Task Tracker Technical Guide

Table of Contents

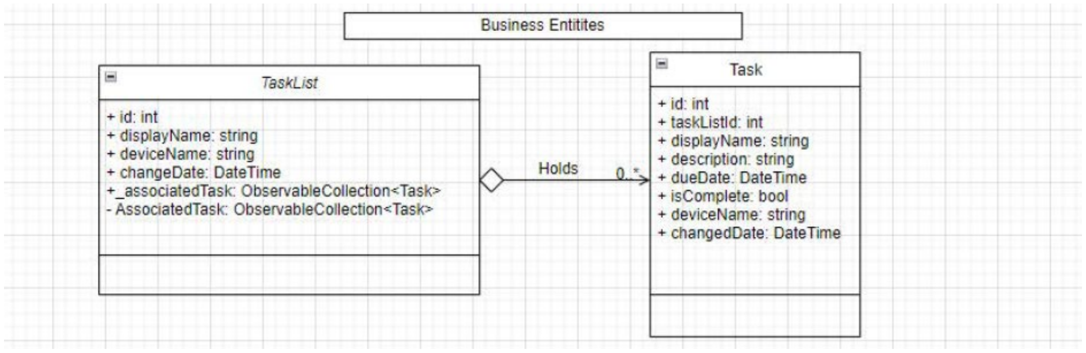
Class Diagram

Business Entities:

The primary entities managed by the application's database are TaskList and Task. These two objects are interconnected, with each Task being associated with a TaskList through a foreign key. This relationship is defined as follows:

1. **TaskList**: Represents a collection or a group of tasks. A TaskList can contain zero or many Task objects. However, it does not maintain direct awareness or reference to the individual Task objects it encompasses.
2. **Task**: Each Task is a discrete item that belongs to a single TaskList. It maintains a reference to its parent TaskList through a foreign key. This design ensures that every Task is explicitly associated with one, and only one, TaskList.

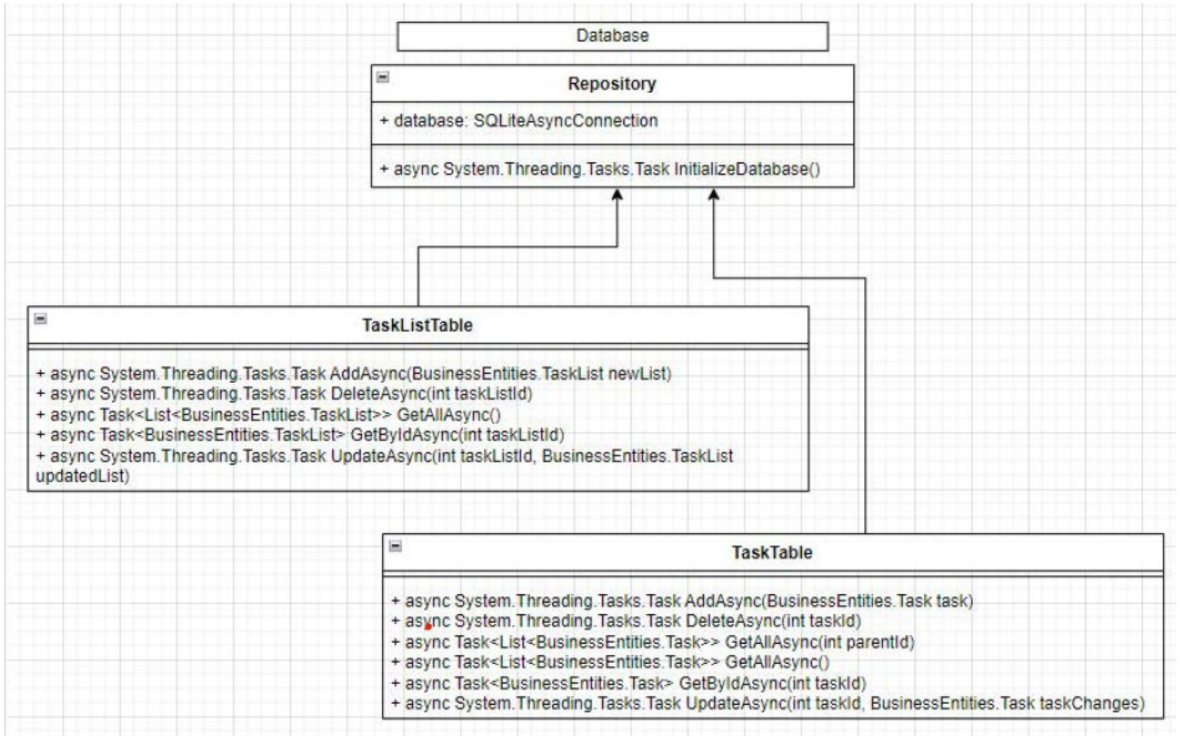
This structure allows for efficient organization and retrieval of tasks, where each TaskList serves as a container for multiple Task objects, and each Task clearly identifies the TaskList it belongs to.



Database:

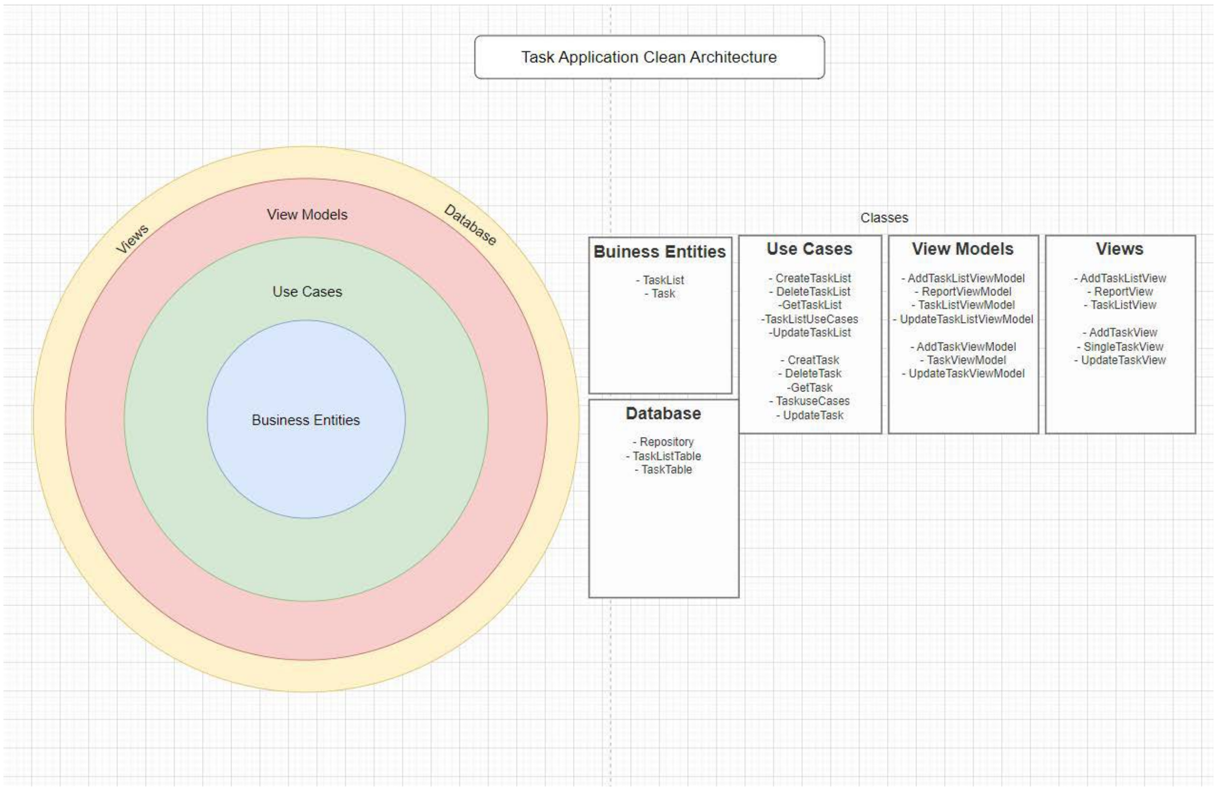
The application leverages a SQLite database that comprises two primary tables: TaskList and Task. Both of these tables inherit from a common Repository class. This inheritance strategy offers several advantages:

1. **Shared Database Connection**: By inheriting from the Repository class, both TaskList and Task tables utilize the same database connection. This shared connection is established in the constructor of the Repository class, ensuring a consistent and centralized management of the database connection.
2. **Database Initialization**: The Repository class provides an InitializeDatabase function, which is accessible to both TaskList and Task. This function is responsible for setting up the database schema, including the creation and configuration of the TaskList and Task tables. It ensures that the database is correctly initialized and ready for use when the application starts.
3. **Efficient Resource Management**: By centralizing the database connection and initialization logic in the Repository class, the application avoids redundancy and streamlines its interactions with the SQLite database. This approach promotes code reusability and simplifies maintenance.


Database
Repository
+ database: SQLiteAsyncConnection
+ async System.Threading.Tasks.Task InitializeDatabase()
TaskListTable
+ async System.Threading.Tasks.Task AddAsync(BusinessEntities.TaskList newList)
+ async System.Threading.Tasks.Task DeleteAsync(int taskListId)
+ async Task<List<BusinessEntities.TaskList>> GetAllAsync()
+ async Task<BusinessEntities.TaskList> GetByIdAsync(int taskListId)
+ async System.Threading.Tasks.Task UpdateAsync(int taskListId, BusinessEntities.TaskList updatedList)
TaskTable
+ async System.Threading.Tasks.Task AddAsync(BusinessEntities.Task task)
+ async System.Threading.Tasks.Task DeleteAsync(int taskId)
+ async Task<List<BusinessEntities.Task>> GetAllAsync(int parentId)
+ async Task<List<BusinessEntities.Task>> GetAllAsync()
+ async Task<BusinessEntities.Task> GetByIdAsync(int taskId)
+ async System.Threading.Tasks.Task UpdateAsync(int taskId, BusinessEntities.Task taskChanges)

Application Clean Architecture

Incorporating clean architecture into the application, separated into four distinct projects - Database, Use Cases, View Models, and Views - brings a host of significant benefits. This modular approach enhances maintainability, allowing for easier updates and modifications to individual components without impacting the overall system. Each isolated layer improves the testability of the application, enabling focused and efficient unit testing. Flexibility is another key advantage; changes in one module, such as the database or user interface, can be implemented with minimal effect on other parts of the application. This architecture also facilitates parallel development, as different teams can concurrently work on separate layers, thus accelerating the development process. Furthermore, the clear separation of concerns makes the application more intuitive and easier to navigate, particularly beneficial for new developers joining the project. Lastly, centralizing business logic in the Use Cases layer ensures consistent and reliable application behavior, adhering to defined business rules and logic.



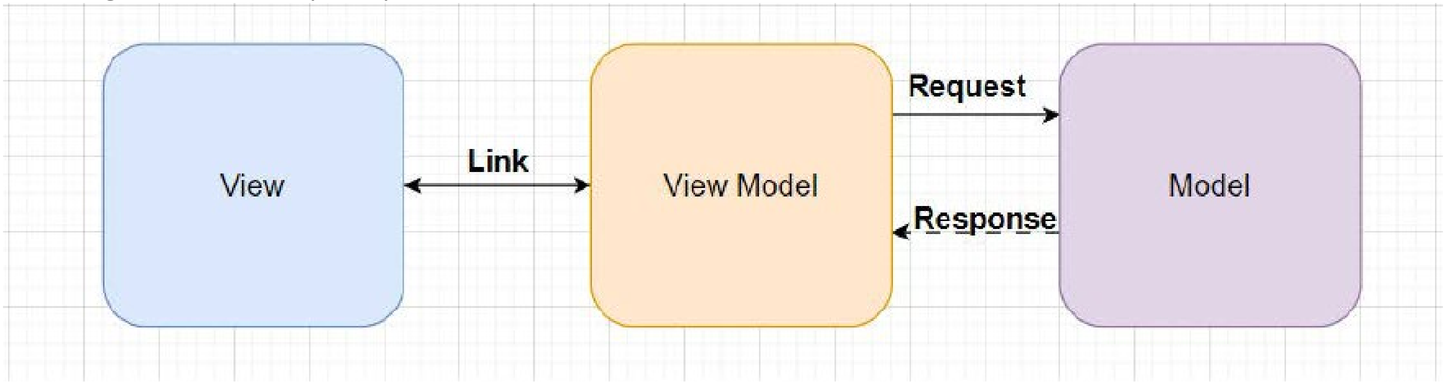
MVVM Approach

The Task Tracker Application implements the Model-View-View Model (MVVM) design pattern, leveraging the capabilities of the Community. Toolkit within the .NET Maui framework. This integration facilitates an efficient architecture, where the 'Use Cases' and 'Database' projects collectively serve as the Model. These components are responsible for directly managing and controlling data.

In this architecture, .NET Maui's built-in dependency injection plays a pivotal role. It seamlessly connects the Model components with other. parts of the application, ensuring a smooth flow of data and functionality.

The View Model layer is embodied by the View Model classes within the Task Tracker Project. These classes are crucial in abstracting and handling the communication between the Model and the Views. This abstraction is vital for maintaining a clean separation of concerns, a core principle of the MVVM pattern.

In the context of this application, the Views are represented by the app content pages. They are designed to interact with the View Model, which in turn, deals with the Model. This setup not only enhances maintainability and testability but also ensures that the UI layer – the Views - remains free from business logic, adhering to the MVVM principles.



Unit Testing

Scope:

The primary objective of this testing plan is to validate the functionality and reliability of all CRUD (Create, Read, Update, Delete) operations performed on the database. This encompasses ensuring that these operations provide accurate, consistent, and reliable data manipulation. The plan aims to cover various scenarios, including normal operations, edge cases, and error handling, to comprehensively assess the robustness of the database interactions.

Implementation:

The testing will be conducted using the xUnit testing framework. To simulate the database environment without affecting the actual SQLite database, the project leverages an in-memory database. This approach facilitates testing in a controlled environment, allowing for the isolation of database operations. The in-memory database is designed to replicate the behavior of the actual database, enabling a thorough evaluation of CRUD functionalities. The tests will be executed to assess the correctness of the database operations, with particular attention to the consistency of the data returned and the system's response to various input conditions. The test results will provide insights into the performance and reliability of the database operations, forming the basis for any necessary refinements or optimizations.

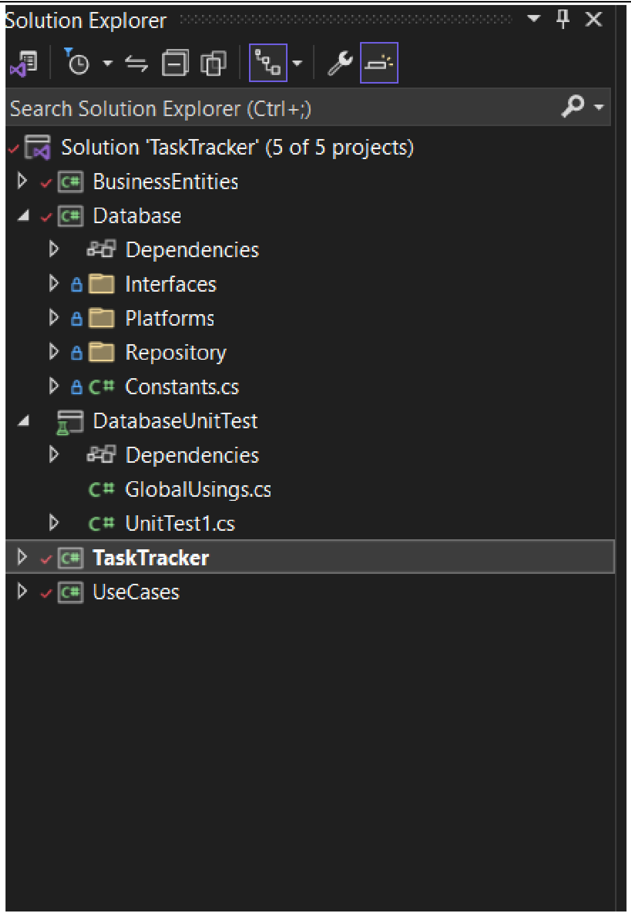
Design:

The application architecture is strategically divided into several modular projects, as illustrated in the accompanying screenshot *(Design Img 1)*. This modularization and loose coupling facilitate efficient and isolated testing of individual components, particularly the database. This design allows for unit testing of the database functionality without the need to modify any other part of the codebase.

A key aspect of this architecture is the use of interfaces and dependency injection, which significantly enhances the flexibility of the database component. By defining interfaces such as ITaskListTable and ITaskTable, the system sets clear contracts for database interactions. This abstraction ensures that other components of the application are not tightly bound to a specific database implementation but rather to these interfaces.

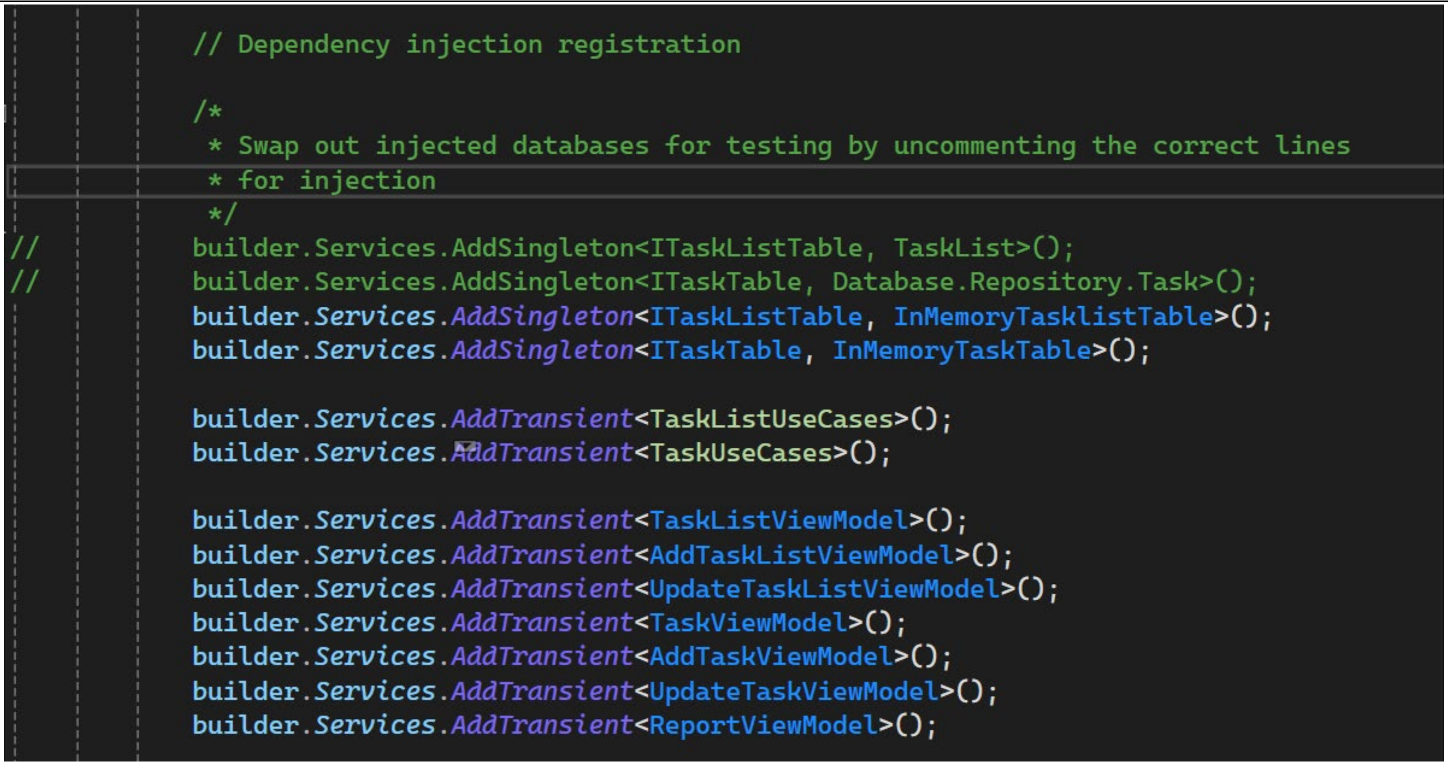
For testing purposes, this design allows for the seamless substitution of the actual database classes with mock or test database classes that implement these interfaces. By simply changing the dependency injection configuration in the .NET MAUI project, the application can switch between using the real database and a test database. This swap is demonstrated in the second screenshot provided *(Design Img 2)*.

This approach not only streamlines the testing process but also contributes to the overall maintainability and scalability of the application. It ensures that any changes or upgrades to the database layer can be implemented with minimal impact on the rest of the application, provided that the contract defined by the interfaces is maintained.



```
Solution 'TaskTracker' (5 of 5 projects)
  BusinessEntities
  Database
    Dependencies
    Interfaces
    Platforms
    Repository
    Constants.cs
  DatabaseUnitTest
    Dependencies
    GlobalUsings.cs
    UnitTest1.cs
  TaskTracker
  UseCases
```

1: Design Img 1



```
        // Dependency injection registration

        /*
         * Swap out injected databases for testing by uncommenting the correct lines
         * for injection
         */
//      builder.Services.AddSingleton<ITaskListTable, TaskList>();
//      builder.Services.AddSingleton<ITaskTable, Database.Repository.Task>();
        builder.Services.AddSingleton<ITaskListTable, InMemoryTasklistTable>();
        builder.Services.AddSingleton<ITaskTable, InMemoryTaskTable>();

        builder.Services.AddTransient<TaskListUseCases>();
        builder.Services.AddTransient<TaskUseCases>();

        builder.Services.AddTransient<TaskListViewModel>();
        builder.Services.AddTransient<AddTaskListViewModel>();
        builder.Services.AddTransient<UpdateTaskListViewModel>();
        builder.Services.AddTransient<TaskViewModel>();
        builder.Services.AddTransient<AddTaskViewModel>();
        builder.Services.AddTransient<UpdateTaskViewModel>();
        builder.Services.AddTransient<ReportViewModel>();
```

2: Design Img 2

Test Scripts

The testing scripts are written in C# using xUnit and can be found in the source code.

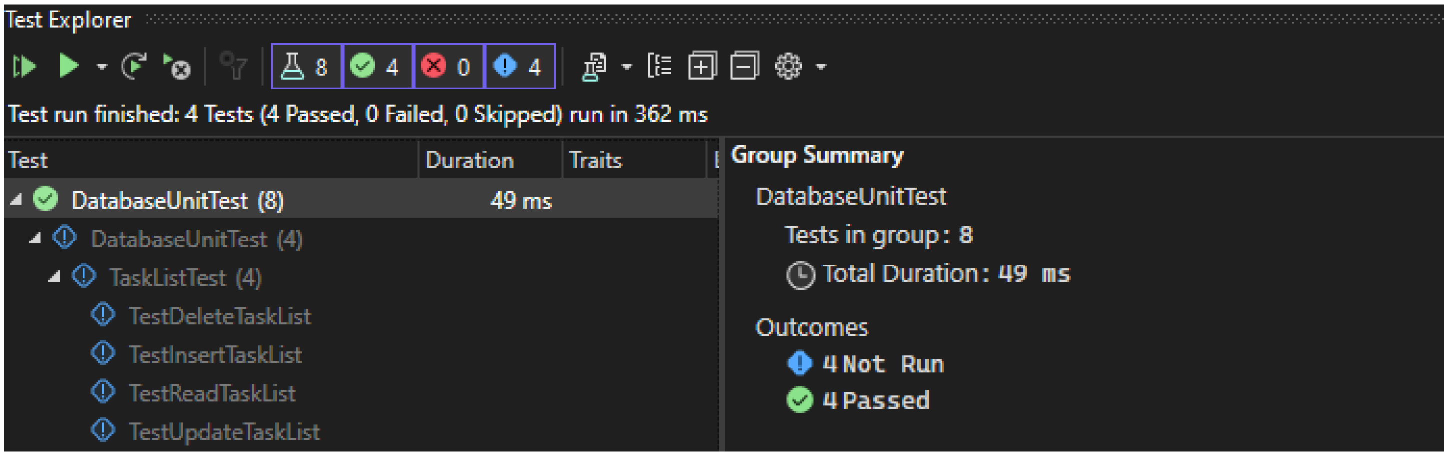
All Scripts test a CRUD feature of the database, as an example creating is being tested below.

```
        [Fact]
        public async Task TestInsertTaskList()
        {
            var newList = new BusinessEntities.TaskList("AddedList");

            await _table.AddAsync(newList);

            var retrievedList = await _table.GetByIdAsync(newList.id);
            Assert.NotNull(retrievedList);
            Assert.Equal("AddedList", retrievedList.displayName);
        }
```

Test Results:


Test Explorer

8 4 0 4

Test run finished: 4 Tests (4 Passed, 0 Failed, 0 Skipped) run in 362 ms

Test	Duration	Traits
DatabaseUnitTest (8)	49 ms	
DatabaseUnitTest (4)		
TaskListTest (4)		
TestDeleteTaskList		
TestInsertTaskList		
TestReadTaskList		
TestUpdateTaskList		

Group Summary
DatabaseUnitTest
Tests in group: 8
Total Duration: 49 ms
Outcomes
4 Not Run
4 Passed

Changes: All the unit test passes and signified the application is working as expected. Since that is the case, there were no changes made.

Panopto Link: [D424 TASK 3: DEVELOPMENT AND TESTING (panopto.com)]()