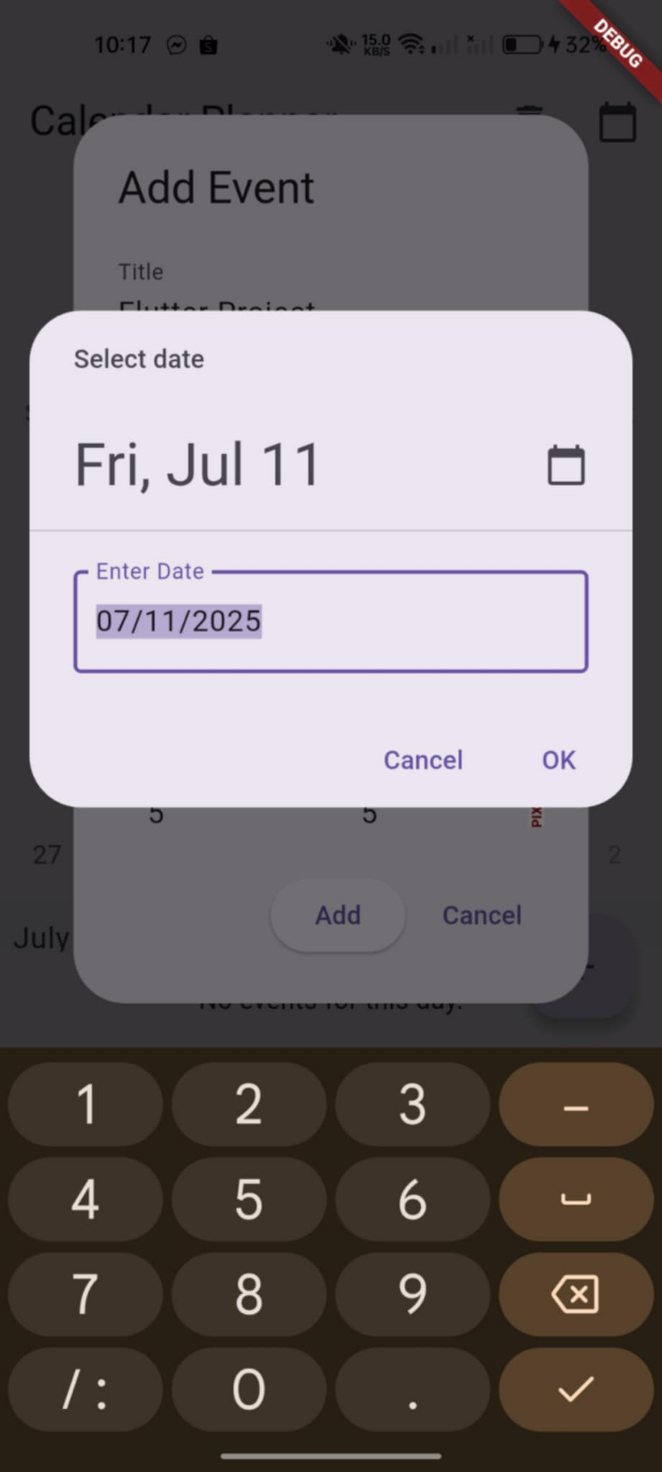
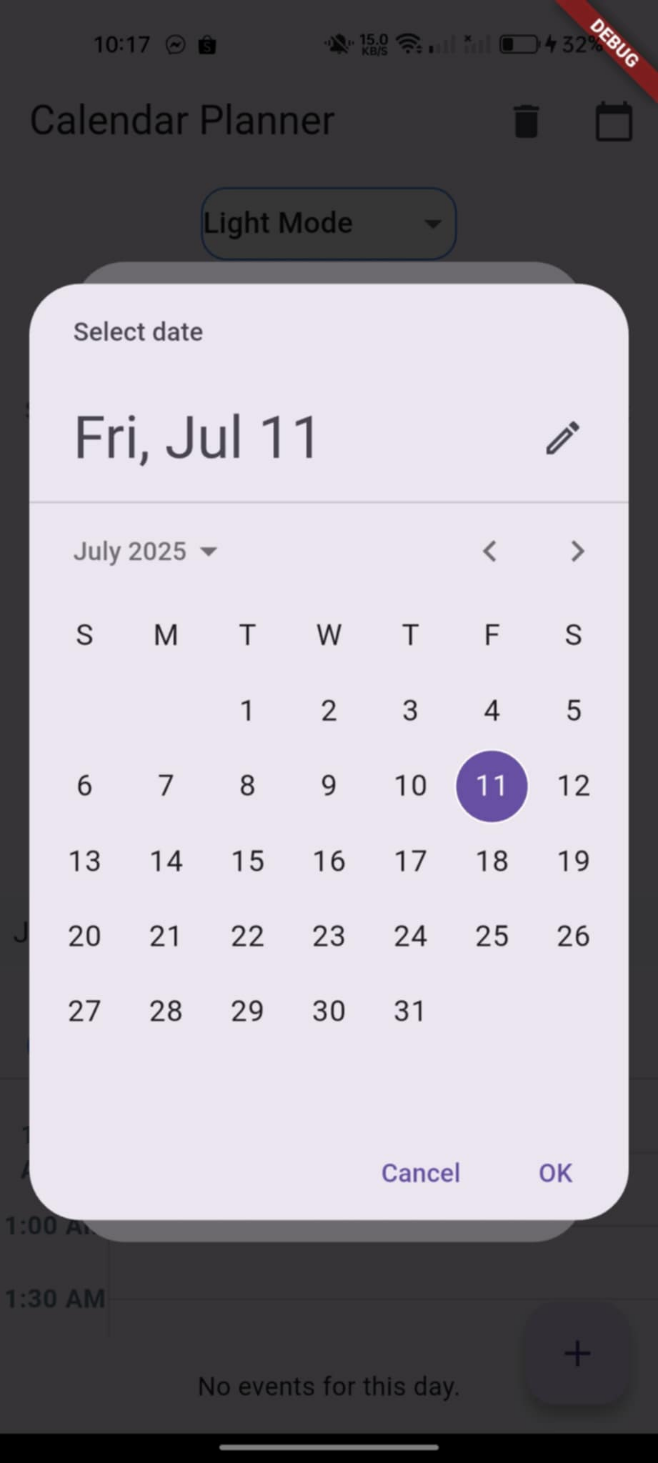
Tanchanco, Jao Algrin

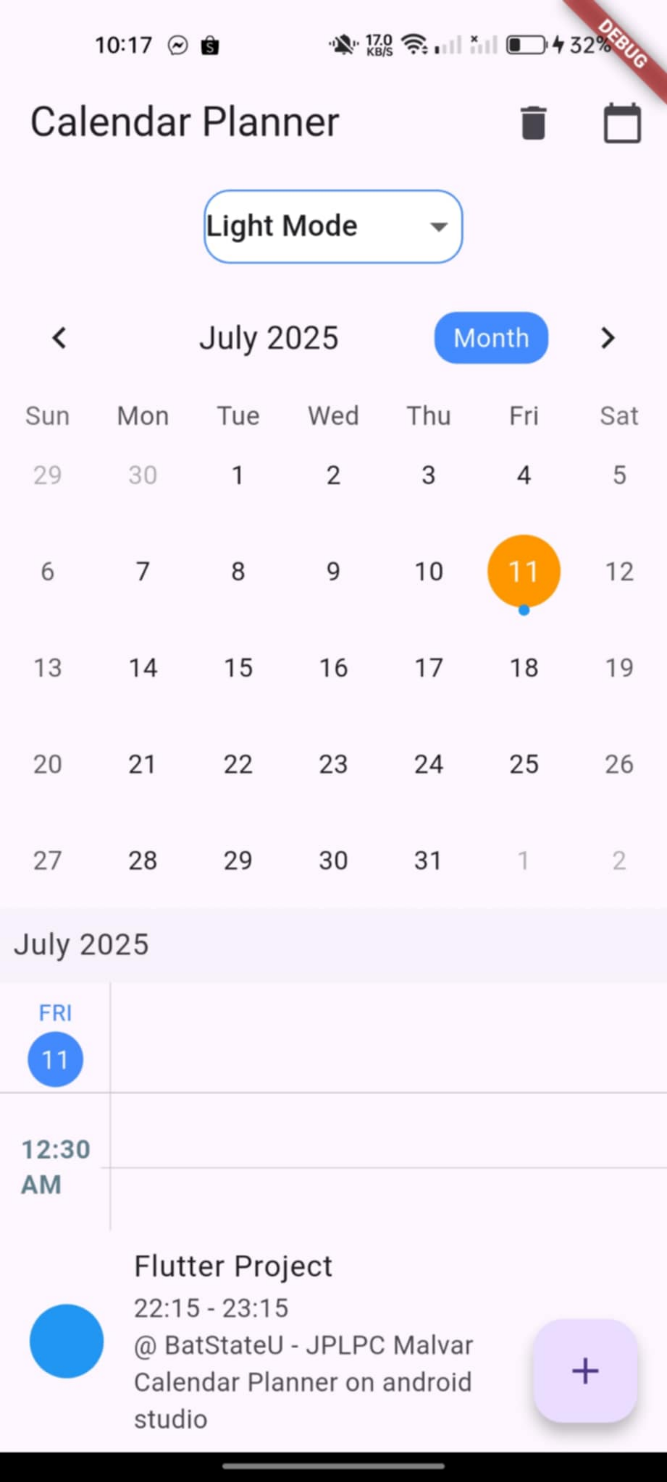
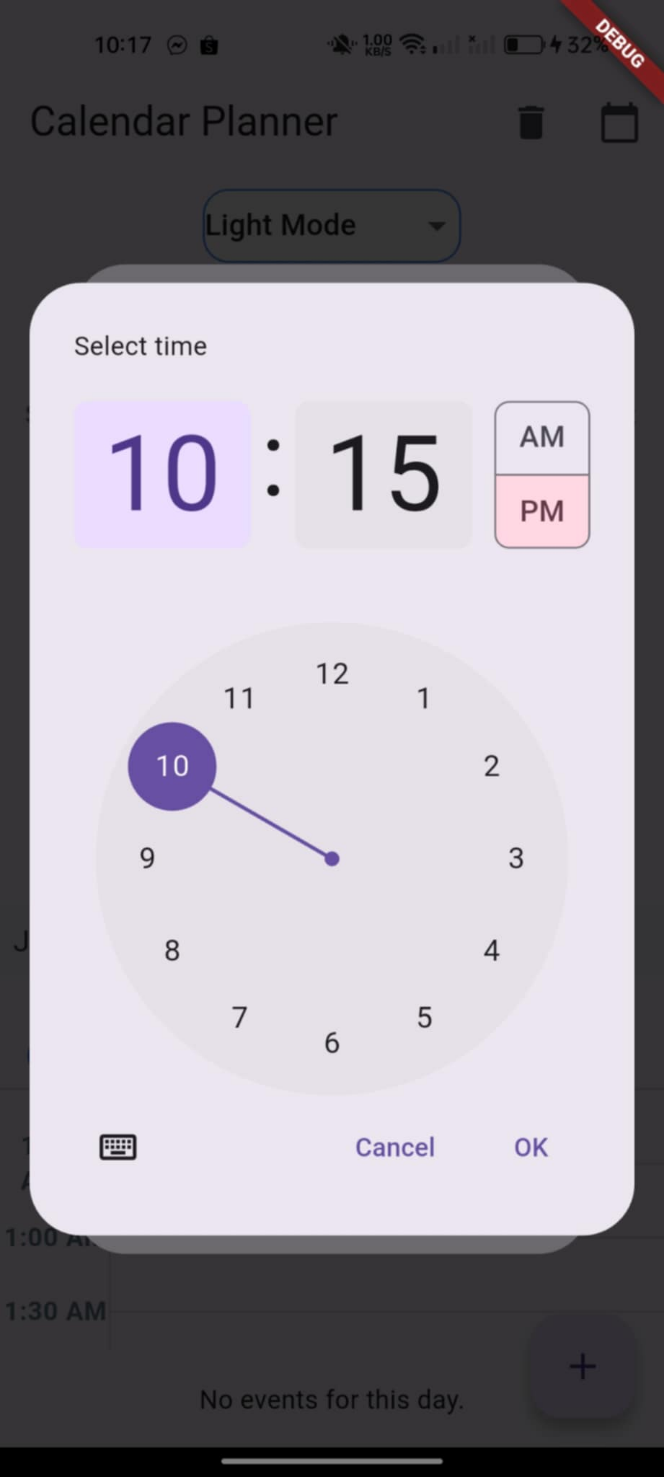
Section: BSIT NT-3301

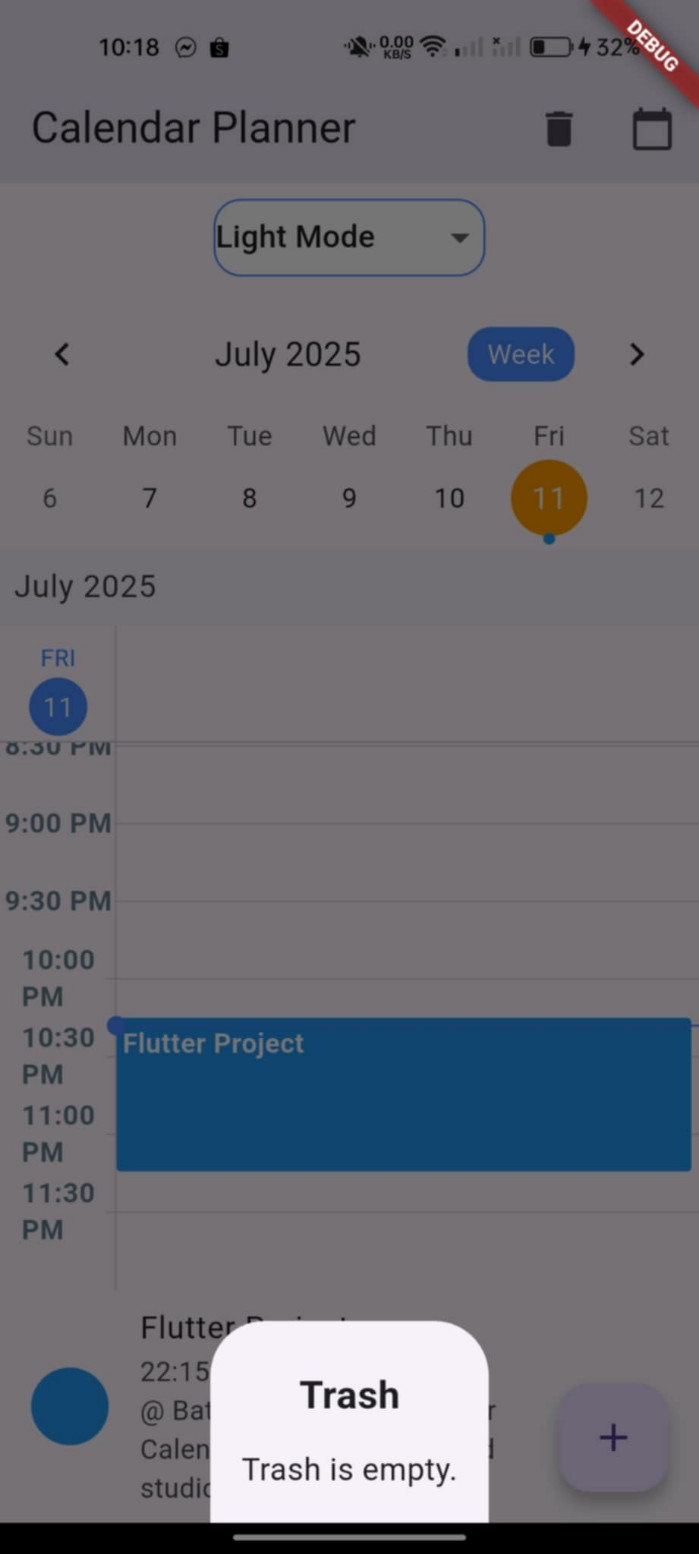
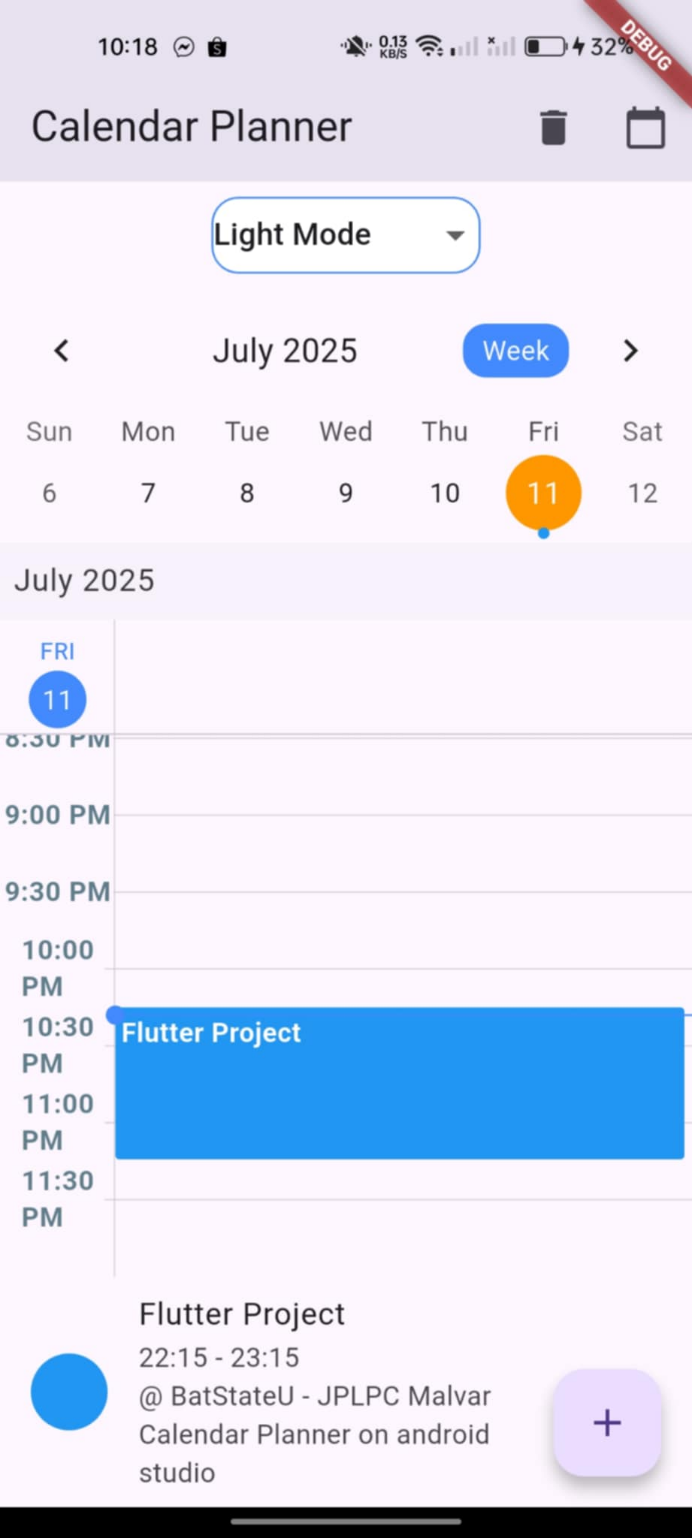
Flutter Progress – Calendar Planner

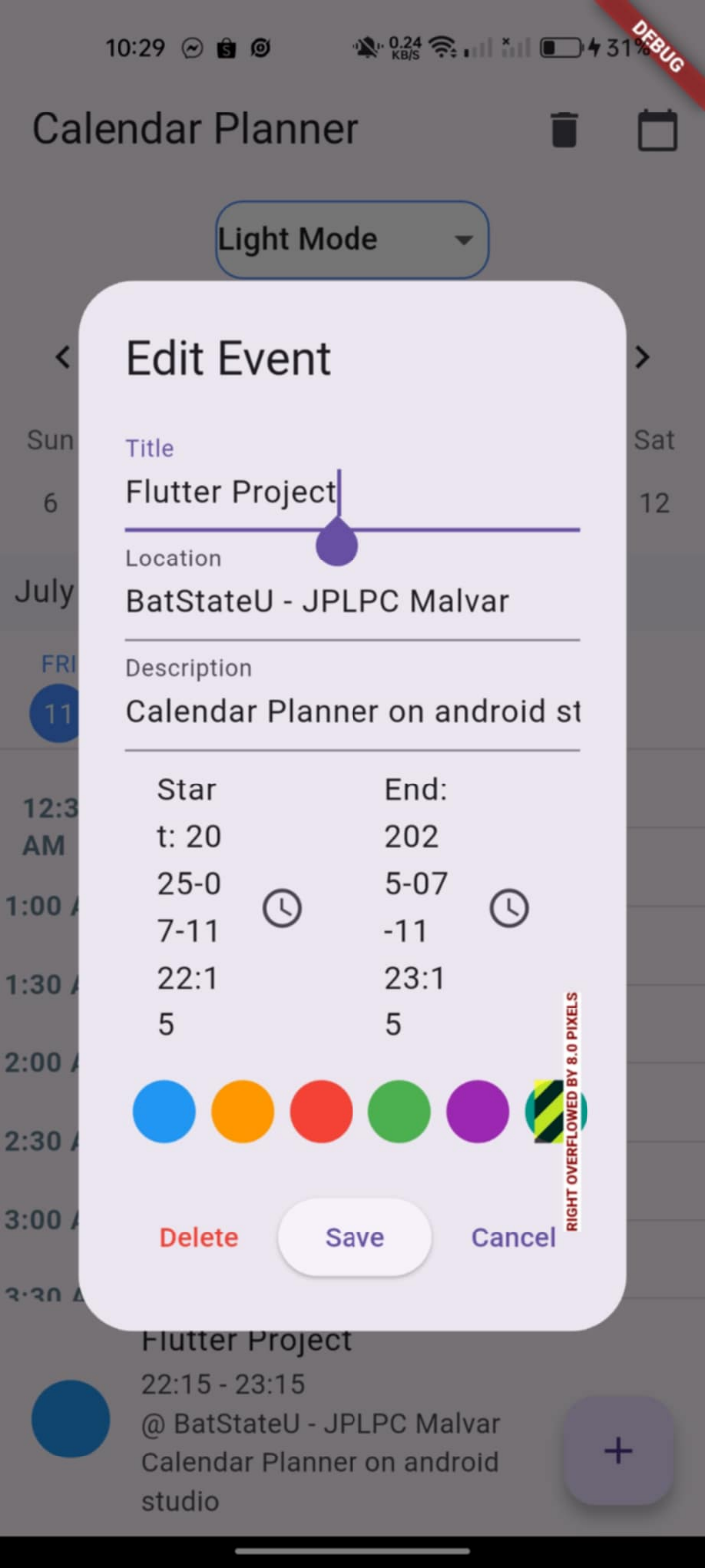












**Code:**

**Main.dart**

import 'package:flutter/material.dart';  
import 'package:table\_calendar/table\_calendar.dart';  
import 'package:syncfusion\_flutter\_calendar/calendar.dart';  
import 'package:dropdown\_button2/dropdown\_button2.dart';  
import 'database\_helper.dart'; // Make sure this import is correct  
  
void main() {  
 runApp(CalendarPlannerApp());  
}  
  
class CalendarPlannerApp extends StatefulWidget {  
 @override  
 \_CalendarPlannerAppState createState() => \_CalendarPlannerAppState();  
}  
  
class \_CalendarPlannerAppState extends State<CalendarPlannerApp> {  
 ThemeMode \_themeMode = ThemeMode.light;  
  
 void \_setThemeMode(ThemeMode mode) {  
 setState(() {  
 \_themeMode = mode;  
 });  
 }  
  
 @override  
 Widget build(BuildContext context) {  
 return MaterialApp(  
 title: 'Calendar Planner',  
 theme: ThemeData(primarySwatch: Colors.*blue*, brightness: Brightness.light),  
 darkTheme: ThemeData(brightness: Brightness.dark, primarySwatch: Colors.*blue*),  
 themeMode: \_themeMode,  
 home: CalendarHomePage(  
 themeMode: \_themeMode,  
 onThemeModeChanged: \_setThemeMode,  
 ),  
 );  
 }  
}  
  
class MyEvent {  
 int? id;  
 String title;  
 DateTime start;  
 DateTime end;  
 Color color;  
 String location;  
 String description;  
 bool isDeleted;  
  
 MyEvent({  
 this.id,  
 required this.title,  
 required this.start,  
 required this.end,  
 required this.color,  
 required this.location,  
 required this.description,  
 this.isDeleted = false,  
 });  
  
 MyEvent copyWith({  
 int? id,  
 String? title,  
 DateTime? start,  
 DateTime? end,  
 Color? color,  
 String? location,  
 String? description,  
 bool? isDeleted,  
 }) {  
 return MyEvent(  
 id: id ?? this.id,  
 title: title ?? this.title,  
 start: start ?? this.start,  
 end: end ?? this.end,  
 color: color ?? this.color,  
 location: location ?? this.location,  
 description: description ?? this.description,  
 isDeleted: isDeleted ?? this.isDeleted,  
 );  
 }  
}  
  
class EventDataSource extends CalendarDataSource {  
 EventDataSource(List<Appointment> source) {  
 appointments = source;  
 }  
}  
  
class CalendarHomePage extends StatefulWidget {  
 final ThemeMode themeMode;  
 final void Function(ThemeMode) onThemeModeChanged;  
  
 CalendarHomePage({required this.themeMode, required this.onThemeModeChanged});  
  
 @override  
 \_CalendarHomePageState createState() => \_CalendarHomePageState();  
}  
  
class \_CalendarHomePageState extends State<CalendarHomePage> {  
 final DatabaseHelper dbHelper = DatabaseHelper();  
 List<MyEvent> \_myEvents = [];  
  
 CalendarFormat \_calendarFormat = CalendarFormat.week;  
 DateTime \_focusedDay = DateTime.now();  
 DateTime? \_selectedDay;  
  
 List<MyEvent> get \_activeEvents => \_myEvents.where((e) => !e.isDeleted).toList();  
  
 @override  
 void initState() {  
 super.initState();  
 \_selectedDay = \_focusedDay;  
 \_loadEvents();  
 }  
  
 Future<void> \_loadEvents() async {  
 // Load ALL events, not just active, so trash works  
 final events = await dbHelper.getAllEvents();  
 setState(() {  
 \_myEvents = events;  
 });  
 }  
  
 List<MyEvent> \_getEventsForDay(DateTime day) {  
 return \_activeEvents  
 .where((event) =>  
 event.start.year == day.year &&  
 event.start.month == day.month &&  
 event.start.day == day.day)  
 .toList();  
 }  
  
 List<Appointment> \_getDataSource() {  
 return \_activeEvents.map((event) {  
 return Appointment(  
 startTime: event.start,  
 endTime: event.end,  
 subject: event.title,  
 color: event.color,  
 notes: event.description,  
 location: event.location,  
 );  
 }).toList();  
 }  
  
 void \_addOrEditEvent({MyEvent? event, DateTime? initialDate}) async {  
 final isEditing = event != null;  
 final result = await showDialog<MyEvent>(  
 context: context,  
 builder: (context) => EventDialog(  
 event: event,  
 initialDate: initialDate ?? \_selectedDay ?? \_focusedDay,  
 ),  
 );  
 if (result != null) {  
 if (isEditing) {  
 if (result.isDeleted) {  
 // Update the event with all new values and set is\_deleted = 1  
 await dbHelper.updateEvent(result.copyWith(id: event!.id, isDeleted: true));  
 } else {  
 await dbHelper.updateEvent(result.copyWith(id: event!.id));  
 }  
 } else {  
 await dbHelper.insertEvent(result);  
 }  
 \_loadEvents();  
 }  
 }  
  
 void \_softDeleteEvent(MyEvent event) async {  
 if (event.id != null) {  
 await dbHelper.softDeleteEvent(event.id!);  
 \_loadEvents();  
 }  
 }  
  
 void \_restoreEvent(MyEvent event) async {  
 if (event.id != null) {  
 await dbHelper.restoreEvent(event.id!);  
 \_loadEvents();  
 }  
 }  
  
 void \_hardDeleteEvent(MyEvent event) async {  
 if (event.id != null) {  
 await dbHelper.hardDeleteEvent(event.id!);  
 \_loadEvents();  
 }  
 }  
  
 void \_openTrash() {  
 showModalBottomSheet(  
 context: context,  
 isScrollControlled: true,  
 builder: (context) {  
 final deletedEvents = \_myEvents.where((e) => e.isDeleted).toList();  
 return Padding(  
 padding: EdgeInsets.only(  
 bottom: MediaQuery.*of*(context).viewInsets.bottom,  
 left: 16,  
 right: 16,  
 top: 24,  
 ),  
 child: Column(  
 mainAxisSize: MainAxisSize.min,  
 children: [  
 Text('Trash',  
 style: TextStyle(fontSize: 20, fontWeight: FontWeight.*bold*)),  
 SizedBox(height: 12),  
 if (deletedEvents.isEmpty)  
 Text('Trash is empty.', style: TextStyle(fontSize: 16)),  
 ...deletedEvents.map((event) => ListTile(  
 leading: CircleAvatar(backgroundColor: event.color),  
 title: Text(event.title),  
 subtitle: Text(  
 '${\_formatDateTime(event.start)} - ${\_formatDateTime(event.end)}'  
 '${event.location.isNotEmpty ? '\n@ ${event.location}' : ''}'  
 '${event.description.isNotEmpty ? '\n${event.description}' : ''}',  
 ),  
 trailing: Row(  
 mainAxisSize: MainAxisSize.min,  
 children: [  
 IconButton(  
 icon: Icon(Icons.*restore*, color: Colors.*green*),  
 tooltip: 'Restore',  
 onPressed: () {  
 \_restoreEvent(event);  
 Navigator.*pop*(context);  
 },  
 ),  
 IconButton(  
 icon: Icon(Icons.*delete\_forever*, color: Colors.*red*),  
 tooltip: 'Delete Forever',  
 onPressed: () {  
 \_hardDeleteEvent(event);  
 Navigator.*pop*(context);  
 },  
 ),  
 ],  
 ),  
 )),  
 SizedBox(height: 16),  
 ],  
 ),  
 );  
 },  
 );  
 }  
  
 String \_formatDateTime(DateTime dt) {  
 return '${dt.year}-${dt.month.toString().padLeft(2, '0')}-${dt.day.toString().padLeft(2, '0')} '  
 '${dt.hour.toString().padLeft(2, '0')}:${dt.minute.toString().padLeft(2, '0')}';  
 }  
  
 Widget \_buildEventList() {  
 final selectedDate = \_selectedDay ?? \_focusedDay;  
 final eventsForDay = \_getEventsForDay(selectedDate);  
  
 if (eventsForDay.isEmpty) {  
 return Padding(  
 padding: const EdgeInsets.all(16.0),  
 child: Text('No events for this day.'),  
 );  
 }  
  
 return ListView.builder(  
 shrinkWrap: true,  
 itemCount: eventsForDay.length,  
 itemBuilder: (context, index) {  
 final event = eventsForDay[index];  
 return ListTile(  
 leading: CircleAvatar(backgroundColor: event.color),  
 title: Text(event.title),  
 subtitle: Text(  
 '${event.start.hour.toString().padLeft(2, '0')}:${event.start.minute.toString().padLeft(2, '0')}'  
 ' - '  
 '${event.end.hour.toString().padLeft(2, '0')}:${event.end.minute.toString().padLeft(2, '0')}'  
 '${event.location.isNotEmpty ? '\n@ ${event.location}' : ''}'  
 '${event.description.isNotEmpty ? '\n${event.description}' : ''}',  
 ),  
 onTap: () => \_addOrEditEvent(event: event),  
 trailing: IconButton(  
 icon: Icon(Icons.*delete*, color: Colors.*red*),  
 tooltip: 'Move to Trash',  
 onPressed: () => \_softDeleteEvent(event),  
 ),  
 );  
 },  
 );  
 }  
  
 @override  
 Widget build(BuildContext context) {  
 return Scaffold(  
 appBar: AppBar(  
 title: Text('Calendar Planner'),  
 actions: [  
 IconButton(  
 icon: Icon(Icons.*delete*),  
 tooltip: 'Trash',  
 onPressed: \_openTrash,  
 ),  
 IconButton(  
 icon: Icon(Icons.*calendar\_today*),  
 onPressed: () async {  
 DateTime? picked = await showDatePicker(  
 context: context,  
 initialDate: \_focusedDay,  
 firstDate: DateTime(2020),  
 lastDate: DateTime(2030),  
 );  
 if (picked != null && picked != \_focusedDay) {  
 setState(() {  
 \_focusedDay = picked;  
 \_selectedDay = picked;  
 });  
 }  
 },  
 ),  
 ],  
 ),  
 floatingActionButton: FloatingActionButton(  
 child: Icon(Icons.*add*),  
 tooltip: 'Add Event',  
 onPressed: () =>  
 \_addOrEditEvent(initialDate: \_selectedDay ?? \_focusedDay),  
 ),  
 body: Column(  
 children: [  
 Padding(  
 padding: const EdgeInsets.symmetric(vertical: 8.0),  
 child: DropdownButtonHideUnderline(  
 child: DropdownButton2<String>(  
 isExpanded: true,  
 value: widget.themeMode == ThemeMode.light ? 'Light' : 'Dark',  
 items: [  
 DropdownMenuItem<String>(  
 value: 'Light',  
 child: Text('Light Mode'),  
 ),  
 DropdownMenuItem<String>(  
 value: 'Dark',  
 child: Text('Dark Mode'),  
 ),  
 ],  
 onChanged: (value) {  
 if (value == 'Light') {  
 widget.onThemeModeChanged(ThemeMode.light);  
 } else {  
 widget.onThemeModeChanged(ThemeMode.dark);  
 }  
 },  
 buttonStyleData: ButtonStyleData(  
 height: 40,  
 width: 140,  
 decoration: BoxDecoration(  
 borderRadius: BorderRadius.circular(14),  
 border: Border.all(color: Colors.*blueAccent*),  
 color: Colors.*white*,  
 ),  
 ),  
 dropdownStyleData: DropdownStyleData(  
 maxHeight: 200,  
 decoration: BoxDecoration(  
 borderRadius: BorderRadius.circular(14),  
 color: Colors.*white*,  
 ),  
 ),  
 menuItemStyleData: MenuItemStyleData(  
 height: 40,  
 padding: EdgeInsets.symmetric(horizontal: 16),  
 ),  
 ),  
 ),  
 ),  
 TableCalendar(  
 firstDay: DateTime.utc(2020, 1, 1),  
 lastDay: DateTime.utc(2030, 12, 31),  
 focusedDay: \_focusedDay,  
 calendarFormat: \_calendarFormat,  
 selectedDayPredicate: (day) {  
 return isSameDay(\_selectedDay, day);  
 },  
 eventLoader: (day) => \_getEventsForDay(day),  
 onDaySelected: (selectedDay, focusedDay) {  
 setState(() {  
 \_selectedDay = selectedDay;  
 \_focusedDay = focusedDay;  
 });  
 },  
 onFormatChanged: (format) {  
 setState(() {  
 \_calendarFormat = format;  
 });  
 },  
 calendarStyle: CalendarStyle(  
 todayDecoration: BoxDecoration(  
 color: Colors.*blueAccent*,  
 shape: BoxShape.circle,  
 ),  
 selectedDecoration: BoxDecoration(  
 color: Colors.*orange*,  
 shape: BoxShape.circle,  
 ),  
 ),  
 headerStyle: HeaderStyle(  
 formatButtonVisible: true,  
 titleCentered: true,  
 formatButtonShowsNext: false,  
 formatButtonDecoration: BoxDecoration(  
 color: Colors.*blueAccent*,  
 borderRadius: BorderRadius.circular(12.0),  
 ),  
 formatButtonTextStyle: TextStyle(color: Colors.*white*),  
 ),  
 calendarBuilders: CalendarBuilders(  
 markerBuilder: (context, date, events) {  
 if (events.isNotEmpty) {  
 return Row(  
 mainAxisAlignment: MainAxisAlignment.center,  
 children: List.generate(events.length, (index) {  
 final event = events[index] as MyEvent;  
 return Container(  
 width: 6,  
 height: 6,  
 margin:  
 EdgeInsets.symmetric(horizontal: 1.0, vertical: 2),  
 decoration: BoxDecoration(  
 shape: BoxShape.circle,  
 color: event.color,  
 ),  
 );  
 }),  
 );  
 }  
 return SizedBox.shrink();  
 },  
 ),  
 ),  
 Expanded(  
 child: Column(  
 children: [  
 Expanded(  
 child: SfCalendar(  
 view: CalendarView.day, // Default to day view  
 dataSource: EventDataSource(\_getDataSource()),  
 timeSlotViewSettings: TimeSlotViewSettings(  
 timeInterval: Duration(minutes: 30),  
 timeFormat: 'h:mm a',  
 timeRulerSize: 60,  
 timeTextStyle: TextStyle(  
 color: Colors.*blueGrey*,  
 fontWeight: FontWeight.*bold*,  
 fontSize: 14,  
 ),  
 ),  
 todayHighlightColor: Colors.*blueAccent*,  
 appointmentTextStyle: TextStyle(  
 color: Colors.*white*,  
 fontWeight: FontWeight.*bold*,  
 ),  
 ),  
 ),  
 \_buildEventList(),  
 ],  
 ),  
 ),  
 ],  
 ),  
 );  
 }  
}  
  
class EventDialog extends StatefulWidget {  
 final MyEvent? event;  
 final DateTime initialDate;  
  
 EventDialog({this.event, required this.initialDate});  
  
 @override  
 \_EventDialogState createState() => \_EventDialogState();  
}  
  
class \_EventDialogState extends State<EventDialog> {  
 late TextEditingController \_titleController;  
 late TextEditingController \_locationController;  
 late TextEditingController \_descriptionController;  
 late DateTime \_start;  
 late DateTime \_end;  
 late Color \_color;  
  
 final List<Color> \_colors = [  
 Colors.*blue*,  
 Colors.*orange*,  
 Colors.*red*,  
 Colors.*green*,  
 Colors.*purple*,  
 Colors.*teal*,  
 ];  
  
 @override  
 void initState() {  
 super.initState();  
 \_titleController = TextEditingController(text: widget.event?.title ?? '');  
 \_locationController =  
 TextEditingController(text: widget.event?.location ?? '');  
 \_descriptionController =  
 TextEditingController(text: widget.event?.description ?? '');  
 \_start = widget.event?.start ?? widget.initialDate;  
 \_end = widget.event?.end ?? \_start.add(Duration(hours: 1));  
 \_color = widget.event?.color ?? \_colors[0];  
 }  
  
 @override  
 Widget build(BuildContext context) {  
 return AlertDialog(  
 title: Text(widget.event != null ? 'Edit Event' : 'Add Event'),  
 content: SingleChildScrollView(  
 child: Column(  
 mainAxisSize: MainAxisSize.min,  
 children: [  
 TextField(  
 controller: \_titleController,  
 decoration: InputDecoration(labelText: 'Title'),  
 autofocus: true,  
 ),  
 TextField(  
 controller: \_locationController,  
 decoration: InputDecoration(labelText: 'Location'),  
 ),  
 TextField(  
 controller: \_descriptionController,  
 decoration: InputDecoration(labelText: 'Description'),  
 ),  
 Row(  
 children: [  
 Expanded(  
 child: ListTile(  
 title: Text('Start: ${\_formatDateTime(\_start)}'),  
 trailing: Icon(Icons.*access\_time*),  
 onTap: () async {  
 final pickedDate = await showDatePicker(  
 context: context,  
 initialDate: \_start,  
 firstDate: DateTime(2020),  
 lastDate: DateTime(2030),  
 );  
 if (pickedDate != null) {  
 final pickedTime = await showTimePicker(  
 context: context,  
 initialTime: TimeOfDay.fromDateTime(\_start),  
 );  
 if (pickedTime != null) {  
 setState(() {  
 \_start = DateTime(  
 pickedDate.year,  
 pickedDate.month,  
 pickedDate.day,  
 pickedTime.hour,  
 pickedTime.minute,  
 );  
 if (\_end.isBefore(\_start)) {  
 \_end = \_start.add(Duration(hours: 1));  
 }  
 });  
 }  
 }  
 },  
 ),  
 ),  
 Expanded(  
 child: ListTile(  
 title: Text('End: ${\_formatDateTime(\_end)}'),  
 trailing: Icon(Icons.*access\_time*),  
 onTap: () async {  
 final pickedDate = await showDatePicker(  
 context: context,  
 initialDate: \_end,  
 firstDate: DateTime(2020),  
 lastDate: DateTime(2030),  
 );  
 if (pickedDate != null) {  
 final pickedTime = await showTimePicker(  
 context: context,  
 initialTime: TimeOfDay.fromDateTime(\_end),  
 );  
 if (pickedTime != null) {  
 setState(() {  
 \_end = DateTime(  
 pickedDate.year,  
 pickedDate.month,  
 pickedDate.day,  
 pickedTime.hour,  
 pickedTime.minute,  
 );  
 if (\_end.isBefore(\_start)) {  
 \_end = \_start.add(Duration(hours: 1));  
 }  
 });  
 }  
 }  
 },  
 ),  
 ),  
 ],  
 ),  
 SizedBox(height: 8),  
 Row(  
 mainAxisAlignment: MainAxisAlignment.center,  
 children: \_colors.map((c) {  
 return GestureDetector(  
 onTap: () {  
 setState(() {  
 \_color = c;  
 });  
 },  
 child: Container(  
 margin: EdgeInsets.symmetric(horizontal: 4),  
 width: 32,  
 height: 32,  
 decoration: BoxDecoration(  
 color: c,  
 shape: BoxShape.circle,  
 border: Border.all(  
 color: \_color == c ? Colors.*black* : Colors.*transparent*,  
 width: 2,  
 ),  
 ),  
 child: \_color == c  
 ? Icon(Icons.*check*, color: Colors.*white*)  
 : null,  
 ),  
 );  
 }).toList(),  
 ),  
 ],  
 ),  
 ),  
 actions: [  
 if (widget.event != null)  
 TextButton(  
 onPressed: () {  
 Navigator.*pop*(context, widget.event!.copyWith(isDeleted: true));  
 },  
 child: Text('Delete', style: TextStyle(color: Colors.*red*)),  
 ),  
 ElevatedButton(  
 onPressed: () {  
 if (\_titleController.text.trim().isNotEmpty) {  
 Navigator.*pop*(  
 context,  
 MyEvent(  
 id: widget.event?.id,  
 title: \_titleController.text.trim(),  
 start: \_start,  
 end: \_end,  
 color: \_color,  
 location: \_locationController.text.trim(),  
 description: \_descriptionController.text.trim(),  
 isDeleted: false,  
 ),  
 );  
 }  
 },  
 child: Text(widget.event != null ? 'Save' : 'Add'),  
 ),  
 TextButton(  
 onPressed: () => Navigator.*pop*(context),  
 child: Text('Cancel'),  
 ),  
 ],  
 );  
 }  
  
 String \_formatDateTime(DateTime dt) {  
 return '${dt.year}-${dt.month.toString().padLeft(2, '0')}-${dt.day.toString().padLeft(2, '0')} '  
 '${dt.hour.toString().padLeft(2, '0')}:${dt.minute.toString().padLeft(2, '0')}';  
 }  
}

**database\_helper.dart:**

import 'package:sqflite/sqflite.dart';  
import 'package:path/path.dart';  
import 'package:flutter/material.dart';  
import 'package:planner/main.dart';  
  
class DatabaseHelper {  
 static final DatabaseHelper *\_instance* = DatabaseHelper.\_internal();  
 static Database? *\_database*;  
  
 factory DatabaseHelper() => *\_instance*;  
  
 DatabaseHelper.\_internal();  
  
 Future<Database> get database async {  
 if (*\_database* != null) return *\_database*!;  
 *\_database* = await \_initDatabase();  
 return *\_database*!;  
 }  
  
 Future<Database> \_initDatabase() async {  
 String path = join(await getDatabasesPath(), 'calendar\_planner.db');  
 return await openDatabase(  
 path,  
 version: 1,  
 onCreate: \_onCreate,  
 );  
 }  
  
 Future<void> \_onCreate(Database db, int version) async {  
 await db.execute('''  
 CREATE TABLE events(  
 id INTEGER PRIMARY KEY AUTOINCREMENT,  
 title TEXT NOT NULL,  
 start\_time TEXT NOT NULL,  
 end\_time TEXT NOT NULL,  
 color INTEGER NOT NULL,  
 location TEXT,  
 description TEXT,  
 is\_deleted INTEGER DEFAULT 0  
 )  
 ''');  
 }  
  
 // Create event  
 Future<int> insertEvent(MyEvent event) async {  
 final db = await database;  
 return await db.insert('events', {  
 'title': event.title,  
 'start\_time': event.start.toIso8601String(),  
 'end\_time': event.end.toIso8601String(),  
 'color': event.color.value,  
 'location': event.location,  
 'description': event.description,  
 'is\_deleted': event.isDeleted ? 1 : 0,  
 });  
 }  
  
 // Read all active events  
 Future<List<MyEvent>> getActiveEvents() async {  
 final db = await database;  
 final List<Map<String, dynamic>> maps = await db.query(  
 'events',  
 where: 'is\_deleted = ?',  
 whereArgs: [0],  
 );  
  
 return List.generate(maps.length, (i) {  
 return MyEvent(  
 id: maps[i]['id'],  
 title: maps[i]['title'],  
 start: DateTime.*parse*(maps[i]['start\_time']),  
 end: DateTime.*parse*(maps[i]['end\_time']),  
 color: Color(maps[i]['color']),  
 location: maps[i]['location'] ?? '',  
 description: maps[i]['description'] ?? '',  
 isDeleted: maps[i]['is\_deleted'] == 1,  
 );  
 });  
 }  
  
 // Read all events (including deleted)  
 Future<List<MyEvent>> getAllEvents() async {  
 final db = await database;  
 final List<Map<String, dynamic>> maps = await db.query('events');  
  
 return List.generate(maps.length, (i) {  
 return MyEvent(  
 id: maps[i]['id'],  
 title: maps[i]['title'],  
 start: DateTime.*parse*(maps[i]['start\_time']),  
 end: DateTime.*parse*(maps[i]['end\_time']),  
 color: Color(maps[i]['color']),  
 location: maps[i]['location'] ?? '',  
 description: maps[i]['description'] ?? '',  
 isDeleted: maps[i]['is\_deleted'] == 1,  
 );  
 });  
 }  
  
 // Update event  
 Future<int> updateEvent(MyEvent event) async {  
 final db = await database;  
 return await db.update(  
 'events',  
 {  
 'title': event.title,  
 'start\_time': event.start.toIso8601String(),  
 'end\_time': event.end.toIso8601String(),  
 'color': event.color.value,  
 'location': event.location,  
 'description': event.description,  
 'is\_deleted': event.isDeleted ? 1 : 0,  
 },  
 where: 'id = ?',  
 whereArgs: [event.id],  
 );  
 }  
  
 // Soft delete event  
 Future<int> softDeleteEvent(int id) async {  
 final db = await database;  
 return await db.update(  
 'events',  
 {'is\_deleted': 1},  
 where: 'id = ?',  
 whereArgs: [id],  
 );  
 }  
  
 // Restore event  
 Future<int> restoreEvent(int id) async {  
 final db = await database;  
 return await db.update(  
 'events',  
 {'is\_deleted': 0},  
 where: 'id = ?',  
 whereArgs: [id],  
 );  
 }  
  
 // Hard delete event  
 Future<int> hardDeleteEvent(int id) async {  
 final db = await database;  
 return await db.delete(  
 'events',  
 where: 'id = ?',  
 whereArgs: [id],  
 );  
 }  
  
 // Get events for a specific day  
 Future<List<MyEvent>> getEventsForDay(DateTime day) async {  
 final db = await database;  
 final startOfDay = DateTime(day.year, day.month, day.day);  
 final endOfDay = startOfDay.add(Duration(days: 1));  
  
 final List<Map<String, dynamic>> maps = await db.query(  
 'events',  
 where: 'start\_time >= ? AND start\_time < ? AND is\_deleted = ?',  
 whereArgs: [startOfDay.toIso8601String(), endOfDay.toIso8601String(), 0],  
 );  
  
 return List.generate(maps.length, (i) {  
 return MyEvent(  
 id: maps[i]['id'],  
 title: maps[i]['title'],  
 start: DateTime.*parse*(maps[i]['start\_time']),  
 end: DateTime.*parse*(maps[i]['end\_time']),  
 color: Color(maps[i]['color']),  
 location: maps[i]['location'] ?? '',  
 description: maps[i]['description'] ?? '',  
 isDeleted: maps[i]['is\_deleted'] == 1,  
 );  
 });  
 }  
  
 // Close database  
 Future<void> close() async {  
 final db = await database;  
 await db.close();  
 }  
}

**build.graddle.kts**

allprojects {  
 repositories {  
 google()  
 mavenCentral()  
 }  
}  
  
val newBuildDir: Directory = rootProject.layout.buildDirectory.dir("../../build").get()  
rootProject.layout.buildDirectory.value(newBuildDir)  
  
subprojects {  
 val newSubprojectBuildDir: Directory = newBuildDir.dir(project.name)  
 project.layout.buildDirectory.value(newSubprojectBuildDir)  
}  
subprojects {  
 project.evaluationDependsOn(":app")  
}  
  
tasks.register<Delete>("clean") {  
 delete(rootProject.layout.buildDirectory)  
}  
  
//android {  
 // ... other config ...  
 //ndkVersion = "27.0.12077973"  
 // ... other config ...  
//}