



Explore | Expand | Enrich

# Mobile Application Development Test Projects

Ethnus  
Mar 2025

<b>Mobile Application Development Test Projects.....</b>	<b>3</b>
College Level (2.5 hours).....	3
State Level (3.5 hours).....	6
National Level (5 hours).....	10



Explore | Expand | Enrich

# Mobile Application Development Test Projects

## College Level (2.5 hours)

- **Objective:** Build a simple, single-screen static Flutter application displaying information, using basic layout widgets and handling a simple button press action.
- **Scenario:** "Innovate Startups" wants a simple digital "business card" app to showcase basic contact info.
- **Prerequisites:**
  - Pre-configured Windows PC: Android Studio, Flutter SDK installed and configured.
  - Android Emulator running OR a physical Android device connected and enabled for debugging.
  - VS Code (with Flutter/Dart extensions) OR Android Studio as IDE.
  - Provided assets: A placeholder logo image (`logo.png` or similar).
- **Detailed Steps/Modules:**
  - **Module 1: Project Setup & Basic Structure (Approx. 30 mins)**
    - 1.1. Open your IDE (VS Code/Android Studio).
    - 1.2. Create a new Flutter project using the command palette or terminal: `flutter create innovate_card_app`.
    - 1.3. Open the created project folder. Locate the `lib/main.dart` file.
    - 1.4. Clean up the default counter app code within `main.dart`. Create a basic `StatelessWidget` for your main application structure (e.g., `InnovateCardApp`) returning a `MaterialApp`.
    - 1.5. Inside `MaterialApp`, define a `home` property pointing to another `StatelessWidget` which will hold the card layout (e.g., `BusinessCardScreen`).

- 1.6. Run the basic template app on your emulator/device to ensure the build environment is working: `flutter run`. You should see a blank screen or basic scaffold.
- **Module 2: Implementing the UI Layout (Approx. 75 mins)**
  - 2.1. In your `BusinessCardScreen` widget, return a `Scaffold` widget. Give it an `AppBar` with a `Text` title (e.g., "Innovate Startups Contact").
  - 2.2. Set the `body` of the `Scaffold` to a `Center` widget containing a `Column` widget to arrange elements vertically.
  - 2.3. Inside the `Column`, add the following widgets (using appropriate padding via `Padding` or `Container` margins):
    - A `CircleAvatar` or `Container` to display the provided `logo.png` asset. (Remember to declare the asset folder in `pubspec.yaml` and create an `assets` folder).
    - A `Text` widget displaying the company name (e.g., "Innovate Startups Inc.") with appropriate styling (font size, weight using `TextStyle`).
    - A `Text` widget displaying a title/slogan (e.g., "Innovating the Future").
    - An `Icon` and `Text` widget pair inside a `Row` for the phone number (e.g., `Icons.phone`, "+1 234 567 890").
    - An `Icon` and `Text` widget pair inside a `Row` for the email address (e.g., `Icons.email`, "contact@innovatestartups.example").
    - A `SizedBox` or `Divider` for visual separation.
  - 2.4. Arrange the widgets within the `Column` and `Rows` using properties like `mainAxisAlignment`, `crossAxisAlignment` for alignment. Use `Padding` widgets or `Container` margins/padding for spacing.
- **Module 3: Basic Interaction (Approx. 45 mins)**

- 3.1. Add an `ElevatedButton` widget at the bottom of the `Column`. Set its `child` to a `Text` widget (e.g., "Show Message").
  - 3.2. Implement the `onPressed` callback for the `ElevatedButton`.
  - 3.3. Inside `onPressed`, use `ScaffoldMessenger.of(context).showSnackBar(...)` to display a simple `SnackBar` with a `Text` message (e.g., "Contact info displayed!").
  - 3.4. Run the app (`flutter run`) and test the layout on different screen sizes (if possible using emulator settings) and verify the button shows the `SnackBar` message when pressed.
- **Deliverables:**
    - Submit the complete Flutter project folder (including `lib`, `pubspec.yaml`, `assets` folder, etc.).
  - **Verification:**
    - Run the application on an emulator or device.
    - Verify the UI matches the described layout and displays all static information (logo, name, title, phone, email).
    - Verify layout elements are reasonably aligned and spaced.
    - Verify clicking the button triggers the `SnackBar` message correctly.

Explore | Expand | Enrich

## State Level (3.5 hours)

- **Objective:** Build a multi-screen Flutter application that fetches data (e.g., list of quotes or articles) from a provided mock API, displays it in a list, allows navigation to a detail view (optional) or an about page, and uses basic state management (`StatefulWidget`).
- **Scenario:** "Innovate Startups" wants a simple app to display inspirational quotes fetched from their new API.
- **Prerequisites:**
  - Same as College Level (Working Flutter environment, Emulator/Device).
  - **Provided Mock API Endpoint URL:** A URL (e.g., `https://api.example.com/quotes`) that returns a JSON array of objects (e.g., `{ "id": 1, "text": "Quote text...", "author": "Author Name" }`).
- **Detailed Steps/Modules:**
  - **Module 1: Project Setup & Packages (Approx. 30 mins)**
    - 1.1. Create a new Flutter project (e.g., `innovate_quotes_app`).
    - 1.2. Open `pubspec.yaml`. Add the `http` package under `dependencies`: (`http: ^1.x.x` - use latest compatible version).
    - 1.3. Run `flutter pub get` in the terminal within the project directory.
    - 1.4. Set up basic `MaterialApp` structure in `main.dart`. Define routes or initial home screen.
  - **Module 2: Fetching and Displaying Data (Approx. 75 mins)**
    - 2.1. Create a new Dart file for your main screen (e.g., `quote_list_screen.dart`). Make this screen a `StatefulWidget` because its content (the list of quotes) will change.
    - 2.2. In the `State` class of `QuoteListScreen`:
      - Declare state variables: `_quotes` (a list to hold fetched quotes, initially empty) and `_isLoading` (a boolean, initially true).

- Override the `initState()` method. Inside `initState()`, call a function to fetch the quotes (e.g., `_fetchQuotes()`).
- 2.3. Create the `_fetchQuotes()` async function:
  - Set `_isLoading = true` using `setState`.
  - Use the `http` package (`import 'package:http/http.dart' as http;`) to make a GET request to the provided mock API URL.
  - Handle the response: Check status code (e.g., 200). If successful, decode the JSON response body (`jsonDecode` from `dart:convert`). Map the decoded JSON list into a list of quote objects (create a simple Quote class or use Maps).
  - Update the state using `setState`: assign the fetched quotes to `_quotes` and set `_isLoading = false`.
  - Include basic error handling (e.g., print error, set an error message state variable, set `_isLoading = false`).
- 2.4. Implement the `build()` method for `QuoteListScreen`:
  - Return a `Scaffold` with an `AppBar` (e.g., "Innovate Quotes").
  - In the `body`, check the `_isLoading` state. If true, display a `CircularProgressIndicator`.
  - If false (and no error): Display a `ListView.builder`.
    - `itemCount`: `_quotes.length`.
    - `itemBuilder`: Returns a widget for each quote (e.g., a `ListTile` or a custom `Card` widget) displaying the quote `text` and `author`.
  - (Optional) If there was an error fetching, display an error message.
- **Module 3: Navigation (Approx. 45 mins)**
  - 3.1. Create a simple second screen (e.g., `about_screen.dart`) as a `StatelessWidget` displaying some static "About" text in a `Scaffold`.

- 3.2. In the `AppBar` of `QuoteListScreen`, add an `IconButton` (e.g., `Icons.info_outline`) to the `actions` list.
- 3.3. Implement the `onPressed` callback for the `IconButton`. Use `Navigator.push()` to navigate to the `AboutScreen`.
  - Example: `Navigator.push(context, MaterialPageRoute(builder: (context) => AboutScreen()));`
- 3.4. Ensure the `AboutScreen` has an `AppBar` allowing the user to navigate back automatically (default behavior) or add an explicit back button if needed.
- 3.5. (Optional Detail View): If time permits, make each item in the `ListView` tappable (`InkWell` or `GestureDetector`). On tap, navigate to a new `QuoteDetailScreen`, passing the selected quote object as an argument through the `MaterialPageRoute`'s settings or constructor. The `QuoteDetailScreen` would then display the full quote details.

○ **Module 4: State Management & Refinement (Approx. 30 mins)**

- 4.1. Add a "Refresh" button (e.g., another `IconButton` in the `AppBar` or a `FloatingActionButton`) to `QuoteListScreen`.
- 4.2. Implement its `onPressed` callback to simply call the `_fetchQuotes()` method again to reload the data. Ensure the loading indicator shows during refresh.
- 4.3. Review the use of `setState` to ensure it's used appropriately to update the UI when data or loading state changes.
- 4.4. Test the app thoroughly: initial load, navigation, refresh functionality, error handling (if possible to simulate API error).

● **Deliverables:**

- Submit the complete Flutter project folder.

● **Verification:**

- Run the application on an emulator or device.



- Verify the app fetches quotes from the API and displays them in a list upon loading.
- Verify the loading indicator is shown correctly.
- Verify navigation to the About screen (or Detail screen) works correctly using the AppBar button (or list tap).
- Verify the Refresh button re-fetches and updates the list.
- (If implemented) Verify error state is handled visually (e.g., error message displayed).

Explore | Expand | Enrich

## National Level (5 hours)

- **Objective:** Build a more complex "Todo List" application demonstrating effective state management (using Provider or Riverpod), local data persistence (`shared_preferences` or `sqflite`), and basic CRUD (Create, Read) interactions with mock API endpoints.
- **Scenario:** "Innovate Startups" needs a mobile companion app for their project management tool, allowing users to manage their personal Todo list. Data should persist locally and optionally sync (read/add) with a backend.
- **Prerequisites:**
  - Same as State Level (Working Flutter environment, Emulator/Device).
  - **Provided Mock API Endpoint URLs:**
    - `GET /api/todos`: Returns array of todos { "id": N, "title": "...", "isCompleted": bool }.
    - `POST /api/todos`: Accepts { "title": "...", "isCompleted": false }, returns the created todo with a new id.
  - Choice of State Management library (e.g., Provider) should align with training. Examples assume `provider`.
- **Detailed Steps/Modules:**
  - **Module 1: Project Setup & State Management Choice (Approx. 45 mins)**
    - 1.1. Create a new Flutter project (e.g., `innovate_todo_app`).
    - 1.2. Add necessary packages to `pubspec.yaml`:
      - State management: `provider: ^6.x.x` (use latest compatible).
      - Persistence: `shared_preferences: ^2.x.x` (easier) OR `sqflite: ^2.x.x + path_provider: ^2.x.x` (more robust, if trained). Let's assume `shared_preferences`.
      - API calls: `http: ^1.x.x`.
      - JSON handling: Relies on `dart:convert`.
    - 1.3. Run `flutter pub get`.

- 1.4. Set up the root widget (`main.dart`) to use the chosen state management provider (e.g., wrap `MaterialApp` with `ChangeNotifierProvider`). Create a basic `TodoProvider` class that extends `ChangeNotifier`.
- 1.5. Define a simple `Todo` model class (`todo.dart`) with fields like `id` (optional, could be `String` or `int`), `title` (`String`), `isCompleted` (`bool`). Include methods for JSON serialization/deserialization (`toJson`, `fromJson`) if needed, especially for `shared_preferences`.
- **Module 2: Displaying Todos & State Management (Approx. 75 mins)**
  - 2.1. Create the main `TodoListScreen` widget (can be `StatelessWidget` now, using `Consumer` or `context.watch` from `Provider`).
  - 2.2. In `TodoProvider`:
    - Define the list of todos: `List<Todo> _todos = [];`. Add a getter `List<Todo> get todos => _todos;`.
    - Add methods to load/fetch todos (e.g., `loadTodos()`).
    - Add methods to modify todos (e.g., `addTodo(String title)`, `toggleTodoStatus(String id)`, `removeTodo(String id)`). **Crucially, call `notifyListeners()` after any state modification.**
  - 2.3. Implement `loadTodos()` in the provider: Initially, this might just fetch from the mock API `GET /api/todos` (similar to State level fetch logic) and populate `_todos`, then call `notifyListeners()`. Error/loading state handling should also be in the provider.
  - 2.4. In `TodoListScreen`'s `build` method:
    - Access the `TodoProvider` using `Provider.of<TodoProvider>(context)` or `context.watch<TodoProvider>()`.
    - Display a loading indicator or error message based on provider state.

- If loaded, display the `provider.todos` list using `ListView.builder`.
- Each list item (e.g., `ListTile`) should show the `todo.title` and a `Checkbox`. The checkbox value should be `todo.isCompleted`. Its `onChanged` should call the provider's `toggleTodoStatus(todo.id)` method.
- Add a way to delete todos (e.g., a dismissible widget wrapper, or an icon button per item) that calls `provider.removeTodo(todo.id)`.
- 2.5. Call `loadTodos()` from the provider when the app/screen initializes (e.g., using `initState` in a `StatefulWidget` wrapper or another mechanism suitable for the chosen provider setup).
- **Module 3: Adding Todos via API & Form (Approx. 75 mins)**
  - 3.1. Add a way to navigate to an "Add Todo" screen or show a dialog/modal bottom sheet. E.g., a `FloatingActionButton` on `TodoListScreen`.
  - 3.2. Create an `AddTodoScreen` or widget containing a `TextField` for the title and a "Save" button.
  - 3.3. Use a `TextEditingController` for the `TextField`.
  - 3.4. Implement the "Save" button's `onPressed`:
    - Get the title from the controller.
    - **Input Validation:** Ensure the title is not empty. Show an error (e.g., `SnackBar` or `TextField` error text) if invalid.
    - If valid:
      - Show a loading indicator (optional).
      - Access the `TodoProvider` (`context.read<TodoProvider>()` inside callbacks).
      - Call a new method in the provider, e.g., `addNewTodoOnline(String title)`.
  - 3.5. Implement `addNewTodoOnline` in `TodoProvider`:

- Make a **POST** request using **http** to the provided **/api/todos** endpoint. Send `{ "title": title, "isCompleted": false }` as the JSON body. Set **Content-Type** header.
- Handle the response. If successful (e.g., status 201), parse the returned created Todo object.
- **Crucially:** Add the *newly created* Todo (from the response) to the local **\_todos** list and call **notifyListeners()** to update the UI immediately.
- Handle API errors.
- After completion (success or error), navigate back from the Add Todo screen/dialog (**Navigator.pop(context)**).
- **Module 4: Local Persistence with SharedPreferences (Approx. 60 mins)**
  - 4.1. Modify **TodoProvider** to save/load from **shared\_preferences**.
  - 4.2. Create **saveTodosToPrefs()** method:
    - Get the current **\_todos** list.
    - Convert the list of **Todo** objects to a list of Maps/JSON strings (`List<String> jsonList = _todos.map((todo) => jsonEncode(todo.toJson())).toList();`). Requires **toJson()** in **Todo** model.
    - Get **SharedPreferences** instance: `final prefs = await SharedPreferences.getInstance();`
    - Save the list: `await prefs.setStringList('todos', jsonList);`
  - 4.3. Create **loadTodosFromPrefs()** method:
    - Get **SharedPreferences** instance.
    - Read the list: `final List<String>? jsonList = prefs.getStringList('todos');`
    - If **jsonList** is not null:
      - Parse it back into **List<Todo>**: `_todos = jsonList.map((jsonString) =>`

`Todo.fromJson(jsonDecode(jsonString))).toList();` Requires `fromJson()` in `Todo` model.

- Else (first run or no data): Initialize `_todos = [];`
- Call `notifyListeners()`.
- 4.4. Modify initial loading: Change `loadTodos()` in the provider to first try `loadTodosFromPrefs()`. You might *still* fetch from the API afterwards to get updates, or just rely on local data for this exercise. Let's prioritize local: Call `loadTodosFromPrefs()` during provider initialization.
- 4.5. Modify data changing methods (`addTodo`, `toggleTodoStatus`, `removeTodo`): After updating the `_todos` list and calling `notifyListeners()`, also call `saveTodosToPrefs()` to persist the changes. *Note: For `addNewTodoOnline`, save to prefs after successfully getting the response and adding to the local list.*
- 4.6. Test persistence: Add/toggle/delete todos, close the app completely, reopen it, and verify the state is restored correctly.
- **Module 5: (Optional) Device Feature / Refinement (Approx. 30 mins)**
  - 5.1. **Geo-locator (if feasible/trained):** Add `geolocator` package. Add a button to `AddTodoScreen` to optionally attach current location (lat/long) to a `Todo` (would require modifying `Todo` model and persistence). Handle location permissions. Display location on `Todo` details (if detail view exists). *Complexity Note: Permissions and emulator support can be tricky.*
  - 5.2. **Pull-to-Refresh:** Wrap the `ListView` in `TodoListScreen` with a `RefreshIndicator` widget. Implement its `onRefresh` callback to call the provider's method for fetching/reloading data from the API (e.g., `provider.loadTodos()` if it includes API fetch logic).
  - 5.3. **Error Handling:** Improve visual feedback for API errors (e.g., display `SnackBar`s or specific error messages on screen instead of just print statements).

- **Deliverables:**

- Submit the complete Flutter project folder.
- **Verification:**
  - Run the application on an emulator or device.
  - Verify Todos are loaded (initially likely empty, then from API/Prefs) and displayed correctly with checkboxes.
  - Verify Checkbox toggles the status, and the change is reflected in the UI (via Provider) and persists after app restart (via SharedPreferences).
  - Verify deleting a Todo removes it from the UI and persists the change.
  - Verify navigating to the "Add Todo" screen/dialog works.
  - Verify the Add Todo form has validation (e.g., non-empty title).
  - Verify adding a valid Todo makes a POST request, updates the UI list, persists the change, and navigates back.
  - (If implemented) Verify pull-to-refresh or other optional features work.

– End of Document –

Explore | Expand | Enrich