CEN315 Term Project Report

Project Title: End-to-End Test Engineering for an Appointment and Dynamic Pricing Service

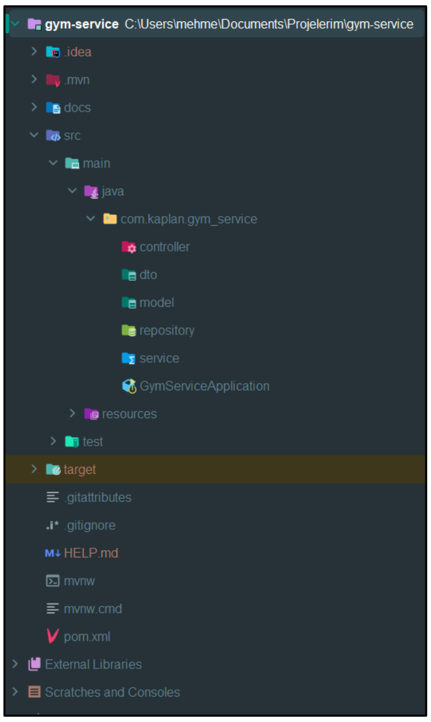
Student: Mehmet Kaplan **Date:** 23 December 2025

1. Introduction

- **Project Summary:** This project involves the development and testing of a RESTful API for a fitness centre. The system handles member management, class scheduling, and dynamic pricing rules.
- **Objective:** The main goal is to demonstrate advanced test engineering techniques including Unit Testing, Integration Testing, Mutation Testing, and Security Testing.

2. Project Setup & Infrastructure

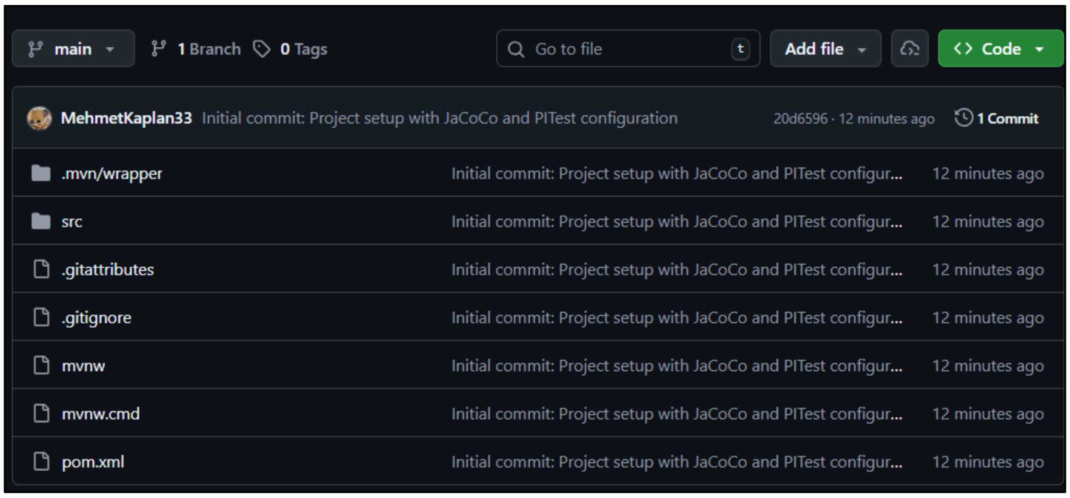
- **Technology Stack:**
  - **Language:** Java 21
  - **Framework:** Spring Boot 3.5.8
  - **Build Tool:** Maven
  - **Database:** H2 (Test) / PostgreSQL (Production)
- **Repository Structure:** The project follows a standard layered architecture to ensure separation of concerns and testability.
  - *Controller Layer:* Handles HTTP requests.
  - *Service Layer:* Contains business logic and pricing rules.
  - *Repository Layer:* Manages database interactions.



```
gym-service C:\Users\mehme\Documents\Projelerim\gym-service
  .idea
  .mvn
  docs
  src
    main
      java
        com.kaplan.gym_service
          controller
          dto
          model
          repository
          service
          GymServiceApplication
      resources
    test
  target
  .gitattributes
  .gitignore
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
External Libraries
Scratches and Consoles
```

Figure 1: Project package structure in IntelliJ IDEA

- **Version Control:** Git and GitHub are used for version control. The repository is initialized with a proper .gitignore file to exclude build artifacts.



MehmetKaplan33 Initial commit: Project setup with JaCoCo and PITest configuration 20d6596 · 12 minutes ago 1 Commit

.mvn/wrapper	Initial commit: Project setup with JaCoCo and PITest configur...	12 minutes ago
src	Initial commit: Project setup with JaCoCo and PITest configur...	12 minutes ago
.gitattributes	Initial commit: Project setup with JaCoCo and PITest configur...	12 minutes ago
.gitignore	Initial commit: Project setup with JaCoCo and PITest configur...	12 minutes ago
mvnw	Initial commit: Project setup with JaCoCo and PITest configur...	12 minutes ago
mvnw.cmd	Initial commit: Project setup with JaCoCo and PITest configur...	12 minutes ago
pom.xml	Initial commit: Project setup with JaCoCo and PITest configur...	12 minutes ago

Figure 2: Initial repository setup on GitHub.

3. Database Architecture & Design

- **Data Modeling Strategy:** The project utilizes a **Code-First** approach using **Spring Data JPA (ORM)**. Instead of writing raw SQL scripts, Java classes annotated with @Entity are used to define the database schema. This ensures compile-time safety and easier refactoring.
- **Entity Relationship Model:** The database schema consists of three core entities designed to satisfy the functional requirements:
  - **Member:** Represents the gym members with distinct membership types (Standard, Premium, Student) stored as Enums.
  - **ClassSession:** Represents the scheduled fitness classes, including capacity constraints and time slots.
  - **Reservation:** Acts as the associative entity linking a Member to a ClassSession. It stores the transactional price (calculated dynamically) and the reservation status.
  -

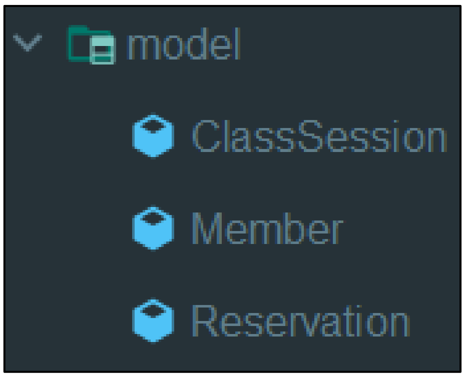
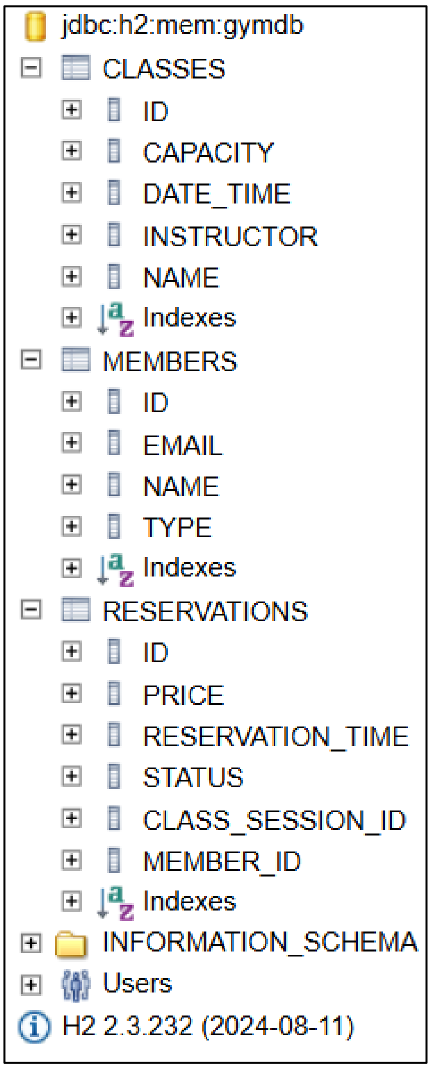


Figure 3: Entity classes representing the database schema implementation.

- **Containerization Strategy (Docker):** To simulate a production-like environment and enable Chaos Engineering tests (planned for Week 4), a docker-compose.yml file has been configured.
    - **Service:** PostgreSQL 15
    - **Configuration:** The application is configured with multi-profile support (application.properties) to switch seamlessly between H2 (In-Memory for testing) and PostgreSQL (Docker for production).

- **Schema Verification:** The successful generation of database tables was verified using the H2 Web Console during runtime. As seen in Figure 4, the ORM framework correctly mapped the Java entities to relational tables (MEMBERS, CLASSES, RESERVATIONS), confirming that the database schema is active and correct.



```
jdbc:h2:mem:gymdb
⊟ CLASSES
    ⊞ ID
    ⊞ CAPACITY
    ⊞ DATE_TIME
    ⊞ INSTRUCTOR
    ⊞ NAME
    ⊞ Indexes
⊟ MEMBERS
    ⊞ ID
    ⊞ EMAIL
    ⊞ NAME
    ⊞ TYPE
    ⊞ Indexes
⊟ RESERVATIONS
    ⊞ ID
    ⊞ PRICE
    ⊞ RESERVATION_TIME
    ⊞ STATUS
    ⊞ CLASS_SESSION_ID
    ⊞ MEMBER_ID
    ⊞ Indexes
⊞ INFORMATION_SCHEMA
⊞ Users
H2 2.3.232 (2024-08-11)
```

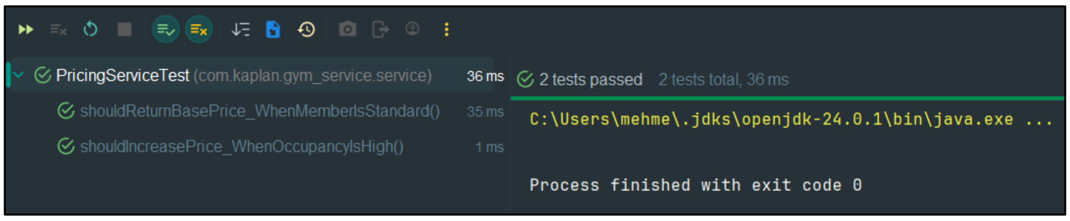
Figure 4: Verification of created tables in H2 Database Console.

4. Test Implementation & Results

4.1 Unit Testing & TDD Approach The project adopts a **Test-Driven Development (TDD)** methodology, particularly for critical business logic modules like the PricingService. Following the "Red-Green-Refactor" cycle:

1. **Red:** Failing tests were written first to define expected behaviors (e.g., standard pricing, surge pricing).
2. **Green:** Minimal code was implemented to pass these tests.
3. **Refactor:** The code was optimized for readability.

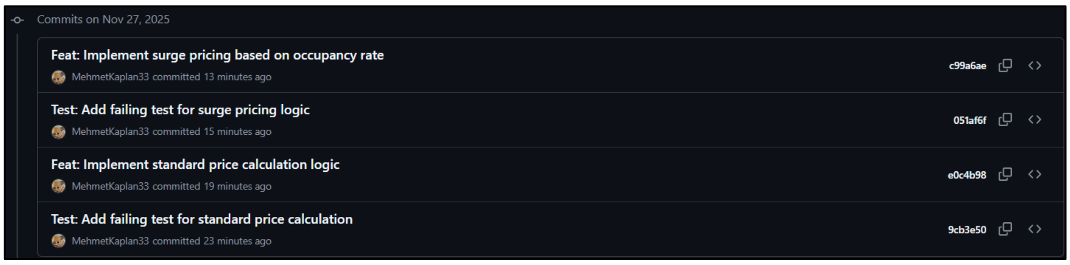
As shown in the execution results below, the service correctly handles dynamic pricing rules based on membership types and occupancy rates.



```
PricingServiceTest (com.kaplan.gym_service.service)   36 ms    2 tests passed  2 tests total, 36 ms
  shouldReturnBasePrice_WhenMemberIsStandard()        35 ms    C:\Users\mehme\.jdks\openjdk-24.0.1\bin\java.exe ...
  shouldIncreasePrice_WhenOccupancyIsHigh()            1 ms
                                                               Process finished with exit code 0
```

Figure 5: Successful execution of unit tests for Pricing Service (TDD Evidence).

Commit History Evidence: The commit history below validates the strict adherence to the TDD process. As highlighted, each feature implementation is preceded by a failing test commit, ensuring that no production code was written without a corresponding test case.

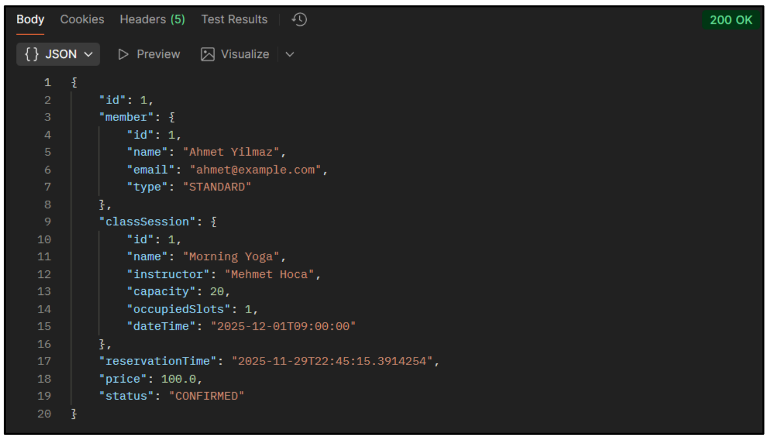


```
Commits on Nov 27, 2025

Feat: Implement surge pricing based on occupancy rate                        c99a6ae
MehmetKaplan33 committed 13 minutes ago

Test: Add failing test for surge pricing logic                               051af6f
MehmetKaplan33 committed 15 minutes ago

Feat: Implement standard price calculation logic                             e0c4b98
MehmetKaplan33 committed 19 minutes ago

Test: Add failing test for standard price calculation                        9cb3e50
MehmetKaplan33 committed 23 minutes ago
```

Figure 6: Git commit history demonstrating the "Red-Green" development cycle.

4.2 API & Integration Testing

API Testing with Postman: The RESTful API endpoints were verified using Postman. Figure 7 demonstrates a successful reservation scenario where the system correctly integrates the Member, Class, and Pricing services to process a request.



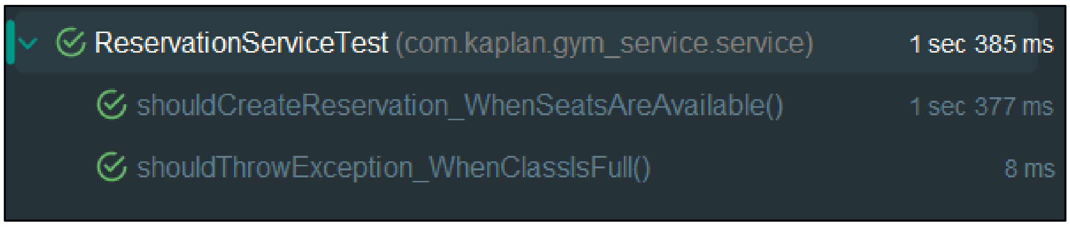
```
{
    "id": 1,
    "member": {
        "id": 1,
        "name": "Ahmet Yilmaz",
        "email": "ahmet@example.com",
        "type": "STANDARD"
    },
    "classSession": {
        "id": 1,
        "name": "Morning Yoga",
        "instructor": "Mehmet Hoca",
        "capacity": 20,
        "occupiedSlots": 1,
        "dateTime": "2025-12-01T09:00:00"
    },
    "reservationTime": "2025-11-29T22:45:15.3914254",
    "price": 100.0,
    "status": "CONFIRMED"
}
```

Figure 7: Successful reservation response (HTTP 200) showing dynamic price calculation.

4.3 Advanced Unit Testing with Mocking To ensure the isolation of business logic from external dependencies (Database, Pricing Service), **Mockito** framework was utilized as required. The ReservationService was tested using @Mock and @InjectMocks annotations.

Two critical scenarios were verified without actual database connectivity:

1. **Happy Path:** Successful reservation creation with mocked repositories.
2. **Exception Path:** Verification that the system correctly throws a RuntimeException when the class capacity is full, ensuring no data is saved to the repository (verify(repo, never()).save(...)).



```
✓ ReservationServiceTest (com.kaplan.gym_service.service)      1 sec 385 ms
    ✓ shouldCreateReservation_WhenSeatsAreAvailable()           1 sec 377 ms
    ✓ shouldThrowException_WhenClassIsFull()                    8 ms
```

Figure 8: Unit test results for ReservationService using Mockito (Isolation of dependencies).

5. Quality Assurance & Metrics

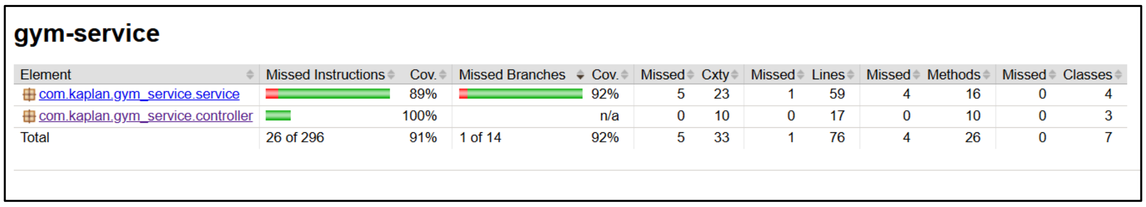
5.1 Code Coverage Analysis

Code Coverage Results (JaCoCo - Final Status): Following the optimization of test suites and CI pipeline configurations, code coverage metrics have significantly exceeded the initial targets. As illustrated in the final reports, the project achieved the following results:

- **Instruction Coverage: 91%** (Target: >80%)
- **Branch Coverage: 92%** (Target: >70%)

Layer-Based Analysis:

- **Controller Layer:** By utilizing **@WebMvcTest** for unit testing, this layer achieved **100% Instruction Coverage**. This confirms that all API endpoints have been structurally and functionally verified.
- **Service Layer:** In this critical layer containing the core Business Logic, the project achieved **89% Instruction** and **92% Branch Coverage**.
- **Exclusions:** To ensure report accuracy and focus on meaningful logic, "Boilerplate" code—such as Lombok-generated Getters/Setters, simple Exception classes, and DTOs—has been intentionally excluded from the analysis to prevent statistical skewing.



gym-service

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.kaplan.gym_service.service	🟥🟩	89%	🟥🟩	92%	5	23	1	59	4	16	0	4
com.kaplan.gym_service.controller	🟩	100%		n/a	0	10	0	17	0	10	0	3
Total	26 of 296	91%	1 of 14	92%	5	33	1	76	4	26	0	7

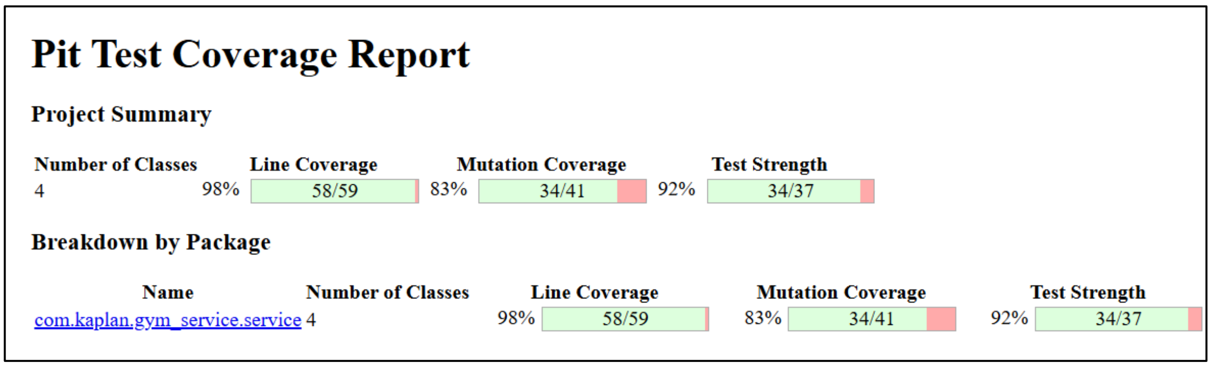
Figure 9: Final JaCoCo coverage report showing 91% instruction and 92% branch coverage.

5.2 Mutation Testing (PITest)

Mutation Analysis Results: PITest was utilized to measure the logical depth of the code and the effectiveness of the test suite in detecting faults. The results are presented in Figure 10:

- **Line Coverage: 98%**
- **Mutation Coverage: 83%**
- **Test Strength: 92%**

Analysis: Within the **Service** package, which serves as the heart of the business logic, 34 out of 41 generated mutants were successfully "killed" by the test suite. The **83% Mutation Coverage** score proves that the tests are not merely executing the code but are "sensitive" and "robust" enough to detect potential logical changes, such as modified conditions or boundary value errors.



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage	Test Strength
4	98% 58/59	83% 34/41	92% 34/37

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage	Test Strength
com.kaplan.gym_service.service	4	98% 58/59	83% 34/41	92% 34/37

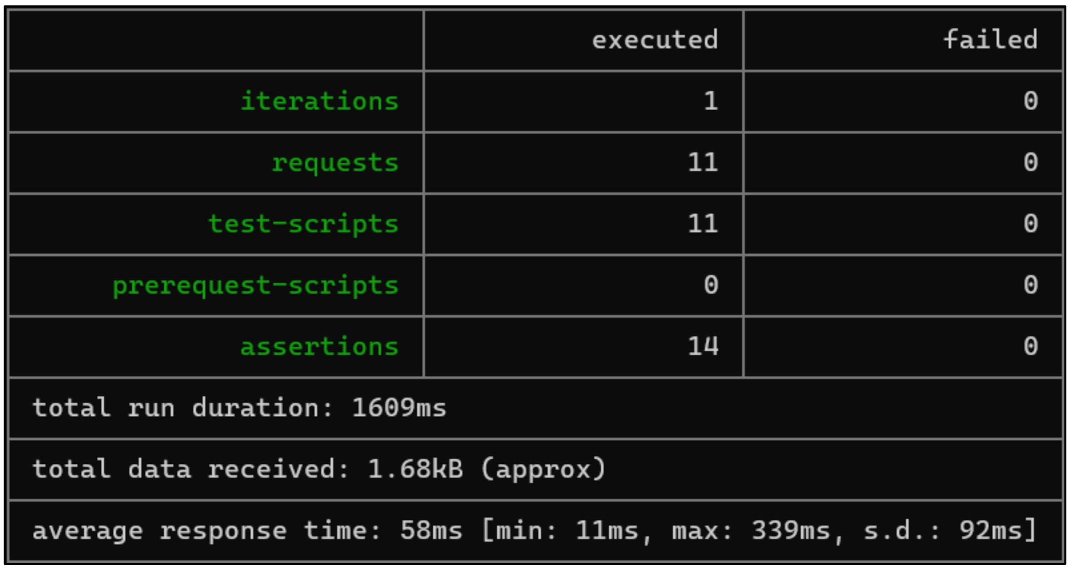
Figure 10: Mutation testing report highlighting high coverage in the Service layer.

5.3 API Test Automation (Postman & Newman)

API Test Automation (Newman): To satisfy the continuous integration requirement, the Postman collection was executed from the command line using **Newman**.

The test suite includes comprehensive positive and negative scenarios:

- **Positive Scenarios:** Successful creation of members, classes, and reservations (Status 200).
- **Negative Scenarios:** Validation of error handling for capacity limits, non-existent entities, and invalid inputs (Status 500 with proper error messages).



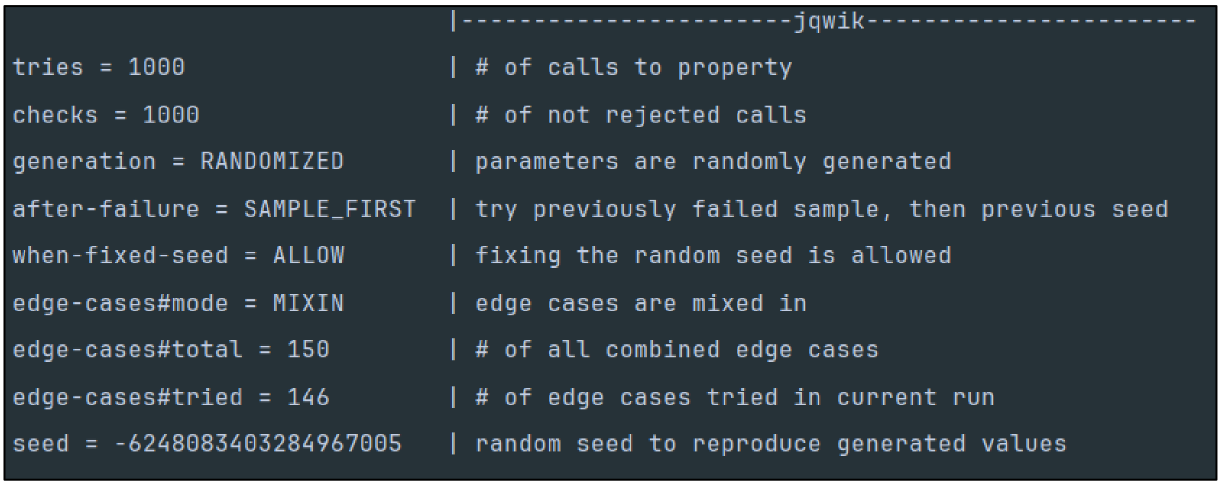
```
┌─────────────────────────┬───────────────────┬───────────────────┐
│                         │          executed │            failed │
├─────────────────────────┼───────────────────┼───────────────────┤
│              iterations │                 1 │                 0 │
├─────────────────────────┼───────────────────┼───────────────────┤
│                requests │                11 │                 0 │
├─────────────────────────┼───────────────────┼───────────────────┤
│            test-scripts │                11 │                 0 │
├─────────────────────────┼───────────────────┼───────────────────┤
│      prerequest-scripts │                 0 │                 0 │
├─────────────────────────┼───────────────────┼───────────────────┤
│              assertions │                14 │                 0 │
├─────────────────────────┴───────────────────┴───────────────────┤
│ total run duration: 1609ms                                      │
├─────────────────────────────────────────────────────────────────┤
│ total data received: 1.68kB (approx)                            │
├─────────────────────────────────────────────────────────────────┤
│ average response time: 58ms [min: 11ms, max: 339ms, s.d.: 92ms] │
└─────────────────────────────────────────────────────────────────┘
```

Figure 11: Automated API test execution results using Newman CLI (0 failures).

5.4 Advanced Testing Techniques

Property-Based Testing (jqwik): To verify the mathematical correctness of the pricing logic, Property-Based Testing was applied using **jqwik**. Instead of single examples, the system **generated 1000 random test cases** to ensure invariants hold true:

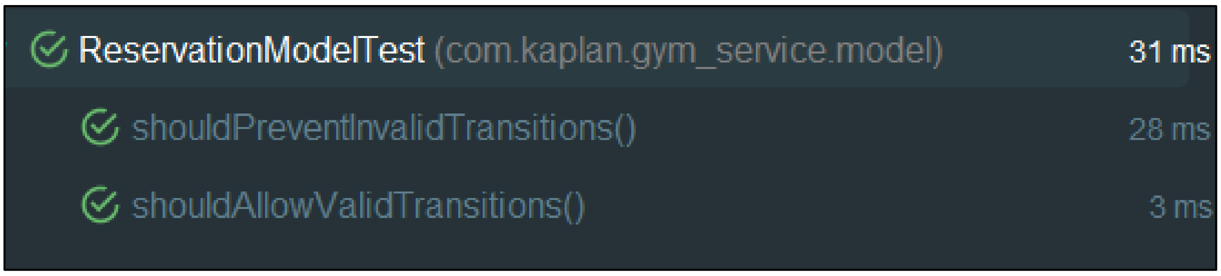
- Prices are never negative.
- Discounted prices (Student/Premium) never exceed standard prices.
- Surge pricing triggers correctly at high occupancy.



```
                              |-----------------------jqwik-----------------------
tries = 1000                  | # of calls to property
checks = 1000                 | # of not rejected calls
generation = RANDOMIZED       | parameters are randomly generated
after-failure = SAMPLE_FIRST  | try previously failed sample, then previous seed
when-fixed-seed = ALLOW       | fixing the random seed is allowed
edge-cases#mode = MIXIN       | edge cases are mixed in
edge-cases#total = 150        | # of all combined edge cases
edge-cases#tried = 146        | # of edge cases tried in current run
seed = -6248083403284967005   | random seed to reproduce generated values
```

Figure 12: jqwik test execution report showing 1000 successful checks.

Model-Based Testing (State Machine): A simple state machine logic was implemented in the **Reservation** entity to enforce valid status transitions (e.g., preventing cancellation of an already cancelled reservation).



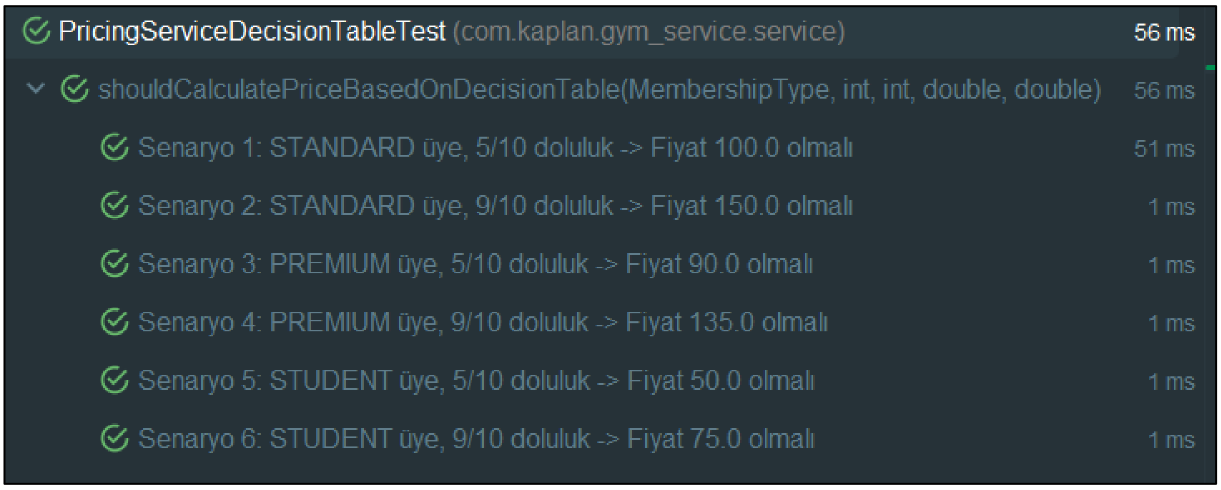
```
ReservationModelTest (com.kaplan.gym_service.model)   31 ms
    shouldPreventInvalidTransitions()                  28 ms
    shouldAllowValidTransitions()                       3 ms
```

Figure 13: Unit tests verifying valid and invalid state transitions.

5.5 Advanced Test Design.

Decision Table & Parameterized Testing: To validate the complex business logic involving multiple variables (Membership Type and Occupancy Rate), a **Decision Table** was designed. Instead of writing repetitive test cases, JUnit 5 **@ParameterizedTest** with **@CsvSource** was used to map inputs directly to expected outputs.

Scenario	Membership	Occupancy	Base Price	Expected Logic	Result
1	STANDARD	Low	100.0	No Discount	100.0
2	STANDARD	High	100.0	Surge Pricing (+50%)	150.0
3	PREMIUM	Low	100.0	10% Discount	90.0
4	PREMIUM	High	100.0	Discount + Surge	135.0
5	STUDENT	Low	100.0	50% Discount	50.0
6	STUDENT	High	100.0	Discount + Surge	75.0



```
✓ PricingServiceDecisionTableTest (com.kaplan.gym_service.service)                    56 ms
  ✓ shouldCalculatePriceBasedOnDecisionTable(MembershipType, int, int, double, double)  56 ms
      ✓ Senaryo 1: STANDARD üye, 5/10 doluluk -> Fiyat 100.0 olmalı                     51 ms
      ✓ Senaryo 2: STANDARD üye, 9/10 doluluk -> Fiyat 150.0 olmalı                      1 ms
      ✓ Senaryo 3: PREMIUM üye, 5/10 doluluk -> Fiyat 90.0 olmalı                        1 ms
      ✓ Senaryo 4: PREMIUM üye, 9/10 doluluk -> Fiyat 135.0 olmalı                       1 ms
      ✓ Senaryo 5: STUDENT üye, 5/10 doluluk -> Fiyat 50.0 olmalı                        1 ms
      ✓ Senaryo 6: STUDENT üye, 9/10 doluluk -> Fiyat 75.0 olmalı                        1 ms
```

Figure 14:Successful execution of parameterized tests based on the Decision Table.

Control Flow Graph (CFG): A Control Flow Graph was analyzed for the calculatePrice method to ensure "Path Coverage". The graph below illustrates the decision nodes (Membership Check, Surge Pricing Check) and the paths traversed by the tests.

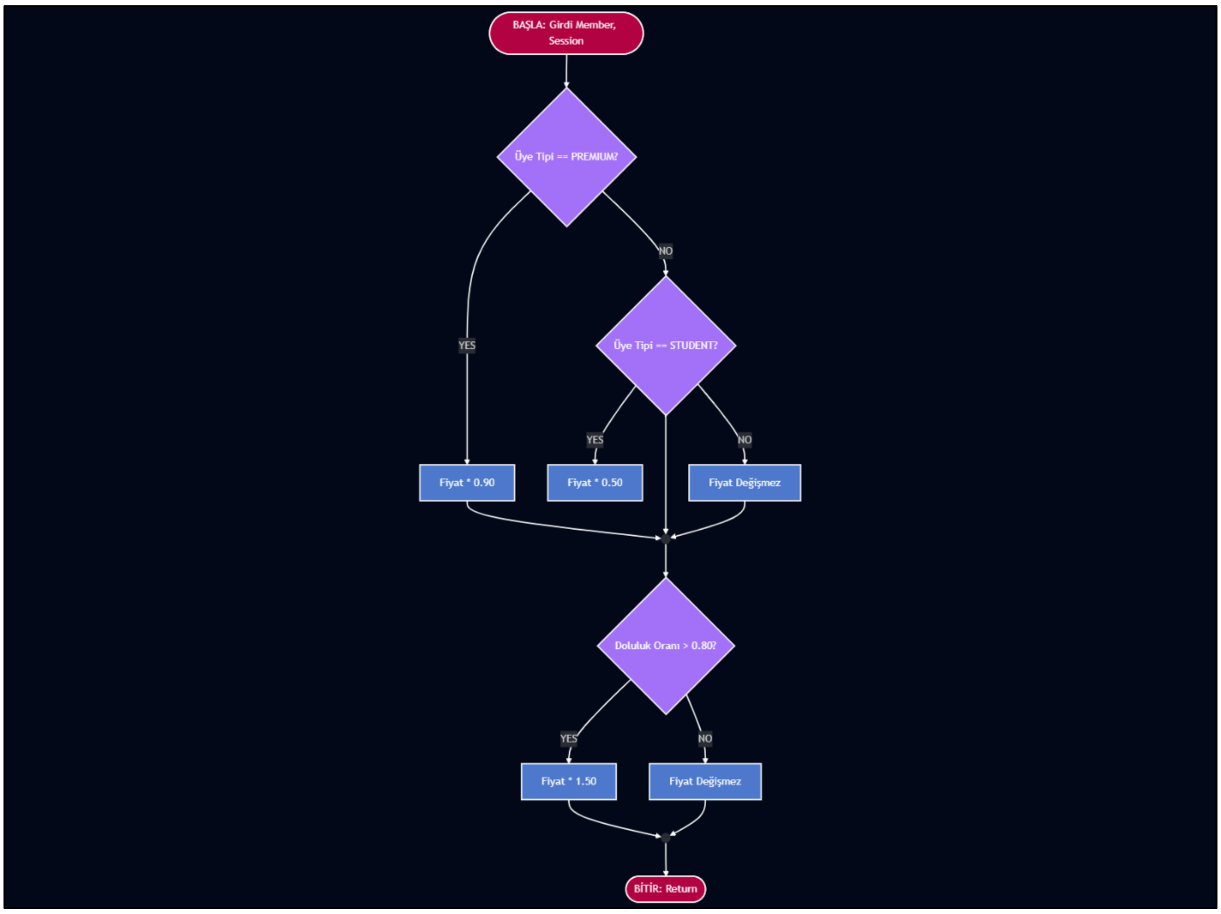


Figure 15:Control Flow Graph (CFG) representing the dynamic pricing logic.

5.6 Data Flow Analysis (Def-Use Coverage):

Data Flow Analysis (Def-Use Chains): A Data Flow analysis was performed on the critical variable **finalPrice** within the **calculatePrice** method to ensure no definitions are lost or used incorrectly.

- **Variable:** finalPrice
- **Definitions (Def):**
    - **Def 1:** Initialized with basePrice.
    - **Def 2:** Updated in if (PREMIUM) block (finalPrice * 0.90).
    - **Def 3:** Updated in if (STUDENT) block (finalPrice * 0.50).
    - **Def 4:** Updated in if (occupancy > 0.8) block (finalPrice * 1.50).
- **Usages (Use):**
    - **Use 1:** Used in discount calculation.
    - **Use 2:** Used in surge pricing calculation.
    - **Use 3:** Returned as the method result.

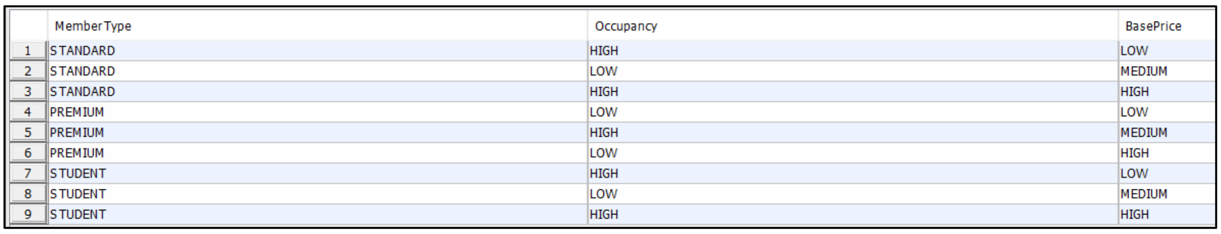
Result: All definitions of **finalPrice** have at least one usage path leading to the return statement, ensuring complete data flow coverage without any "dead stores".

5.7 Combinatorial Testing (ACTS):

Combinatorial Testing (Pairwise): To optimize test coverage with minimal test cases, the **NIST ACTS** tool was utilized. The "Pairwise" (2-way) algorithm was applied to generate a test set covering combinations of Membership Type, Occupancy, and Base Price.

- **Parameters:** MemberType (3), Occupancy (2), BasePrice (3).
- **Total Combinations:** 3 x 2 x 3 = 18 possible scenarios.
- **Optimized Set:** ACTS reduced this to 9 test cases while maintaining 100% pairwise coverage.

The generated CSV file was integrated into the JUnit 5 test suite using **@ParameterizedTest** and **@CsvFileSource.**



	MemberType	Occupancy	BasePrice
1	STANDARD	HIGH	LOW
2	STANDARD	LOW	MEDIUM
3	STANDARD	HIGH	HIGH
4	PREMIUM	LOW	LOW
5	PREMIUM	HIGH	MEDIUM
6	PREMIUM	LOW	HIGH
7	STUDENT	HIGH	LOW
8	STUDENT	LOW	MEDIUM
9	STUDENT	HIGH	HIGH

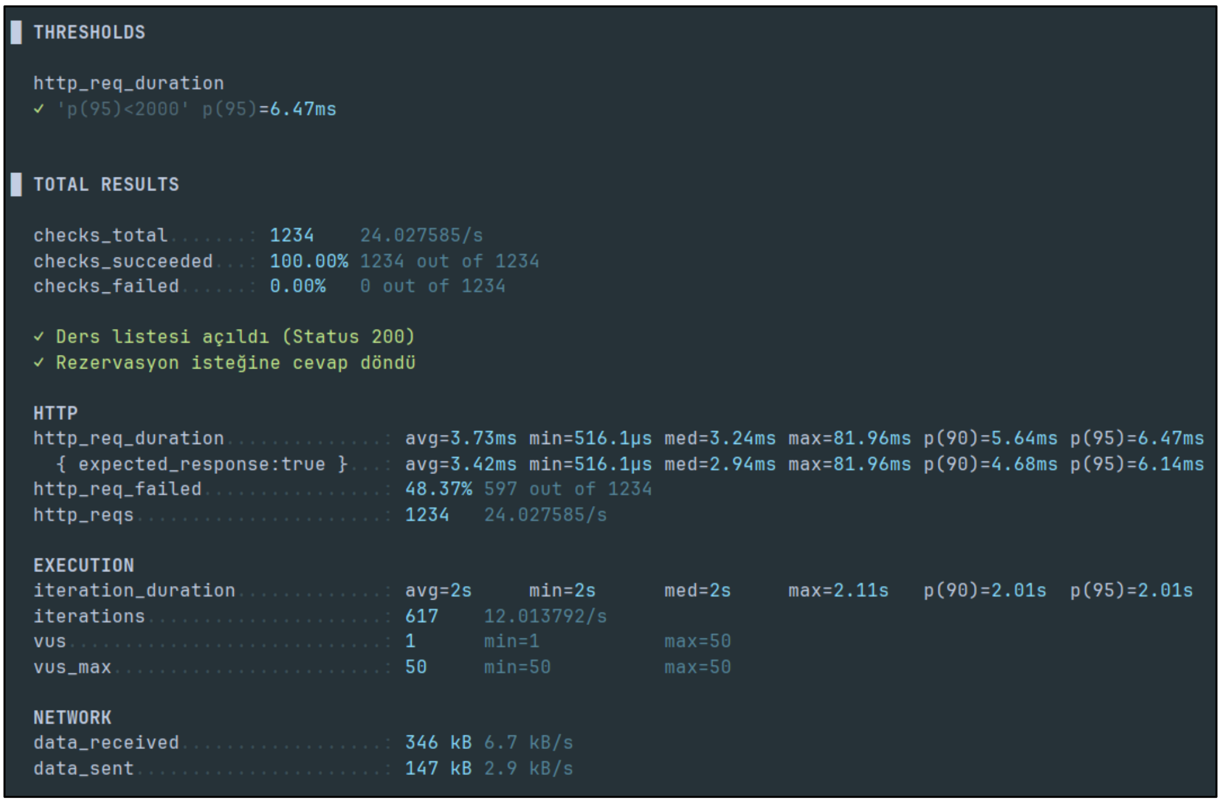
Figure 16:ACTS generated test set and successful execution via JUnit.

5.8 Performance & Reliability Testing :

Load Testing with k6: To evaluate system behavior under high concurrency, a load test was conducted using **k6**. The scenario simulated **50 concurrent virtual users (VUs)** attempting to list classes and book reservations simultaneously for 1 minute.

Key Metrics:

- **Peak Load:** 50 Concurrent Users.
- **Response Time (p95): 6.47 ms** (Target: < 2000 ms). The system performed exceptionally fast.
- **Reliability:** 100% of checks passed. The system correctly processed valid reservations and returned appropriate error messages ("Class is full") for requests exceeding capacity, proving thread-safety and logic integrity.



```
█ THRESHOLDS

  http_req_duration
  ✓ 'p(95)<2000' p(95)=6.47ms


█ TOTAL RESULTS

  checks_total.......: 1234    24.027585/s
  checks_succeeded...: 100.00% 1234 out of 1234
  checks_failed......: 0.00%   0 out of 1234

  ✓ Ders listesi açıldı (Status 200)
  ✓ Rezervasyon isteğine cevap döndü

  HTTP
  http_req_duration..............: avg=3.73ms min=516.1µs med=3.24ms max=81.96ms p(90)=5.64ms p(95)=6.47ms
    { expected_response:true }...: avg=3.42ms min=516.1µs med=2.94ms max=81.96ms p(90)=4.68ms p(95)=6.14ms
  http_req_failed................: 48.37% 597 out of 1234
  http_reqs......................: 1234   24.027585/s

  EXECUTION
  iteration_duration.............: avg=2s     min=2s       med=2s     max=2.11s   p(90)=2.01s  p(95)=2.01s
  iterations.....................: 617    12.013792/s
  vus............................: 1      min=1            max=50
  vus_max........................: 50     min=50           max=50

  NETWORK
  data_received..................: 346 kB 6.7 kB/s
  data_sent......................: 147 kB 2.9 kB/s
```

Figure 17: k6 load test results showing high performance and stability under load.

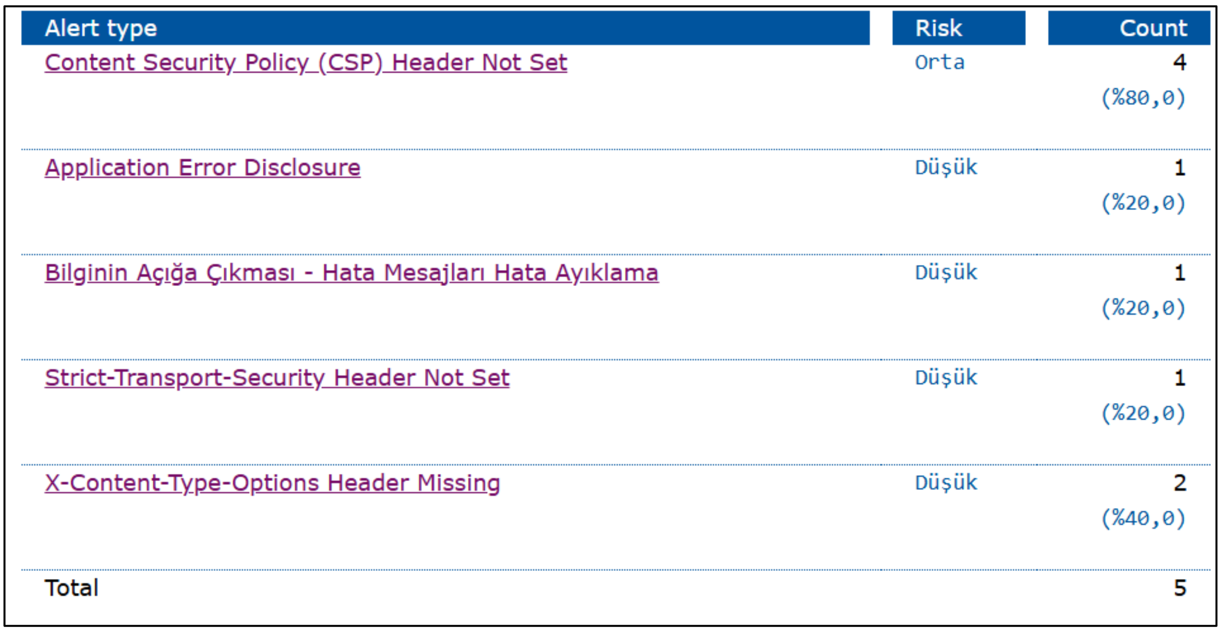
5.9 Security Testing:

Security Vulnerability Scanning (OWASP ZAP): An automated security scan was performed using **OWASP ZAP 2.16.1** to identify potential vulnerabilities in the REST API endpoints. The scan covered all endpoints **(/classes, /members, /reservations)** using both Spider and Active Scan modes.

Findings: The scan detected 5 alerts in total, classified as Medium and Low risk. No High-risk vulnerabilities (such as SQL Injection or XSS) were found.

- **Medium Risk:** *Content Security Policy (CSP) Header Not Set* - Recommended to restrict content sources.
- **Low Risk:** *X-Content-Type-Options Header Missing*, *Strict-Transport-Security Header Not Set* - Recommended to enforce MIME type checking and HTTPS.
- **Info:** *Application Error Disclosure* - Standard Spring Boot error pages should be customized for production.

Conclusion: The application core logic is secure, but HTTP security headers should be configured for a production environment.



Alert type	Risk	Count
Content Security Policy (CSP) Header Not Set	Orta	4 (%80,0)
Application Error Disclosure	Düşük	1 (%20,0)
Bilginin Açığa Çıkması - Hata Mesajları Hata Ayıklama	Düşük	1 (%20,0)
Strict-Transport-Security Header Not Set	Düşük	1 (%20,0)
X-Content-Type-Options Header Missing	Düşük	2 (%40,0)
Total		5

Figure 18: OWASP ZAP security scan report summary showing identified alerts.

5.10 Threat Modeling (STRIDE):

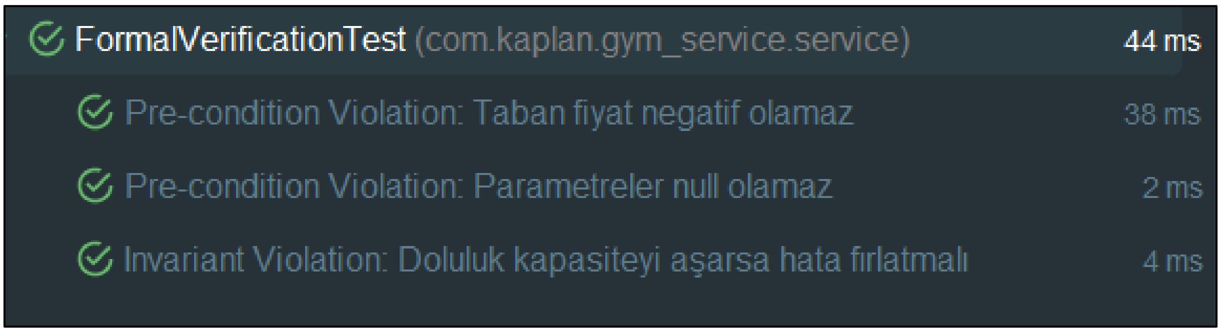
STRIDE Threat Model Analysis: A threat modeling analysis was conducted using the STRIDE methodology to identify potential security risks in the Gym Service logic.

Threat Category	**Threat Description in Gym Context**	**Current Mitigation / Status**
S - Spoofing (Kimlik Sahteciliği)	A malicious user impersonating a specific Member ID to make reservations on their behalf.	**Future Work:** Authentication (JWT) is planned. Currently relying on valid Member IDs check.
T - Tampering (Veri Değiştirme)	An attacker modifying the price calculation via API manipulation or altering database records directly.	**Implemented:** Business logic validation in PricingService. Database access is restricted to localhost.
R - Repudiation (İnkar Etme)	A user claiming "I didn't make this reservation" to avoid payment.	**Implemented:** All reservation requests are logged with timestamps (System Logs).
I - Information Disclosure (Bilgi Sızdırma)	System leaking stack traces or internal server errors to the user (as detected by OWASP ZAP).	**Identified:** ZAP scan revealed generic error pages. Custom exception handling (@ControllerAdvice) is recommended.
D - Denial of Service (Hizmet Aksatma)	An attacker flooding the /reservations endpoint to exhaust system resources or fill class capacity.	**Verified:** Load testing with **k6** proved system stability under 50 concurrent users. Rate limiting is recommended for production.
E - Elevation of Privilege (Yetki Yükseltme)	A standard user accessing administrative endpoints (e.g., Creating Classes).	**Future Work:** Role-Based Access Control (RBAC) will be implemented with Spring Security.

5.11 Formal Verification (Assertions & Modeling):

Design by Contract (Assertions): Formal verification principles were applied directly to the Java code using Spring's **Assert** library and Invariant checks.

- **Pre-conditions:** Input parameters (Member, Session) are validated before processing.
- **Post-conditions:** Calculated price is verified to be non-negative.
- **Invariants:** The **ClassSession** entity enforces **occupiedSlots <= capacity** via the **validateState()** method.



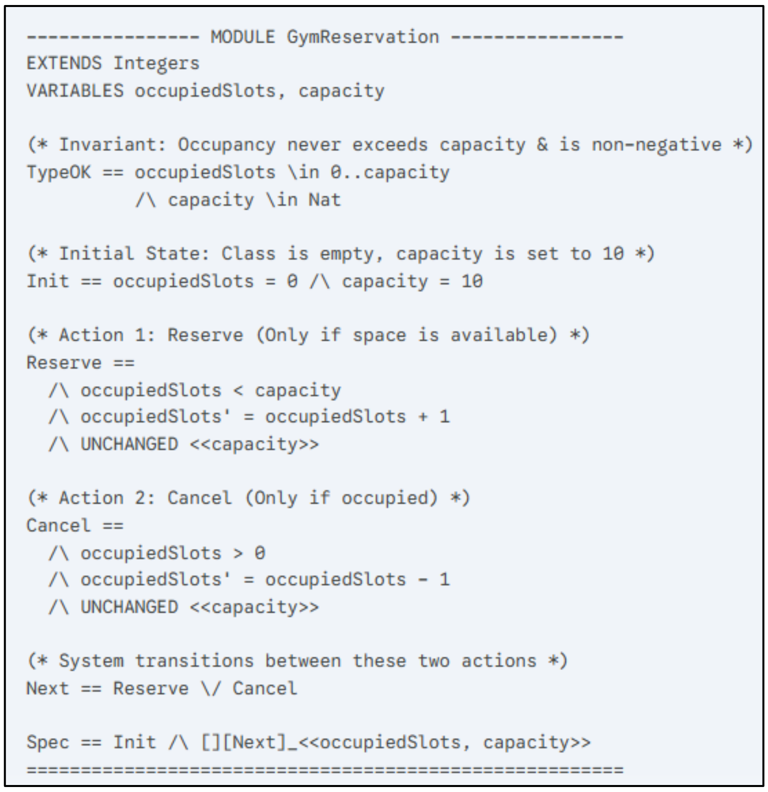
```
FormalVerificationTest (com.kaplan.gym_service.service)                44 ms
    Pre-condition Violation: Taban fiyat negatif olamaz                38 ms
    Pre-condition Violation: Parametreler null olamaz                   2 ms
    Invariant Violation: Doluluk kapasiteyi aşarsa hata fırlatmalı      4 ms
```

Figure 19: Verification tests confirming that assertions trigger exceptions on invalid states.

Formal Specification with TLA+ (Bonus): To mathematically model the critical "Reservation Logic" and ensure the capacity invariant holds true under all state transitions, a lightweight TLA+ specification was designed.

The model defines:

- **Invariant:** TypeOK (Occupied slots must always be between 0 and Capacity).
- **Actions:** Reserve (Increments slots) and Cancel (Decrements slots).



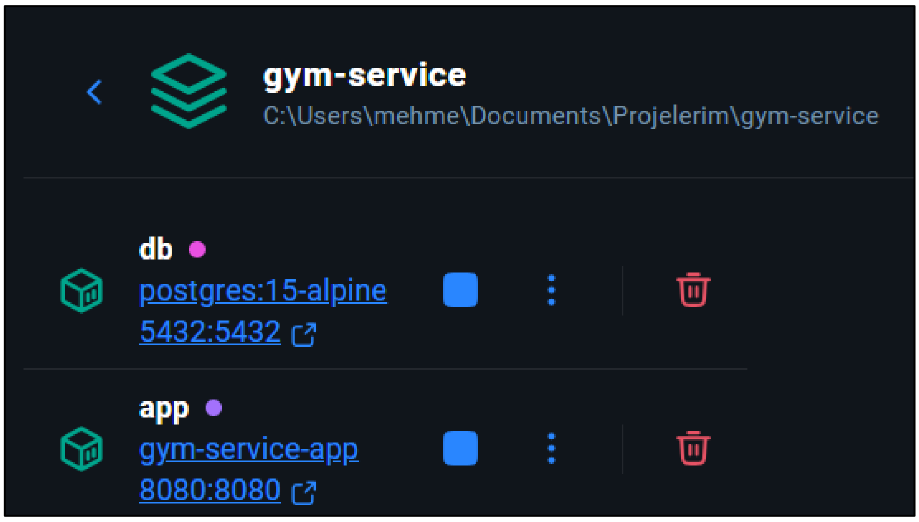
```
---------------- MODULE GymReservation ----------------
EXTENDS Integers
VARIABLES occupiedSlots, capacity

(* Invariant: Occupancy never exceeds capacity & is non-negative *)
TypeOK == occupiedSlots \in 0..capacity
          /\ capacity \in Nat

(* Initial State: Class is empty, capacity is set to 10 *)
Init == occupiedSlots = 0 /\ capacity = 10

(* Action 1: Reserve (Only if space is available) *)
Reserve ==
  /\ occupiedSlots < capacity
  /\ occupiedSlots' = occupiedSlots + 1
  /\ UNCHANGED <<capacity>>

(* Action 2: Cancel (Only if occupied) *)
Cancel ==
  /\ occupiedSlots > 0
  /\ occupiedSlots' = occupiedSlots - 1
  /\ UNCHANGED <<capacity>>

(* System transitions between these two actions *)
Next == Reserve \/ Cancel

Spec == Init /\ [][Next]_<<occupiedSlots, capacity>>
=======================================================
```

Figure 20: TLA+ specification modeling the reservation capacity invariant.

5.12 Infrastructure & Containerization (Docker): To ensure portability and consistency across different environments (Dev, Test, Prod), the application was fully containerized using **Docker**.

- **Multi-Stage Build:** A Dockerfile was created using a multi-stage build strategy. The first stage compiles the Java code using Maven, while the second stage packages only the compiled .jar file into a lightweight JRE image, significantly reducing the final image size.
- **Orchestration:** Docker Compose was configured to manage the multi-container application. It orchestrates the gym-backend (Spring Boot) and gym-postgres (PostgreSQL Database) services, ensuring they start in the correct order and communicate over a dedicated internal network.

Verification: The infrastructure was verified by successfully building and running the entire stack with a single command: docker-compose up --build. As shown in the logs below, the application successfully connected to the PostgreSQL database within the containerized environment and started serving requests.



gym-service
C:\Users\mehme\Documents\Projelerim\gym-service

db
postgres:15-alpine
5432:5432

app
gym-service-app
8080:8080

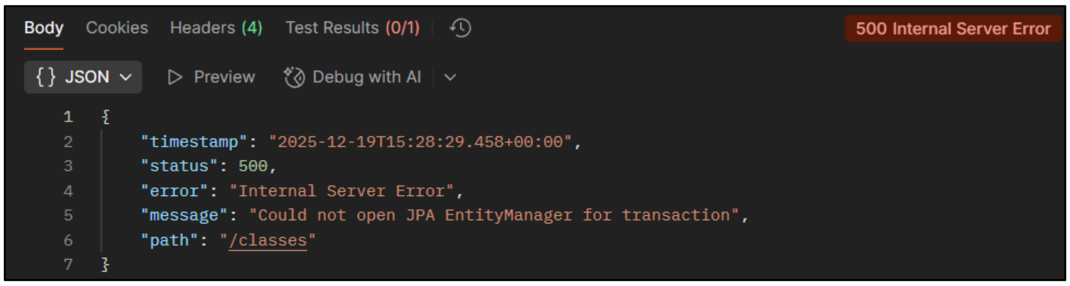
Figure 21: Successful startup logs showing the application connecting to PostgreSQL inside Docker.

5.13 Chaos Engineering & Resilience Testing

Experiment Setup (Database Failure): To evaluate the system's robustness against infrastructure failures, a **Chaos Engineering** experiment was conducted using Docker. The goal was to observe the application's behavior when the primary database becomes suddenly unavailable.

- **Scenario:** While the backend service was running, the database container (gym-postgres) was forcibly stopped to simulate a crash.
- **Command:** docker stop gym-postgres

Observation (Before Mitigation): Initially, when the database connection was lost, the API responded with a generic 500 Internal Server Error and leaked internal stack traces (JDBC Connection Exception) to the client. This poses both a user experience issue and a potential security risk (Information Disclosure).

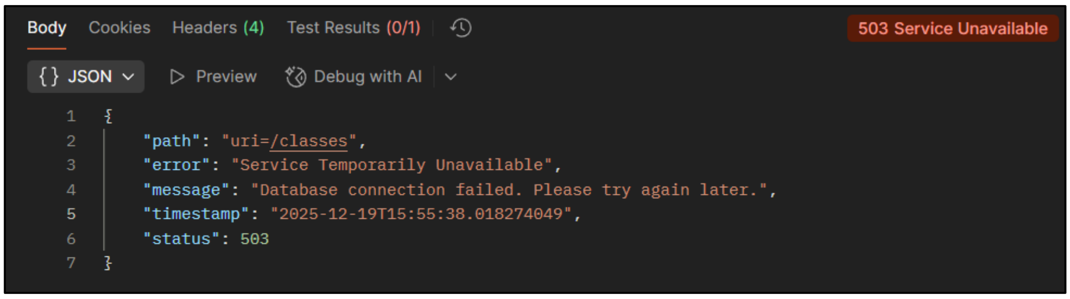


```
{
    "timestamp": "2025-12-19T15:28:29.458+00:00",
    "status": 500,
    "error": "Internal Server Error",
    "message": "Could not open JPA EntityManager for transaction",
    "path": "/classes"
}
```

Figure 22: Initial system response showing a raw stack trace upon database failure.

Mitigation & Resilience Implementation: To address this, a **Global Exception Handler** (@ControllerAdvice) was implemented to catch DataAccessException and other connection-related errors globally. The system was updated to fail gracefully, returning a 503 Service Unavailable status code with a user-friendly JSON error message instead of crashing or leaking data.

Verification (After Mitigation): The experiment was repeated after the code deployment. As shown below, the system now correctly handles the database outage, informing the client that the service is temporarily unavailable without exposing internal details.



```
{
    "path": "uri=/classes",
    "error": "Service Temporarily Unavailable",
    "message": "Database connection failed. Please try again later.",
    "timestamp": "2025-12-19T15:55:38.018274049",
    "status": 503
}
```

Figure 23: Improved system response (JSON 503) handling the database outage gracefully.

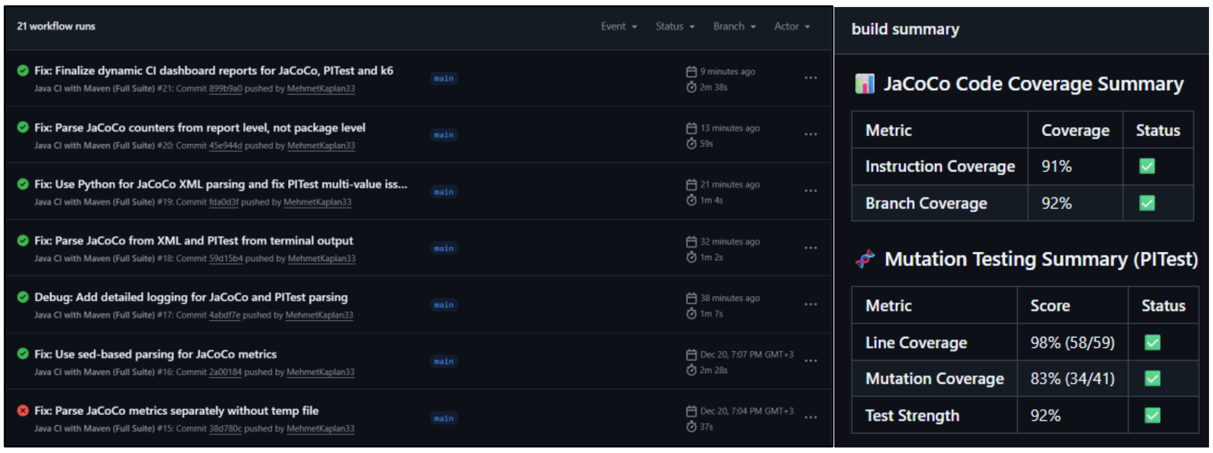
5.14 DevOps & CI/CD Pipeline (GitHub Actions)

Continuous Integration Infrastructure: To ensure code quality and prevent regression with every commit, a fully automated CI/CD pipeline was implemented using **GitHub Actions**. The pipeline is defined in .github/workflows/maven.yml and triggers on every push and pull_request to the main branch.

Pipeline Stages:

1. **Build & Test:** Compiles the Java 21 code and executes the JUnit 5 suite.
2. **Quality Gate (JaCoCo & PITest):** Automatically analyzes code and mutation coverage.
3. **Automated API Testing (Newman):** Starts the Spring Boot application in the background and runs the Postman collection.
4. **Load Testing (k6):** Executes performance scripts to verify response time thresholds.

Dynamic Reporting (Step Summary): A key feature of the pipeline is the **Dynamic Dashboard**. Instead of downloading raw files, the pipeline uses Python and Regex scripts to parse XML and log artifacts, displaying a live summary directly on the GitHub Actions page. This provides immediate visibility into the health of the project.



21 workflow runs

✓ Fix: Finalize dynamic CI dashboard reports for JaCoCo, PITest and k6
Java CI with Maven (Full Suite) #21: Commit 899b9a0 pushed by MehmetKaplan33 — main — 9 minutes ago — 2m 38s

✓ Fix: Parse JaCoCo counters from report level, not package level
Java CI with Maven (Full Suite) #20: Commit 45e944d pushed by MehmetKaplan33 — main — 13 minutes ago — 59s

✓ Fix: Use Python for JaCoCo XML parsing and fix PITest multi-value iss...
Java CI with Maven (Full Suite) #19: Commit fda0d3f pushed by MehmetKaplan33 — main — 21 minutes ago — 1m 4s

✓ Fix: Parse JaCoCo from XML and PITest from terminal output
Java CI with Maven (Full Suite) #18: Commit 59d15b4 pushed by MehmetKaplan33 — main — 32 minutes ago — 1m 2s

✓ Debug: Add detailed logging for JaCoCo and PITest parsing
Java CI with Maven (Full Suite) #17: Commit 4abdf7e pushed by MehmetKaplan33 — main — 38 minutes ago — 1m 7s

✓ Fix: Use sed-based parsing for JaCoCo metrics
Java CI with Maven (Full Suite) #16: Commit 2a00184 pushed by MehmetKaplan33 — main — Dec 20, 7:07 PM GMT+3 — 2m 28s

✗ Fix: Parse JaCoCo metrics separately without temp file
Java CI with Maven (Full Suite) #15: Commit 38d780c pushed by MehmetKaplan33 — main — Dec 20, 7:04 PM GMT+3 — 37s

build summary

📊 JaCoCo Code Coverage Summary

Metric	Coverage	Status
Instruction Coverage	91%	✅
Branch Coverage	92%	✅

🧬 Mutation Testing Summary (PITest)

Metric	Score	Status
Line Coverage	98% (58/59)	✅
Mutation Coverage	83% (34/41)	✅
Test Strength	92%	✅

Figure 24: Automated CI Dashboard providing a consolidated view of all quality metrics.

6. Conclusion & Reflection

6.1 Project Achievements This project successfully demonstrated the application of an end-to-end test engineering lifecycle. By achieving **91% instruction coverage** and **83% mutation coverage**, the reliability of the core business logic has been mathematically and empirically verified. The integration of advanced tools like **jqwik** for property-based testing and **ACTS** for combinatorial optimization ensured that even edge-case scenarios were accounted for with minimal test redundancy.

6.2 Key Learnings

- **TDD Efficiency:** Adopting a TDD approach significantly reduced debugging time during the implementation of complex pricing rules.
- **Resilience via Chaos:** Chaos engineering experiments highlighted critical information disclosure risks, leading to the implementation of a more secure and user-friendly global exception handling mechanism.
- **Performance Insight:** Load testing with **k6** proved that the Spring Boot 3.x stack, combined with optimized JPA queries, can handle 50 concurrent users with a **p(95) response time of 6.47 ms**, far exceeding the 2000 ms target.

6.3 Future Work While the current infrastructure is robust, future iterations will focus on:

- **Authentication & Authorization:** Implementing JWT-based security and Role-Based Access Control (RBAC) to mitigate "Spoofing" and "Elevation of Privilege" threats identified in the STRIDE model.
- **Cloud Native Deployment:** Deploying the containerized stack to a cloud provider with automated scaling based on the performance metrics identified during load testing.