Todo MVVM App Analysis

Project Structure:

```
📂 lib/
├── 📂 models/          # MODEL Layer
│   └── todo.dart       # Data structure
├── 📂 services/         # MODEL Layer
│   └── todo_service.dart       # Data structure
├── 📂 viewmodels/      # VIEWMODEL Layer
│   └── todo_viewmodel.dart # Business logic
├── 📂 views/           # VIEW Layer
│   ├── add_todo_view.dart
│   └── todo_list_view.dart  # UI components
├── pubspec.yaml
└── main.dart           # App setup
```

- Create a flutter project with `flutter create todo` and copy the files to start the project.
- Use `flutter run` to build and start the project.

The `pubspec.yaml`

```yaml
name: todo
description: A simple Flutter MVVM Todo example for educational purposes
version: 1.0.0+1

environment:
  sdk: '>=3.0.0 <4.0.0'
  flutter: ">=3.10.0"

dependencies:
  flutter:
    sdk: flutter
  provider: ^6.1.2
```

- The project name is `todo`: we can access the lib directory with `package:todo` namespace.
- Notice we use `provider` package for MVVM: we must run `flutter pub get` to download and use the provider package.

Model

```
class Todo {
  final String id;
  final String title;
  final String description;
  final bool isCompleted;
  final DateTime createdAt;

  Todo({
    required this.id,
    required this.title,
    this.description = "",
    this.isCompleted = false,
    DateTime? createdAt,
  }) : createdAt = createdAt ?? DateTime.now();
```

Creates a copy of this Todo with some updated values.

```
Todo copyWith({
  String? id,
  String? title,
  String? description,
  bool? isCompleted,
  DateTime? createdAt,
}) {
  return Todo(
    id: id ?? this.id,
    title: title ?? this.title,
    description: description ?? this.description,
    isCompleted: isCompleted ?? this.isCompleted,
    createdAt: createdAt ?? this.createdAt,
  );
}
```

JSON Transformation for Serialization

- In Flutter (and Dart in general), a Map<String, dynamic> is a natural representation of JSON-like data.
- Map<String, dynamic> is the bridge between Todo Dart objects and JSON.
- We can use these transformations whenever you need to serialize/deserialize your models.
- It works well with local storage (SQLite, SharedPreferences) and remote APIs (REST/GraphQL).
- It is easy to extend if you later add fields.

1. Dart Map and JSON

- A JSON object is essentially a set of key–value pairs.
- Dart's Map<String, dynamic> matches this structure perfectly:
    - String = JSON keys
    - dynamic = JSON values (which can be strings, numbers, bools, lists, nested maps, etc.)

2. toMap() and fromMap()

- toMap() converts your model object (Todo) into a Map → which can then be directly converted to JSON (e.g., jsonEncode(todo.toMap())).
- fromMap() reconstructs your model from that map → typically after decoding JSON (e.g., Todo.fromMap(jsonDecode(jsonString))).

```
Map<String, dynamic> toMap() {
  return {
    'id': id,
    'title': title,
    'description': description,
    'isCompleted': isCompleted,
    'createdAt': createdAt.toIso8601String(),
  };
}

factory Todo.fromMap(Map<String, dynamic> map) {
  return Todo(
    id: map['id'],
    title: map['title'],
    description: map['description'],
    isCompleted: map['isCompleted'] ?? false,
    createdAt: DateTime.parse(map['createdAt']),
  );
}
```

Other utility methods override.

```
@override
String toString() {
  return 'Todo(id: $id, title: $title, isCompleted: $isCompleted)';
}

@override
bool operator ==(Object other) {
  if (identical(this, other)) return true;
  return other is Todo && other.id == id;
}

@override
int get hashCode => id.hashCode;
```

Service/Utility Class

Role of Service

- **Handles data operations** (fetch, save, update, delete)
- Simulates a **repository or service layer**
- Can interact with:
  - Databases
  - REST APIs
  - Local storage

In MVVM

- **Service** = Responsible for data access & related business logic
- **ViewModel** = Uses the service to manage state for the View
- **View** = Displays the data, no direct DB/API knowledge

Benefits of Service Layer

- Hides the **connection** between Model ↔️ Database
- Promotes **separation of concerns**
- Easier to **test** (mock/fake services)
- Flexible: swap out DB/API without changing the View or ViewModel

```
class TodoService {
  // Simulated database using a list
  final List<Todo> _todos = [];

  /// Get all todos
  List<Todo> getAllTodos() {
    return List.unmodifiable(_todos);
  }
```

- List.unmodifiable(_todos) creates a read-only copy of the list.
- That means outside code can see the todos but cannot modify them.

CRUD

```
/// Add a new todo
void addTodo(Todo todo) {
  _todos.add(todo);
}

/// Update an existing todo
void updateTodo(Todo updatedTodo) {
  final index = _todos.indexWhere((todo) => todo.id == updatedTodo.id);
  if (index != -1) {
    _todos[index] = updatedTodo;
  }
}

/// Delete a todo by ID
void deleteTodo(String id) {
  _todos.removeWhere((todo) => todo.id == id);
}
```

Utility

```
/// Toggle todo completion status
void toggleTodo(String id) {
  final index = _todos.indexWhere((todo) => todo.id == id);
  if (index != -1) {
    final todo = _todos[index];
    _todos[index] = todo.copyWith(isCompleted: !todo.isCompleted);
  }
}
```

Search features:

```
  /// Get completed todos
  List<Todo> getCompletedTodos() {
    return _todos.where((todo) => todo.isCompleted).toList();
  }

  /// Get pending todos
  List<Todo> getPendingTodos() {
    return _todos.where((todo) => !todo.isCompleted).toList();
  }

  /// Search todos by title
  List<Todo> searchTodos(String query) {
    if (query.isEmpty) return getAllTodos();
    return _todos.where((todo) =>
      todo.title.toLowerCase().contains(query.toLowerCase()) ||
      todo.description.toLowerCase().contains(query.toLowerCase())
    ).toList();
  }
}
```

ViewModel

- The ViewModel has the business logic and its related data structure: `List<Todo>` to contain Todo objects.
- It aggregates (contains) the TodoService object (`_todoService`).

```
class TodoViewModel extends ChangeNotifier {
  final TodoService _todoService;

  // Private state variables
  List<Todo> _todos = [];
  bool _isLoading = false;
  String _searchQuery = '';
  TodoFilter _currentFilter = TodoFilter.all;
```

Constructor

- The constructor takes a parameter: this._todoService.
- The `this._` syntax means: assign the argument to a private field named _todoService.
- This is dependency injection → instead of creating a TodoService inside the ViewModel, you inject it from outside.
- Benefit: makes your code testable, flexible, and decoupled.

```
// Constructor - Dependency injection of the service
TodoViewModel(this._todoService) {
  _loadTodos();
}
```

TodoFilter & Extension

```
/// Enum for todo filters
enum TodoFilter {
  all,
  completed,
  pending,
}

/// Extension to get display names for filters
extension TodoFilterExtension on TodoFilter {
  String get displayName {
    switch (this) {
      case TodoFilter.all:
        return 'All';
      case TodoFilter.completed:
        return 'Completed';
      case TodoFilter.pending:
        return 'Pending';
    }
  }
}
```

Public getters & properties

```
// Public getters - Expose state to the View
List<Todo> get todos => _getFilteredTodos();
bool get isLoading => _isLoading;
String get searchQuery => _searchQuery;
TodoFilter get currentFilter => _currentFilter;

// Computed properties to count the Todos
int get totalTodos => _todos.length;
int get completedTodos =>
  _todos.where((todo) => todo.isCompleted).length;
int get pendingTodos =>
  _todos.where((todo) => !todo.isCompleted).length;
```

The _loadTodos() method simulates delayed loading of Todo items.

```
/// Load todos from service
  void _loadTodos() {
    _todos = _todoService.getAllTodos();
  }
```

CRUD operations

```
void addTodo(String title, String description) {
  if (title.trim().isEmpty) return;

  final newTodo = Todo(
    id: DateTime.now().millisecondsSinceEpoch.toString(),
    title: title.trim(),
    description: description.trim(),
  );
  _todoService.addTodo(newTodo);
  _todos = _todoService.getAllTodos();
  notifyListeners();
}
/// Update search query
void updateSearchQuery(String query) {
  _searchQuery = query;
  notifyListeners();
}
/// Delete a todo
void deleteTodo(String id) {
  _todoService.deleteTodo(id);
  _todos = _todoService.getAllTodos();
  notifyListeners();
}
```

The service function simulates DB layer, so in this code, we store the Todo data into DB and updates current _todo list.

```
    _todoService.addTodo(newTodo);
    _todos = _todoService.getAllTodos();
```

```
  /// Toggle todo completion status
  void toggleTodo(String id) {
    _todoService.toggleTodo(id);
    _todos = _todoService.getAllTodos();
    notifyListeners();
  }

  /// Clear search
  void clearSearch() {
    _searchQuery = '';
    notifyListeners();
  }
```

Filter functions

```
/// Set filter
void setFilter(TodoFilter filter) {
  _currentFilter = filter;
  notifyListeners();
}
```

Get filtered todos based on current filter and search query

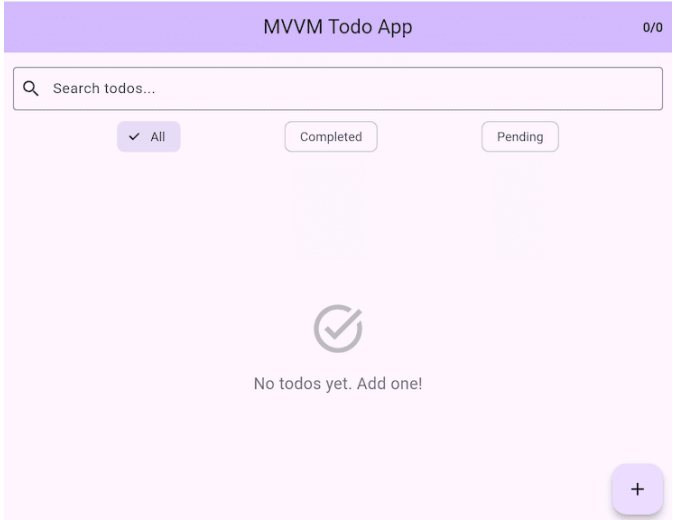
```
List<Todo> _getFilteredTodos() {
  List<Todo> filteredTodos;

  // Apply filter
  switch (_currentFilter) {
    case TodoFilter.completed:
      filteredTodos = _todos.where(
        (todo) => todo.isCompleted).toList();
      break;
    case TodoFilter.pending:
      filteredTodos = _todos.where(
        (todo) => !todo.isCompleted).toList();
      break;
    case TodoFilter.all:
      filteredTodos = _todos;
      break;
  }
  // Apply search
  if (_searchQuery.isNotEmpty) {
    filteredTodos = filteredTodos.where((todo) =>
      todo.title.toLowerCase().contains(_searchQuery.toLowerCase()) ||
      todo.description.toLowerCase().contains(_searchQuery.toLowerCase())
    ).toList();
  }

  return filteredTodos;
}
```

```
  /// Set loading state
  void _setLoading(bool loading) {
    _isLoading = loading;
    notifyListeners();
  }

  /// Clear all completed todos
  void clearCompleted() {
    final completedIds = _todos
        .where((todo) => todo.isCompleted)
        .map((todo) => todo.id)
        .toList();

    for (final id in completedIds) {
      _todoService.deleteTodo(id);
    }

    _todos = _todoService.getAllTodos();
    notifyListeners();
  }

  /// Mark all todos as completed
  void markAllCompleted() {
    for (final todo in _todos.where((todo) => !todo.isCompleted)) {
      _todoService.toggleTodo(todo.id);
    }
    _todos = _todoService.getAllTodos();
    notifyListeners();
  }
}
```

TodoList View

This screen demonstrates how Views should:

1. Handle user input
2. Validate data (UI-level validation)
3. Call ViewModel methods to perform actions
4. Not contain business logic



Title Text & Display Statistics

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('MVVM Todo App'),
        actions: [
          Consumer<TodoViewModel>( // Redrawn with a trigger
            builder: (context, viewModel, child) {
              return Text(
                '${viewModel.completedTodos}/${viewModel.totalTodos}',
              ),
            },
          ),
        ],
      ),
```

Notice that all the ViewModel data can be accessed using the `viewModel` object.

Search bar & Todo & + Button

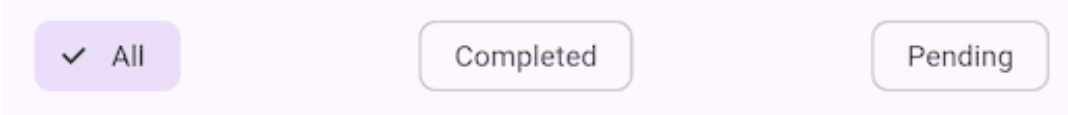
```
      body: Column(
        children: [
          _buildSearchAndFilter(),
          Expanded(child:_buildTodoList(),),
          _buildActionButtons(),
        ],
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: () => _navigateToAddTodo(context),
      ),
    );
  }
```

_buildSearchAndFilter

FilterChip

A FilterChip is like a toggle button with a label, used here so the user can switch between todo filters.

```
return FilterChip(
  label: Text(filter.displayName),                // What's shown on the chip
  selected: viewModel.currentFilter == filter,    // Is it selected?
  onSelected: (_) =>
      context.read<TodoViewModel>().setFilter(filter), // Action when tapped
);
```



Build search and filter widgets

```
Widget _buildSearchAndFilter() {
  return Consumer<TodoViewModel>(
    builder: (context, viewModel, child) {
      return Container(
        child: Column(
          children: [
            TextField(
              onChanged: context.read<TodoViewModel>().updateSearchQuery,
            ),
            Row(
              children: TodoFilter.values.map((filter) {
                return FilterChip(
                  selected: viewModel.currentFilter == filter,
                  onSelected: (_) => context.read<TodoViewModel>().setFilter(filter),
                );
              }).toList(),
            ),
          ],
        ),
      );
    },
  );
}
```

_buildTodoList

Build the todo list

```
Widget _buildTodoList() {
  return Consumer<TodoViewModel>(
    builder: (context, viewModel, child) {
      if (viewModel.isLoading) {
        return const Center(child: CircularProgressIndicator());
      }

      final todos = viewModel.todos;

      return ListView.builder(
        itemCount: todos.length,
        itemBuilder: (context, index) {
          final todo = todos[index];
          return _buildTodoItem(context, todo, viewModel);
        },
      );
    },
  );
}
```

_buildTodoItem

Build individual todo item

```
Widget _buildTodoItem(BuildContext context, Todo todo, TodoViewModel viewModel) {
  return Card(
    child: ListTile(
      leading: Checkbox(
        value: todo.isCompleted,
        onChanged: (_) => viewModel.toggleTodo(todo.id),
      ),
      title: Text(todo.title,),
      trailing: Row(
        children: [
          Text(_formatDate(todo.createdAt),),
          IconButton(
            onPressed: () => _showDeleteConfirmation(context, todo, viewModel),
          ),
        ],
      ),
    ),
  );
}
```

_buildActionButtons

Build action buttons

```
Widget _buildActionButtons() {
  return Consumer<TodoViewModel>(
    builder: (context, viewModel, child) {
      if (viewModel.totalTodos == 0) return const SizedBox.shrink();

      return Container(
        child: Row(
          children: [
            ElevatedButton.icon(
              onPressed: viewModel.pendingTodos > 0
                ? context.read<TodoViewModel>().markAllCompleted
                : null,
              label: const Text('Complete All'),
            ),
            ElevatedButton.icon(
              onPressed: viewModel.completedTodos > 0
                ? context.read<TodoViewModel>().markAllCompleted.clearCompleted
                : null,
              label: const Text('Clear Completed'),
            ),
          ],
        ),
      );
    },
  );
}
```

_navigateToAddTodo

Navigate to add todo screen

```
floatingActionButton: FloatingActionButton(
  onPressed: () => _navigateToAddTodo(context),
),

void _navigateToAddTodo(BuildContext context) {
  Navigator.of(context).push(
    MaterialPageRoute(
      builder: (context) => const AddTodoView(),
    ),
  );
}
```

_showDeleteConfirmation

Show delete confirmation dialog

- Notice that we should use `viewModel.deleteTodo(todo.id);`, not `context.read<TodoViewMode>().deleteTodo(todo.id)` because the context is Dialog context.

```
void _showDeleteConfirmation(BuildContext context, Todo todo, TodoViewModel viewModel) {
  showDialog(
    context: context,
    builder: (BuildContext context) {
      return AlertDialog(
        title: const Text('Delete Todo'),
        content: Text('Are you sure you want to delete "${todo.title}"?'),
        actions: [
          TextButton(
            onPressed: () => Navigator.of(context).pop(),
            child: const Text('Cancel'),
          ),
          TextButton(
            onPressed: () {
              viewModel.deleteTodo(todo.id);
              Navigator.of(context).pop();
            },
            child: const Text('Delete', style: TextStyle(color: Colors.red)),
          ),
        ]
```

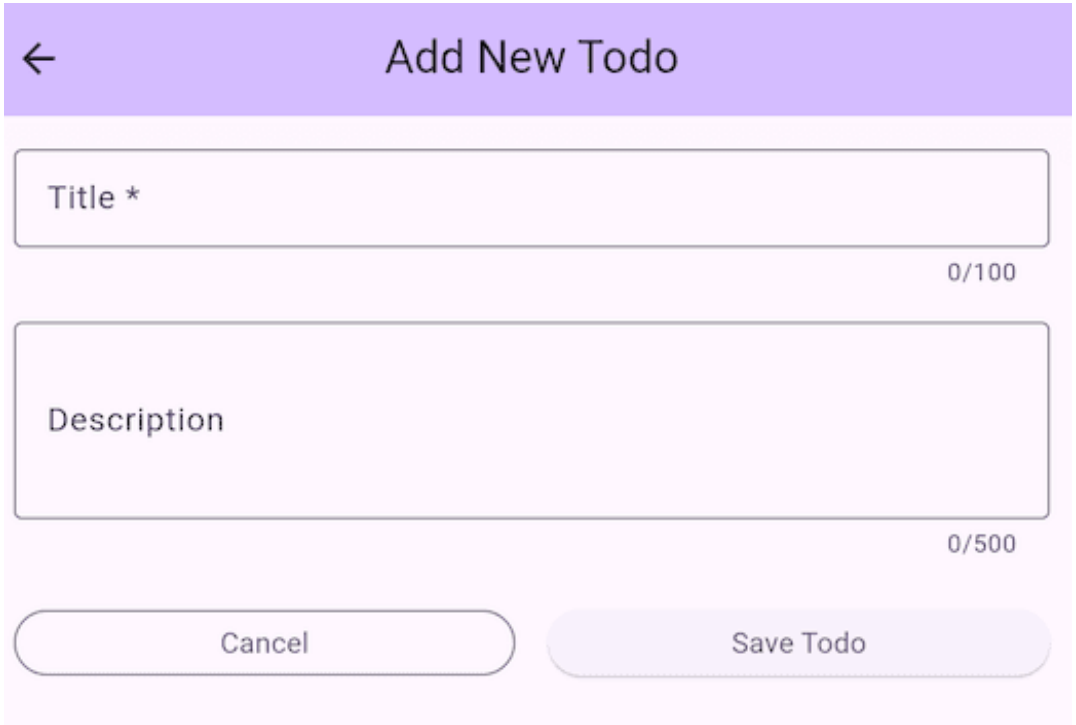
_formatDate

Format date for display

```
  String _formatDate(DateTime date) {
    final now = DateTime.now();
    final difference = now.difference(date);

    if (difference.inDays == 0) {
      return 'Today';
    } else if (difference.inDays == 1) {
      return 'Yesterday';
    } else if (difference.inDays < 7) {
      return '${difference.inDays} days ago';
    } else {
      return '${date.day}/${date.month}/${date.year}';
    }
  }
}
```

AddTodoView Stateful View

This widget is shown when users click the + button.



StatefuleWiget & Related State

```
class AddTodoView extends StatefulWidget {
  const AddTodoView({super.key});

  @override
  State<AddTodoView> createState() => _AddTodoViewState();
}

class _AddTodoViewState extends State<AddTodoView> {
  final _formKey = GlobalKey<FormState>();
  final _titleController = TextEditingController();
  final _descriptionController = TextEditingController();

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

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text('Add New Todo'),
    ),
    body: Padding(
      child: Form(
        child: Column(
          children: [
            TextFormField(controller: _titleController,),
            TextFormField(controller: _descriptionController,),
            Row(
              children: [
                Expanded(
                  child: OutlinedButton(
                    onPressed: () => Navigator.of(context).pop(),
                    child: const Text('Cancel'),
                  ),
                ),
                Expanded(
                  child: ElevatedButton(
                    onPressed: _saveTodo,
                    child: const Text('Save Todo'),
                  ),
                ),
              ],
            ),
          ],
        ),
      ),
    ),
  );
}
```

Save todo - demonstrates how View interacts with ViewModel

```
void _saveTodo() {
  if (_formKey.currentState?.validate() ?? false) {
    context.read<TodoViewModel>().addTodo(
      _titleController.text,
      _descriptionController.text,
    );

    // Show success message
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(
        content: Text('Todo added successfully!'),
        backgroundColor: Colors.green,
      ),
    );

    // Navigate back
    Navigator.of(context).pop();
  }
}
```

main.dart

Main application entry point: this file demonstrates MVVM setup with dependency injection:

1. Creates service instances
2. Creates ViewModel instances with injected services
3. Provides ViewModels to the widget tree using Provider
4. Keeps the UI and business logic separated

We used `ChangeNotifierProvider` in this way.

```
void main() {
  runApp(const MyApp());
}

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

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'State Management with ChangeNotifier',
      home: ChangeNotifierProvider(
        // Create the CounterModel instance
        create: (context) => CounterModel(),
        child: const CounterScreen(),
      ),
    );
  }
}
```

However, we use `TodoService` as an argument (dependency injection) to the TodoViewModel.

So, we need to use MultiProvider.

MultiProvider

- A Flutter widget from the Provider package.
- Lets you register multiple providers at once (instead of nesting them deeply).
- Provides these objects to the widget tree so any descendant can read or watch them.

```dart
import 'package:flutter/material.dart';
import 'package:provider/provider.dart';
import 'models/todo.dart';
import 'views/todo_list_screen.dart';
import 'viewmodels/todo_viewmodel.dart';
import 'services/todo_service.dart';

void main() {
  runApp(const MVVMTodoApp());
}

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

  @override
  Widget build(BuildContext context) {
    return MultiProvider(
      providers: [
        Provider<TodoService>(
          create: (context) => TodoService(),
        ),

        ChangeNotifierProvider<TodoViewModel>(
          create: (context) => TodoViewModel(
            Provider.of<TodoService>(context, listen: false),
          ),
        ),
      ],
      child: MaterialApp(
        title: 'MVVM Todo App',
        home: const TodoListView(),
      ),
    );
  }
}
```

Another way to use Provider:

```
void mainWithSampleData() {
  runApp(
    MultiProvider(
      providers: [
        Provider<TodoService>(create: (_) => TodoService()),
        ChangeNotifierProvider<TodoViewModel>(
          create: (context) => TodoViewModel(context.read<TodoService>()),
        ),
      ],
      child: const MaterialApp(
        title: 'MVVM Todo App',
        home: TodoListView(),
        debugShowCheckedModeBanner: false,
      ),
    ),
  );
}
```

In the case you need to add sample data, you can do it as follows:

```
create: (context) {
  final service = TodoService();
  _SampleDataInitializer.addSampleData(service);
  return service;
},

...

class _SampleDataInitializer {
  static void addSampleData(TodoService service) {
    service.addTodo(
      Todo(
        id: '1',
        title: 'Learn Flutter MVVM',
        description: 'Understand the MVVM pattern',
      ),
    );
    ...
  }
}
```