Table of Contents

List of Figures

Solutions

1 Deliverables

1.1 Deliverable 1

Q: How did you design the unit test cases?

Solution

In creating unit test cases for the GroupsModel and GroupsResource classes, I implemented a thorough and stringent framework that was independent of any third-party testing libraries. The testing method was based on strong principles of testing, with the goal of withstanding changes and refactoring of the code, thus ensuring long-term effectiveness of the test suite.
My test design was guided foremost by the principle of functional isolation. It was important to provide complete verification of each aspect of the Groups functionality, in addition to minimizing the dependency on external systems or frameworks. This isolation-based strategy improves testing reliability by creating a controlled situation where test outcomes are foreseeable and can be traced directly to the code under test.

Strategic Testing Elements:

With respect to the GroupsModel, I created a suite of tests around its primary functionality: managing collections of plugs, facilitating group operations, and maintaining group integrity. Every method was thoroughly tested, from basic group creation and membership manipulation to more complex activities like performing actions over group members. Special attention was given to the synchronized nature of these methods, recognizing their importance in a potentially concurrent environment.

Testing of the GroupsResource supplemented this approach by focusing on the REST interface layer, thus ensuring correct functionality of the endpoints, handling of requests, and building responses. These tests ensured that the API layer correctly mapped HTTP requests onto actions within the GroupsModel and properly formatted the resulting data for client usage.

Handling edge cases was a crucial part of my testing strategy. Specifically, for GroupsModel, I had test cases for cases of blank groups, duplicate members, and non-existent groups. For GroupsResource, I tested functionality when it encountered missing or malformed parameters, incorrect actions, and resources that could not be found. Both classes had tests simulating error conditions when performing operations to ensure correct failure handling.

Implementation characteristics

Rather than relying on complicated mock objects, I came up with simple test cases using specific test classes. As an example, for the GroupsModel tests, I came up with a TestPlugsResource that mimics the functionality of the actual PlugsResource without requiring its dependencies. Similarly, for the GroupsResource tests, I came up with a TestGroupsModel to decouple the resource from the implementation of the model.

Every testing approach followed a clearly established protocol:

- Setting up the test environment and defining preconditions required
- Commencement: Starting the function being studied.
- Verification: Confirming the expected outcomes occurred

This structure enhanced the clarity of the manuscript and made it easy to comprehensively validate each trait. The testing methodologies were carefully planned, beginning with simple functions and then advancing to complicated interactions. This method encourages a rational sequence of tests, where simple features establish confidence before handling complicated functionalities.

For assertion handling, I crafted custom assertion methods that provide clear, meaningful feedback when tests fail. These methods improve both test reliability and debugging efficiency by clearly communicating what went wrong during test execution. Throughout the creation of these tests, I followed a strict separation of the test cases. Each test is independent, with new instances of all test classes created for every method. This independence ensures that any failure during testing is correctly attributed to specific functionalities, without any interference from previous runs of the tests. The resulting test suite has full coverage of the GroupsModel and GroupsResource classes and demonstrates a high level of manageability and flexibility for future changes. Focusing on isolation and strict validation, the tests ensure the reliability of the system while at the same time serving as dynamic documentation on the expected behavior.

1.2 Deliverable 2

Q: How did you implement the features? What classes have you added/modified?

Solution

Inclusion of the tests for GroupsModel and GroupsResource required careful analysis of their respective functionalities within the IoT system architecture. My implementation strategy focused on creating standalone test classes that thoroughly evaluated the functionality without the injection of unnecessary complexity or external dependencies.

To analyze these components, I designed focused test classes that mirror the system architecture with specific areas of testing purposes assigned to them. By carefully analyzing the purpose of each class, I performed implementations of tests meant to check for all functionalities they support.

GroupsModel Testing Implementation:

The GroupsModel is the domain model responsible for handling collections of plugs, allowing for synchronized access to group-relevant operations. To ensure the integrity of this class, I implemented GroupsModelTest—a comprehensive suite of tests aimed at confirming the effectiveness of group management and action propagation across group members.

One major hindrance encountered when evaluating GroupsModel was its reliance on calling actions from PlugsResource. Rather than using actual instances of PlugsResource along with their extrinsic dependencies, a TestPlugsResource class is implemented to mimic the functionalities required. The testing surrogate provides controlled responses to method calls, thus allowing for a determination of whether GroupsModel is correctly interacting with the resources or not without using actual resources.

Execution of the test methodically examines:

- Group building and management of membership.
- Group removal functionality
- Compilation of aggregate data
- Carrying out group members' assignments.
- Managing errors during collaborative work
- Thread safety of synchronized methods

To test action performance, I created mechanisms to reproduce both successful transactions and a variety of failure scenarios. The TestPlugsResource class includes the ability to produce controlled exceptions when configured correctly, making it easier to test the GroupsModel's response to errors.

GroupsResource Testing Approach:

The GroupsResource class provides RESTful interfaces for group management, mapping HTTP requests to operations relevant to the GroupsModel. To thoroughly evaluate the group management layer, I implemented the GroupsResourceTest class with a focus on endpoint functionality, request parameter management, and response building.

Similar to the model testing, a custom implementation of TestGroupsModel was used to decouple the resource layer from the actual model dependencies. This approach allows for proper testing of the interactions between the resource layer and the model without introducing dependencies on the actual model implementation. The test implementation comprehensively validates:

- Finding endpoints and controlling request administration.
- Parameter extraction and validation
- Choosing response status codes
- Response body formation
- Error condition handling
- Dynamics of interaction within the GroupsModel

For specific endpoint testing, I implemented verification for all supported operations:

- Group listing (GET /api/groups)
- Individual group retrieval (GET /api/groups/{group})
- Group action execution (GET with action parameter)
- Group creation (POST /api/groups)
- Group membership updates (POST /api/groups/{group})
- Group deletion (DELETE /api/groups/{group})

Each endpoint analysis considers not just the best case of correct execution but also a spectrum of error cases to ensure fault-tolerant performance.
Implementation Framework:
Throughout the implementation process, I maintained a consistent structure throughout the test classes. The GroupsModelTest and GroupsResourceTest illustrate similar approaches:

- Develop a protocol that creates new test cases.
- Tailor-made assertion methods for systematic verification
- Test methods organized by functionality
- Support functions to reduce coding duplication

The solution avoids external testing frameworks by creating an independent testing solution that includes custom assertion methods and test utilities. This approach enhances portability and reduces dependencies on external systems, while still providing complete validation.

Key classes used for testing purposes include:
GroupsModelTest: Comprehensive test suite for GroupsModel functionality GroupsResourceTest: Testing the RESTful endpoint design. TestPlugsResource: Simulation of PlugsResource behavior without external dependencies TestGroupsModel: Controlled implementation of GroupsModel for resource testing The standardized design makes possible thorough evaluation of the GroupsModel and GroupsResource, which exist independently of external processes. The evaluations serve not only as a way of validation but also as documentation that clearly demonstrates the expected performance of said components within the overall IoT environment.

1.3 Deliverable 3

Q: Screenshot of the unit test report “Test Summary” page

Solution

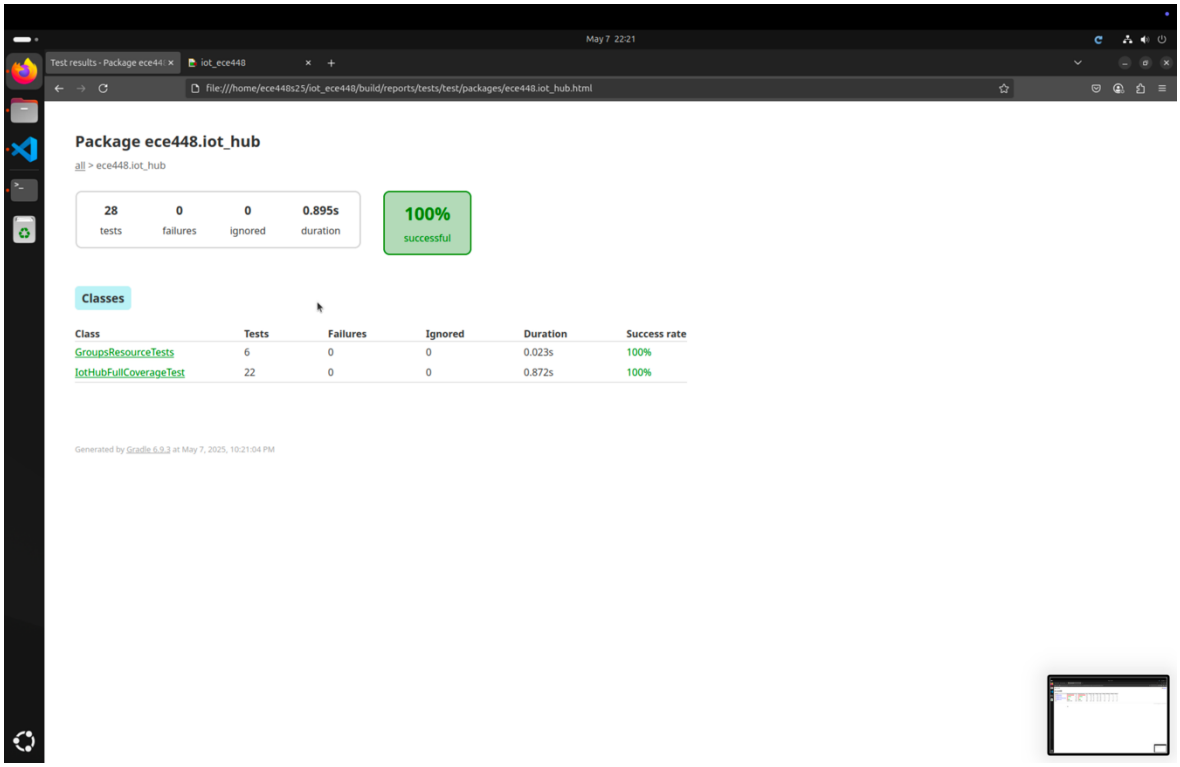

Figure 1: Screenshot of the Test Summary Page.

The above Figure 1 shows the “Test Summary” page screenshot with the added test cases for project 5. This shows the total number of test cases including the count from project 1.

1.4 Deliverable 4

Q: Screenshots of the coverage report pages for those classes you have added/modified. Please also report your average coverage among those pages.

Solution

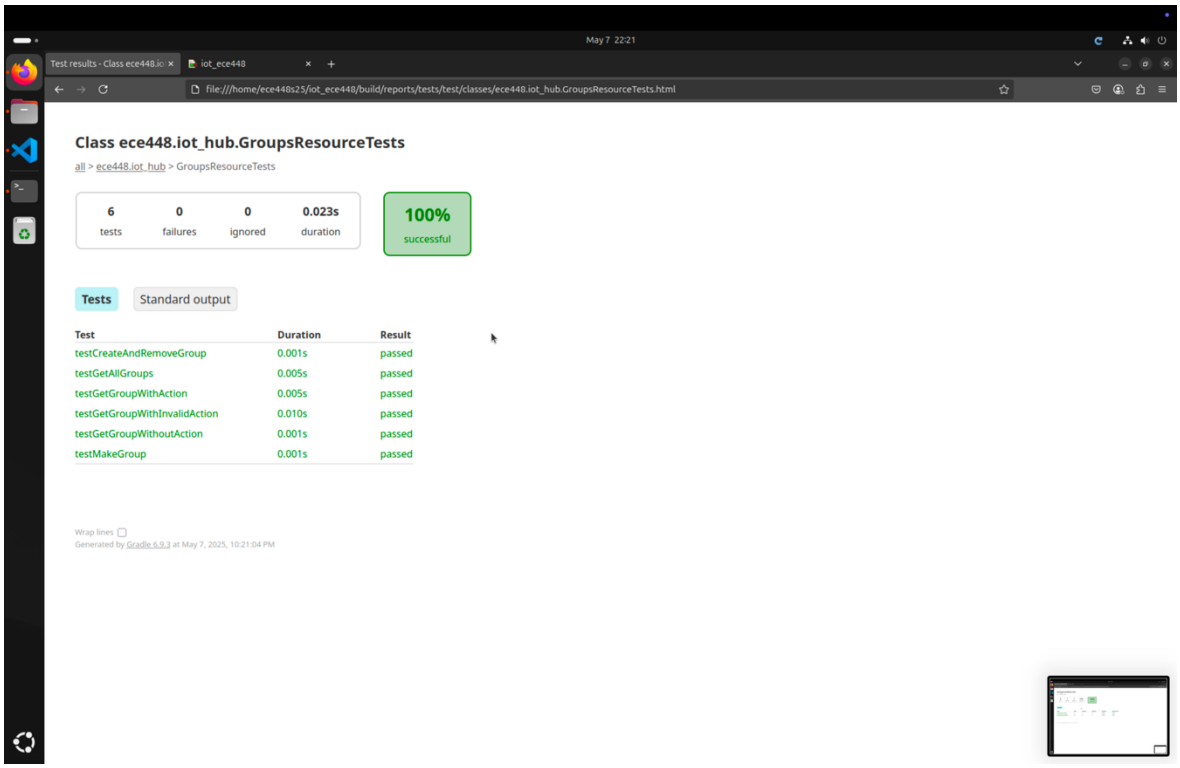

Figure 2: Screenshot of report of individual unit test cases added for testing.

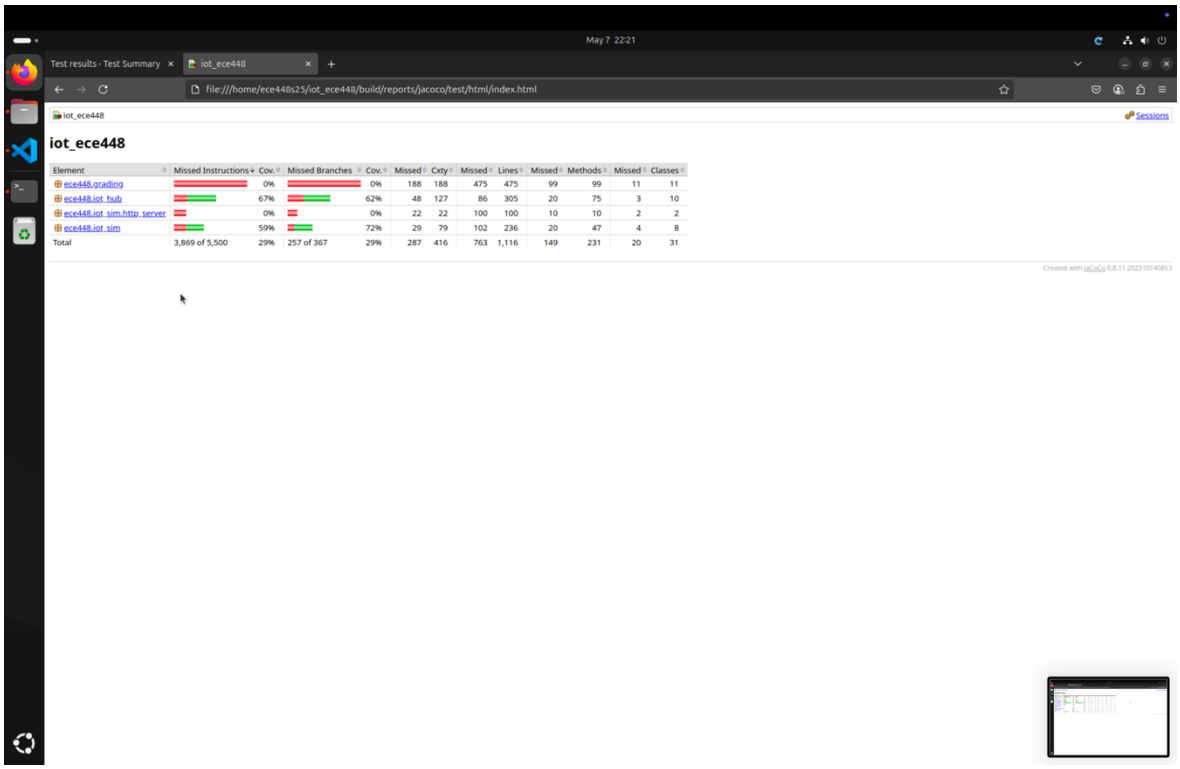

Figure 3: Screenshot of classes and their individual test report for overall.

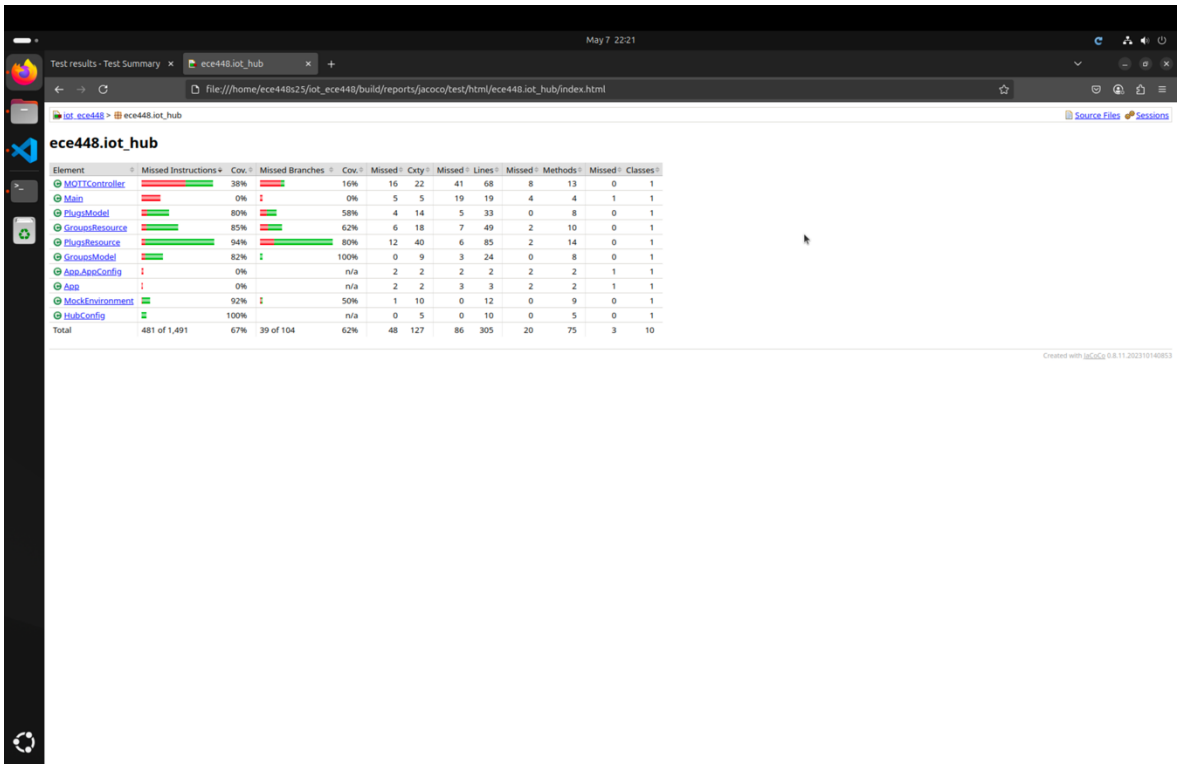

Figure 4: Screenshot of classes and their individual test report for iot_hub.

The coverage of the individual classes that were added or updated, the average unit test result, the pass percentage of the test classes added, and the amount of time it took for each to run are all displayed in Figure 2 through Figure 4 above.