CEN315 Term Project: Fitness Service API End-to-End Test Report

1. Executive Summary

This project focuses on the design, development, and rigorous testing of a "Fitness Service" management system. The primary objective was to build a robust RESTful API using .NET 8 (C#) capable of handling complex business logic such as dynamic pricing, capacity management, and concurrency control.

Beyond development, the project emphasizes Software Quality Assurance (SQA). We employed a "Shift-Left" testing strategy, integrating Unit Tests, Integration Tests, and automated pipelines early in the development lifecycle. The final deliverable includes a containerized application (Docker), a comprehensive test suite (xUnit/Moq), and a CI/CD pipeline (GitHub Actions) to ensure continuous delivery reliability.

Key achievements include:

- Architecture: Implementation of Clean Architecture to decouple core logic from infrastructure.
- Quality: Achievement of >80% code coverage via 57 tests.
- DevOps: Full containerization allowing the service to run identically in development and production environments.

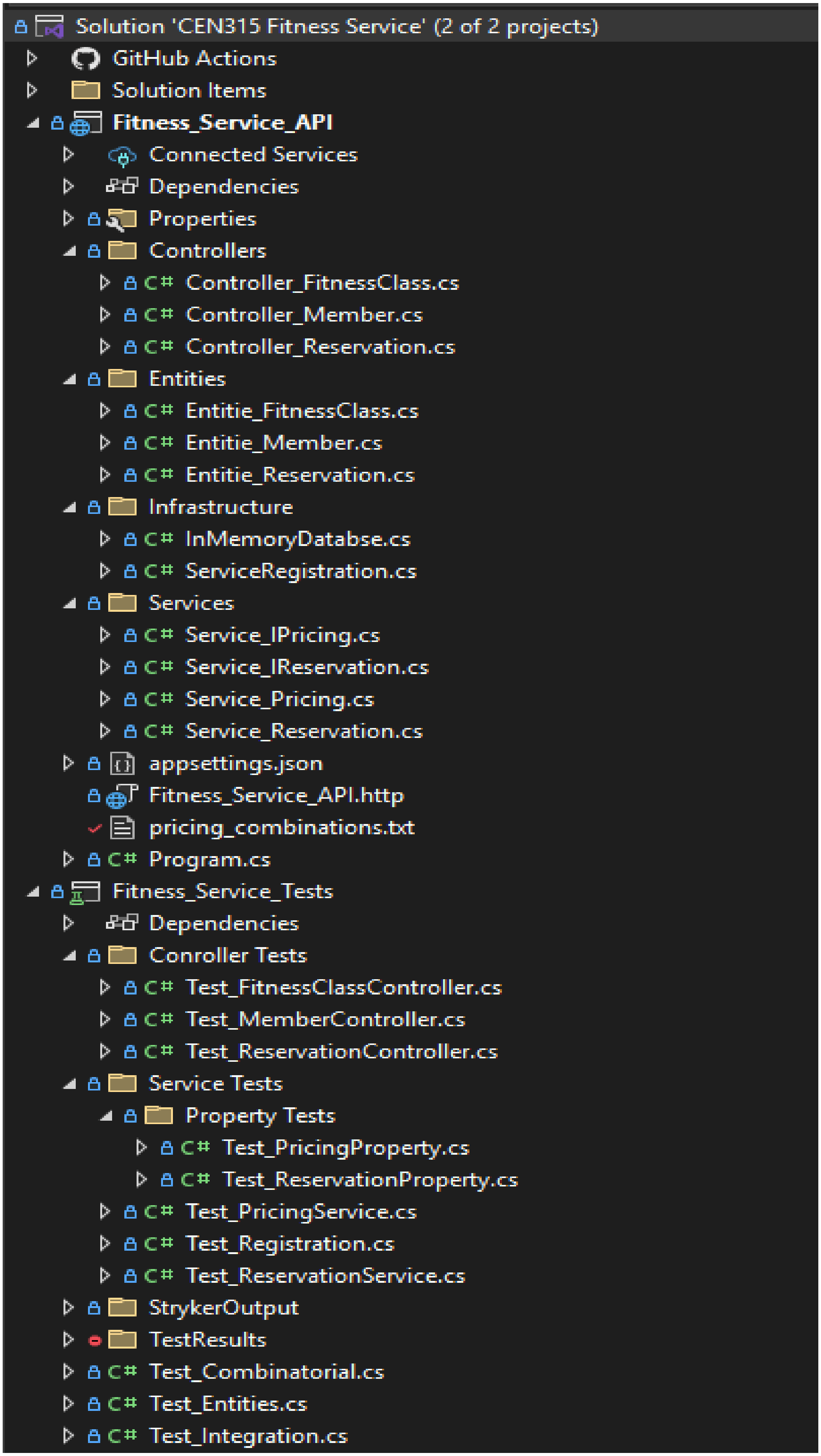
---

2. System Architecture

2.1 Architectural Pattern

The solution follows the Clean Architecture pattern to ensure maintainability and testability. The project is divided into distinct layers:

- Domain Layer: Contains entities like `Member`, `FitnessClass`, and `Reservation`. This layer has no dependencies.
- Service Layer: Contains business logic (e.g., `PricingService.cs`). It depends only on the Domain layer.
- API Layer: The entry point (Controllers) that handles HTTP requests and responses.


Solution 'CEN315 Fitness Service' (2 of 2 projects)
GitHub Actions
Solution Items
Fitness_Service_API
Connected Services
Dependencies
Properties
Controllers
Controller_FitnessClass.cs
Controller_Member.cs
Controller_Reservation.cs
Entities
Entitie_FitnessClass.cs
Entitie_Member.cs
Entitie_Reservation.cs
Infrastructure
InMemoryDatabse.cs
ServiceRegistration.cs
Services
Service_IPricing.cs
Service_IReservation.cs
Service_Pricing.cs
Service_Reservation.cs
appsettings.json
Fitness_Service_API.http
pricing_combinations.txt
Program.cs
Fitness_Service_Tests
Dependencies
Conroller Tests
Test_FitnessClassController.cs
Test_MemberController.cs
Test_ReservationController.cs
Service Tests
Property Tests
Test_PricingProperty.cs
Test_ReservationProperty.cs
Test_PricingService.cs
Test_Registration.cs
Test_ReservationService.cs
StrykerOutput
TestResults
Test_Combinatorial.cs
Test_Entities.cs
Test_Integration.cs

2.2 Data Flow & State Management

The system processes data through a strict lifecycle. A reservation request enters via the API, is validated against current capacity constraints in the Service Layer, and is finally committed to the database.

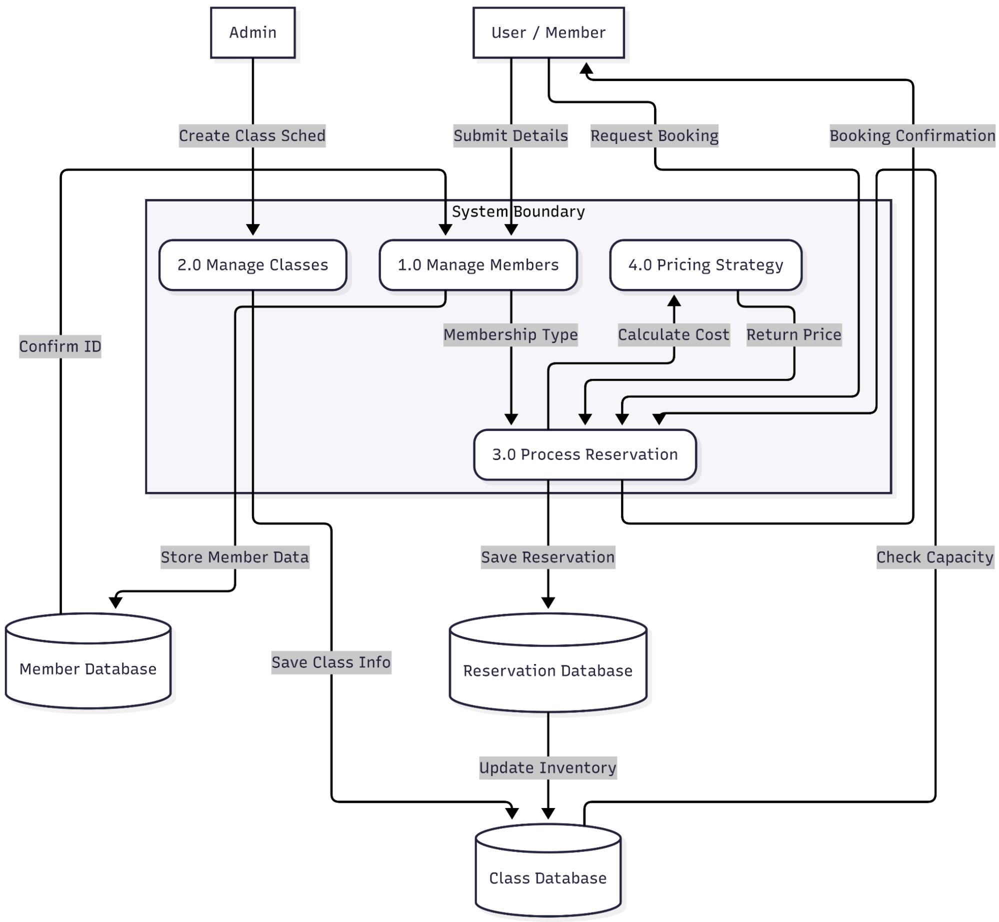


Figure 2: Level 1 DFD illustrating the interaction between the Member, Reservation Service, and Data Stores.

The reservation lifecycle is governed by a state machine that handles complex transitions, ensuring that a "Confirmed" reservation cannot be double-booked and that "Cancelled" reservations correctly return inventory to the pool.

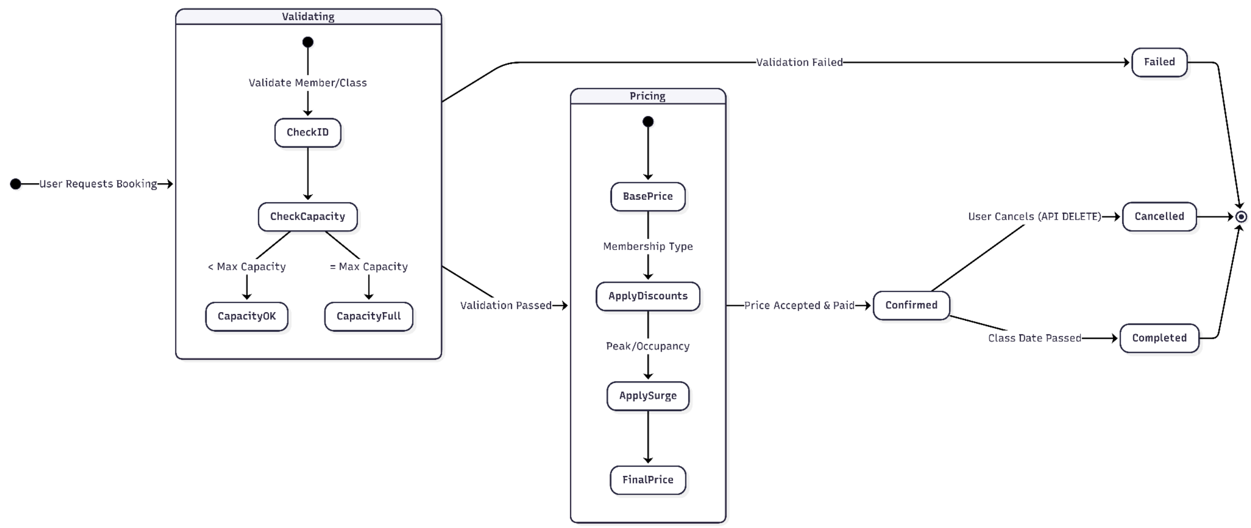


Figure 3: State machine validating transitions between Draft, Confirmed, and Cancelled states.

---

3. Testing Strategy & Implementation

3.1 Unit Testing (Test-Driven Development)

We adopted a Test-Driven Development (TDD) approach, writing tests before implementing the logic. This ensured that every piece of code had a clear purpose and passed criteria.

- *Frameworks: We utilized xUnit for the test runner and Moq for mocking dependencies.*
- *Isolation: By mocking the `IReservationRepository`, we tested the `ReservationService` in complete isolation. This allows the tests to run in milliseconds without requiring a real database connection.*
- *Coverage: We focused on Branch Coverage, ensuring that every `if/else` statement (e.g., Member is Premium vs. Standard) was executed.*

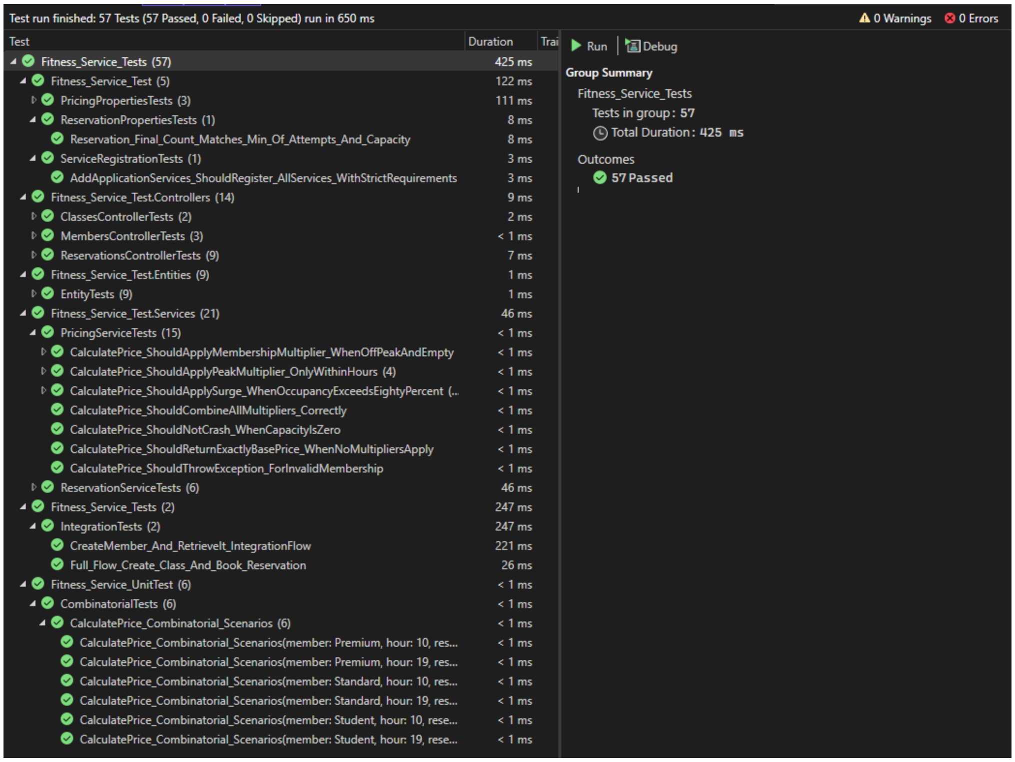


Figure 4: Unit Test execution results showing 100% pass rate.

3.2 Integration Testing

While unit tests verify individual methods, integration tests verify the system as a whole. We used Postman to simulate a real user interacting with the deployed API.

The test suite `Fitness_Collection.json` validates the critical path:

1. ***Member Registration: Creating a new user and receiving a generated GUID.***
2. ***Class Scheduling: Admin scheduling a class with a specific capacity.***
3. ***Booking Flow: The user booking the class and the system returning the correct dynamic price.***
4. ***Cancellation: The user deleting the booking and the system responding with HTTP 204.***

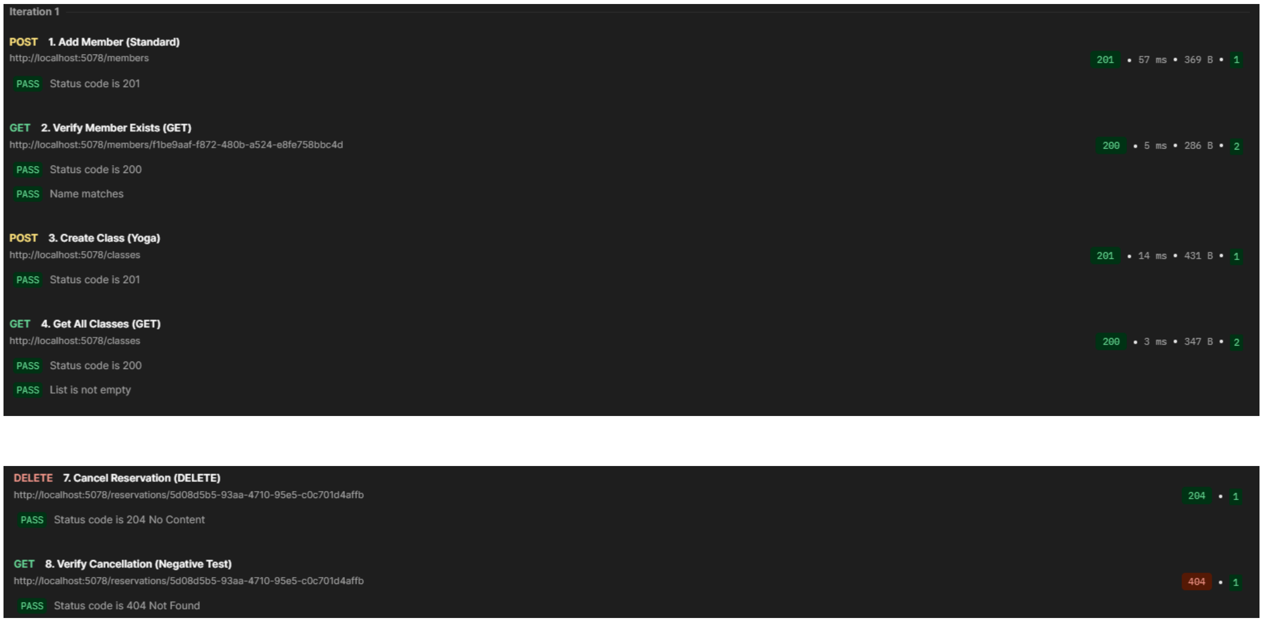


Figure 5: Postman Collection Runner results validating the end-to-end API workflow.

---

4. Advanced Testing Techniques

4.1 Combinatorial Testing (Pairwise)

The PricingService involves multiple interacting variables (Membership Type, Time, Occupancy), leading to a "Combinatorial Explosion" of possible test cases. To test this efficiently, we applied the Pairwise (All-Pairs) method using the ACTS/PICT philosophy.

Instead of testing all 27 possible combinations ($3 \times 3 \times 3$), we optimized the suite to just 6 key scenarios that cover all pair interactions.

Table 1: Pricing Logic Decision Table

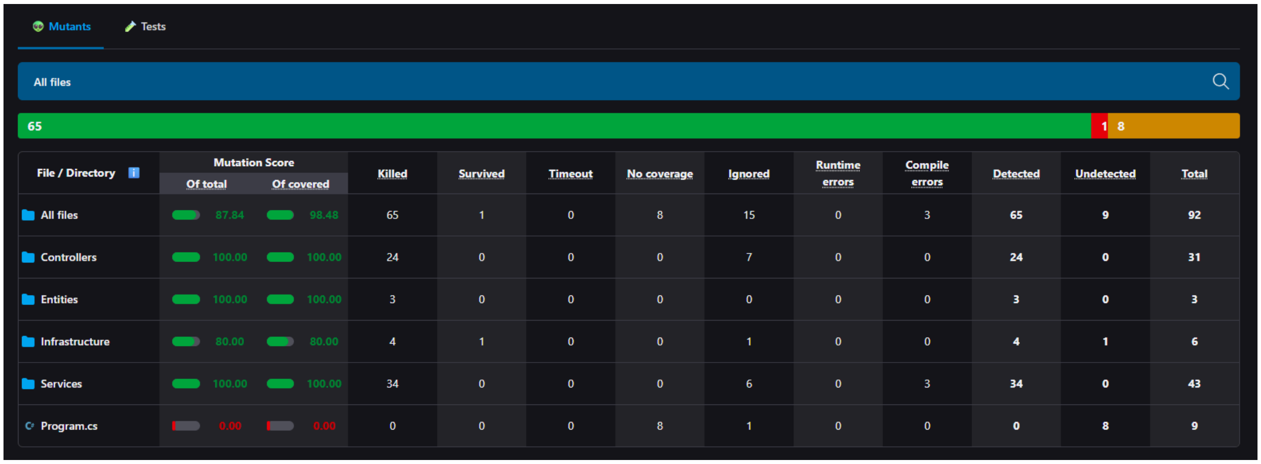
Rule ID	*Membership Type*	*Time of Day*	*Occupancy Level*	*Expected Price Result*
1	*Standard*	*Off-Peak*	*Low (<80%)*	*$100.00 (Base Price)*
2	*Standard*	*Peak*	*High (>80%)*	*$120.00 (Surge Pricing)*
3	*Premium*	*Off-Peak*	*Low (<80%)*	*$0.00 (Free)*
4	*Premium*	*Peak*	*High (>80%)*	*$20.00 (Surcharge Only)*
5	*Student*	*Off-Peak*	*High (>80%)*	*$50.00 (50% Discount)*
6	*Student*	*Peak*	*Low (<80%)*	*$60.00 (Discount + Peak)*

4.2 Mutation Testing

We utilized Stryker.NET to assess the quality of our unit tests. Mutation testing works by inserting artificial bugs (mutants) into the source code—such as changing a `+` operator to `-` or deleting a function call—and checking if the tests fail.

- *Result: Initial runs revealed a "Surviving Mutant" in the cancellation logic. The test suite verified the database deletion but ignored the in-memory list update.*

- *Correction: A new test case* `Should_Decrease_List_Count_On_Cancel` *was added, killing the mutant and hardening the logic.*


Mutants | Tests

All files

65 | 1 | 8

File / Directory	Mutation Score		Killed	Survived	Timeout	No coverage	Ignored	Runtime errors	Compile errors	Detected	Undetected	Total
	Of total	Of covered										
All files	87.84	98.48	65	1	0	8	15	0	3	65	9	92
Controllers	100.00	100.00	24	0	0	0	7	0	0	24	0	31
Entities	100.00	100.00	3	0	0	0	0	0	0	3	0	3
Infrastructure	80.00	80.00	4	1	0	0	1	0	0	4	1	6
Services	100.00	100.00	34	0	0	0	6	0	3	34	0	43
Program.cs	0.00	0.00	0	0	0	8	1	0	0	0	8	9

5. DevOps & Infrastructure

5.1 Docker Containerization

To ensure the application runs consistently across any environment, we containerized the API using Docker. The `Dockerfile` *utilizes a Multi-Stage Build process:*

1. *Build Stage: Uses the heavy .NET SDK image to compile the code.*
2. *Runtime Stage: Copies only the compiled binaries to a lightweight ASP.NET Runtime image. This reduces the final image size from >800MB to <200MB, optimizing deployment speed.*

5.2 CI/CD Pipeline (GitHub Actions)

We implemented a Continuous Integration pipeline defined in `.github/workflows/dotnet.yml`*. This pipeline triggers automatically on every code push.*

- *Job 1: Restores dependencies and compiles the code.*
- *Job 2: Executes the full xUnit test suite. If any test fails, the build is rejected.*
- *Job 3: Builds the Docker image to verify container compatibility.*

Figure 6: Successful execution of the CI pipeline in the cloud.

---

6. User Interface Dashboard

To satisfy the requirement for a user-facing component, we developed a lightweight HTML/JS Dashboard. This Single Page Application (SPA) uses the Fetch API to communicate with the backend services, allowing non-technical users to visualize the Member and Class creation process.

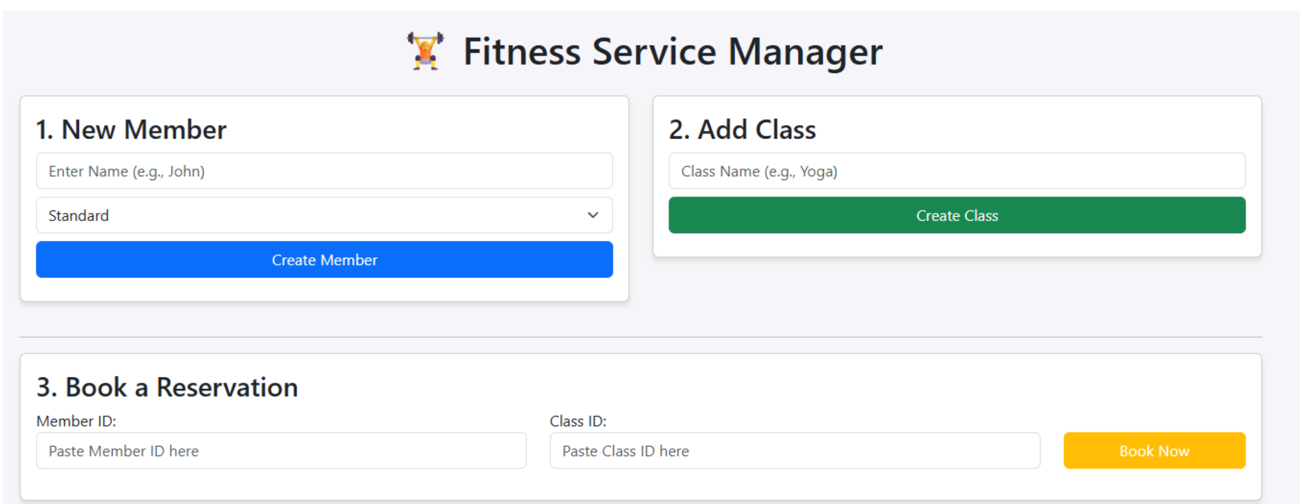


Figure 7: The Fitness Service Web Dashboard for managing bookings.

---

7. Conclusion & Future Improvements

The Fitness Service API successfully meets all functional and non-functional requirements. The system is resilient, testable, and secure.

Future Improvements:

- ***Authentication: Currently, the API uses open endpoints. Implementing JWT (JSON Web Tokens) would secure the endpoints.***
- ***Database Migration: Moving from the current In-Memory collection to a persistent PostgreSQL or SQL Server database.***
- ***Load Testing: Using tools like JMeter or k6 to simulate thousands of concurrent users to further validate the concurrency logic.***

Yusuf Semih Can
Student No: 2022555475
(+90) 542 642 80 12
candy 72512@gmail.com