Okay, let's break down the approach to building a full-stack CRUD module in Flutter, based on the provided document.

1. Project Setup and Structure

- **Flutter Project:** Create a new Flutter project.
- **Packages:** Add necessary packages:
  - flutter_bloc: For the BLOC pattern.
  - http or dio: For making API requests.
  - sqflite or hive: For local database.
- **Layered Architecture:** Organize your project into layers:
  - **Data Layer:**
    - models/: Define data models (e.g., task.dart).
    - repositories/: Abstract data sources.
  - **Business Logic Layer:**
    - blocs/: Implement BLOC components for state management.
    - services/: API services.
  - **Presentation Layer:**
    - screens/: UI screens for listing, adding, editing, etc.
    - widgets/: Reusable UI components.

2. Data Model

- Define a Dart class representing your data (e.g., Task) with fields like title, description, status, createdDate, and priority.

<!-- end list -->

```
class Task {
  int? id;
  String title;
  String description;
  String status;
  DateTime createdDate;
  String priority;

  Task({
    this.id,
    required this.title,
    required this.description,
    required this.status,
    required this.createdDate,
    required this.priority,
  });

  // Serialization/Deserialization (e.g., toJson, fromJson)
}
```

3. API Integration

- **API Service:** Create a class to handle API calls using http or dio.
  - Implement methods for CRUD operations (e.g., getTasks, createTask, updateTask, deleteTask).
  - Handle JSON serialization/deserialization.
  - Include error handling and retry logic.

    - (If needed) Implement token-based authentication.

<!-- end list -->

```
class TaskApiService {
  final String baseUrl = 'YOUR_API_BASE_URL'; // Replace with your API
URL
  final http.Client client; // Or use dio

  TaskApiService({required this.client});

  Future<List<Task>> getTasks() async {
    final response = await client.get(Uri.parse('$baseUrl/tasks'));
    if (response.statusCode == 200) {
      // ... decode JSON and return list of Task objects
    } else {
      throw Exception('Failed to load tasks');
    }
  }

  // Implement createTask, updateTask, deleteTask...
}
```

4. Local Data Persistence

- **SQLite (sqflite) or Hive:** Choose a local database solution.
- **Database Helper:** Create a class to manage database operations:
    - Initialize the database.
    - Define the database schema and tables.
    - Implement methods for CRUD operations on the local database.
    - Use transactions for multiple record changes.
    - Create indexes for efficient queries.

<!-- end list -->

```
import 'package:sqflite/sqflite.dart';

class TaskDatabaseHelper {
  static Database? _database;
  static const String _tableName = 'tasks';

  Future<Database> get database async {
    if (_database != null) return _database!;
    _database = await _initDatabase();
    return _database!;
  }

  Future<Database> _initDatabase() async {
    final path = await getDatabasesPath();
    final dbPath = '$path/tasks.db';
    return await openDatabase(
      dbPath,
      version: 1,
```

```
      onCreate: _onCreate,
    );
  }

  Future<void> _onCreate(Database db, int version) async {
    await db.execute('''
      CREATE TABLE $_tableName (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        title TEXT,
        description TEXT,
        status TEXT,
        createdDate INTEGER, // Store as Unix timestamp
        priority TEXT
      )
    ''');
    await db.execute('CREATE INDEX idx_tasks_status ON $_tableName
(status)'); // Example index
  }

  // Implement CRUD operations (insertTask, getTasks, updateTask,
deleteTask)
}
```

5. Repository Pattern

- **TaskRepository:** Abstracts the data sources (API and local database).
    - Provides a unified interface for data access to the BLOC.
    - Decides whether to fetch data from the API or local database (e.g., based on network connectivity).
    - Handles data synchronization between local and remote data.

<!-- end list -->

```
class TaskRepository {
  final TaskApiService apiService;
  final TaskDatabaseHelper databaseHelper;

  TaskRepository({required this.apiService, required
this.databaseHelper});

  Future<List<Task>> getTasks() async {
    try {
      final tasks = await apiService.getTasks();
      // Optionally save to local database
      return tasks;
    } catch (e) {
      // If API fails, fetch from local database
      return await databaseHelper.getTasks();
    }
  }
```

```
  // Implement other CRUD operations, handling both API and local
database
}
```

6. BLOC Pattern

- **BLOCs and Events:** Create BLOC components to manage the state of your task list.
    - Define events (e.g., LoadTasks, AddTask, UpdateTask, DeleteTask).
    - Define states (e.g., TasksLoading, TasksLoaded, TasksError).
- **TaskBloc:** Handles the business logic.
    - Responds to events by interacting with the TaskRepository.
    - Emits appropriate states to notify the UI.
    - Handles error states.

<!-- end list -->

```
import 'package:flutter_bloc/flutter_bloc.dart';

// Events
abstract class TaskEvent {}
class LoadTasks extends TaskEvent {}
class AddTask extends TaskEvent { final Task task; AddTask(this.task);
}
class UpdateTask extends TaskEvent { final Task task;
UpdateTask(this.task); }
class DeleteTask extends TaskEvent { final int id;
DeleteTask(this.id); }

// States
abstract class TaskState {}
class TasksLoading extends TaskState {}
class TasksLoaded extends TaskState { final List<Task> tasks;
TasksLoaded(this.tasks); }
class TasksError extends TaskState { final String message;
TasksError(this.message); }

class TaskBloc extends Bloc<TaskEvent, TaskState> {
  final TaskRepository taskRepository;

  TaskBloc({required this.taskRepository}) : super(TasksLoading()) {
    on<LoadTasks>((event, emit) async {
      emit(TasksLoading());
      try {
        final tasks = await taskRepository.getTasks();
        emit(TasksLoaded(tasks: tasks));
      } catch (e) {
        emit(TasksError(message: e.toString()));
      }
    });

    // Handle AddTask, UpdateTask, DeleteTask events...
```

```
  }
}
```

7. UI (Presentation Layer)

- **Screens:** Create screens for:
  - **Task List:** Displays the list of tasks (using ListView or GridView).
  - **Add Task:** A form to create new tasks.
  - **Edit Task:** A form to edit existing tasks.
- **Widgets:** Create reusable UI components (e.g., task card).
- **BlocBuilder:** Use BlocBuilder to connect the UI to the TaskBloc and rebuild when the state changes.
- **User Feedback:** Provide feedback using SnackBar for success/failure.
- **Navigation:** Implement proper navigation between screens.
- **Validation:** Use form validators for add/edit operations.
- **Responsive UI:** Design the UI to adapt to different screen sizes.
- **Loading/Empty States:** Handle loading states and display appropriate messages when there are no tasks.
- **Error Handling:** Display error messages to the user.
- **Optimistic Updates:** Consider optimistic updates for a smoother user experience.
- **Delete Confirmation:** Use dialogs or swipe-to-delete for confirmation before deleting.

<!-- end list -->

```
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';

class TaskListScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Tasks')),
      body: BlocBuilder<TaskBloc, TaskState>(
        builder: (context, state) {
          if (state is TasksLoading) {
            return Center(child: CircularProgressIndicator());
          } else if (state is TasksLoaded) {
            return ListView.builder(
              itemCount: state.tasks.length,
              itemBuilder: (context, index) {
                final task = state.tasks[index];
                return TaskCard(task: task); // Custom widget
              },
            );
          } else if (state is TasksError) {
            return Center(child: Text('Error: ${state.message}'));
          } else {
            return Container(); // Or a default empty state widget
          }
        },
      ),
```

```
      floatingActionButton: FloatingActionButton(
        child: Icon(Icons.add),
        onPressed: () {
          Navigator.push(context, MaterialPageRoute(builder: (context)
=> AddTaskScreen()));
        },
      ),
    );
  }
}

class TaskCard extends StatelessWidget {
  final Task task;
  const TaskCard({Key? key, required this.task}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Card(
      child: ListTile(
        title: Text(task.title),
        subtitle: Text(task.description),
        trailing: Row(
          mainAxisSize: MainAxisSize.min,
          children: [
            IconButton(
              icon: Icon(Icons.edit),
              onPressed: () {
                Navigator.push(context, MaterialPageRoute(builder:
(context) => EditTaskScreen(task: task)));
              },
            ),
            IconButton(
              icon: Icon(Icons.delete),
              onPressed: () {
                context.read<TaskBloc>().add(DeleteTask(id:
task.id!));
              },
            ),
          ],
        ),
      ),
    );
  }
}

class AddTaskScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
```

```
    return Scaffold(
      appBar: AppBar(title: Text('Add Task')),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: TaskForm(), // Custom form widget
      ),
    );
  }
}

class EditTaskScreen extends StatelessWidget {
  final Task task;
  const EditTaskScreen({Key? key, required this.task}) : super(key:
key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Edit Task')),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: TaskForm(task: task), // Reuse form, pass task data
      ),
    );
  }
}

class TaskForm extends StatefulWidget {
  final Task? task;
  const TaskForm({Key? key, this.task}) : super(key: key);

  @override
  _TaskFormState createState() => _TaskFormState();
}

class _TaskFormState extends State<TaskForm> {
  final _formKey = GlobalKey<FormState>();
  final _titleController = TextEditingController();
  final _descriptionController = TextEditingController();
  final _statusController = TextEditingController();
  final _priorityController = TextEditingController();

  @override
  void initState() {
    super.initState();
    if (widget.task != null) {
      _titleController.text = widget.task!.title;
      _descriptionController.text = widget.task!.description;
```

```
    _statusController.text = widget.task!.status;
    _priorityController.text = widget.task!.priority;
  }
}

@override
void dispose() {
  _titleController.dispose();
  _descriptionController.dispose();
  _statusController.dispose();
  _priorityController.dispose();
  super.dispose();
}

@override
Widget build(BuildContext context) {
  return Form(
    key: _formKey,
    child: Column(
      children: [
        TextFormField(
          controller: _titleController,
          decoration: InputDecoration(labelText: 'Title'),
          validator: (value) {
            if (value == null || value.isEmpty) {
              return 'Please enter a title';
            }
            return null;
          },
        ),
        TextFormField(
          controller: _descriptionController,
          decoration: InputDecoration(labelText: 'Description'),
          validator: (value) {
            if (value == null || value.isEmpty) {
              return 'Please enter a description';
            }
            return null;
          },
        ),
        TextFormField(
          controller: _statusController,
          decoration: InputDecoration(labelText: 'Status'),
          validator: (value) {
            if (value == null || value.isEmpty) {
              return 'Please enter a status';
            }
            return null;
```

```
            },
          ),
          TextFormField(
            controller: _priorityController,
            decoration: InputDecoration(labelText: 'Priority'),
            validator: (value) {
              if (value == null || value.isEmpty) {
                return 'Please enter a priority';
              }
              return null;
            },
          ),
          ElevatedButton(
            onPressed: () {
              if (_formKey.currentState!.validate()) {
                final task = Task(
                  id: widget.task?.id,
                  title: _titleController.text,
                  description: _descriptionController.text,
                  status: _statusController.text,
                  createdDate: widget.task?.createdDate ??
DateTime.now(),
                  priority: _priorityController.text,
                );
                if (widget.task == null) {
                  context.read<TaskBloc>().add(AddTask(task));
                } else {
                  context.read<TaskBloc>().add(UpdateTask(task));
                }
                Navigator.pop(context);
              }
            },
            child: Text(widget.task == null ? 'Add Task' : 'Update
Task'),
          ),
        ],
      ),
    );
  }
}
```

8. Dependency Injection

- Consider using a dependency injection package (e.g., get_it, provider) for better testability and managing dependencies.

9. Additional Considerations

- **Network Connectivity:** Handle network changes (e.g., using the connectivity_plus package).
- **Logging:** Add logging for debugging.

- **Testing:** Write unit and integration tests.
- **README:** Provide clear documentation.

This detailed breakdown, combined with the code snippets, should give you a solid foundation for building your full-stack CRUD module in Flutter!