### Report created January 23, 2023

**Green Dragon Group Test Summary Report**

Team Members: Amy Chan, Mark Ciochina, Gabriel Tefteller

# Step #1) Purpose of the document

This Test Summary Report document outlines the scope of tests performed on the Intra-Mural Sports Scheduling (IMSS) application and details their results and the status of the application as a whole. This document should serve as the basis for management and development teams to determine the appropriate next steps to take in product development.

# Step #2) Application Overview

The IMSS App (“the App”) is designed to manage recordkeeping and scheduling for the Intra-Mural program by providing functionalities to create teams, schedule games, and organize seasons. Users are allowed to create profiles that are broken into player, captain, referee and admin roles which all have different levels of access to interact with team management and scheduling functionalities. The app should allow each user role to interact with specified front-end elements through a browser, and have role-dependent access to alter database records.

# Step #3) Testing Scope

## Intended scope:

1. End-to-end front-end automated testing through a browser to make sure all required features specified in the Software Requirements Specifications (SRS) document are working as intended.
2. Unit test back-end service layer to make sure business logic for each function and user role performs the correct actions.
3. Perform integration tests on exposed API endpoints to determine that the service produces expected results when queried by a client.

## Out of scope:

1. Making alterations to existing application code
2. Altering database structure or functionality
3. Performing back-end unit tests on anything besides the service layer

## Items not tested:

* User acceptance of app
* Non-functional tests of app performance or User Interface (UI) design
* Unimplemented features included in the SRS

# Step #4) Metrics

## Test Case summary

|  | Planned | Executed | Passed | Passed With Restrictions | Failed |
| --- | --- | --- | --- | --- | --- |
| Front-end | 73 | 73 | 68 | 1 | 4 |
| Unit | 27 | 27 | 27 | 0 | 0 |
| Integration | 47 | 47 | 46 | 1 | 0 |
| Total: | 147 | 147 | 141 | 2 | 4 |

Table 1: Results of planned test cases. For additional details see Requirements Traceability Matrix

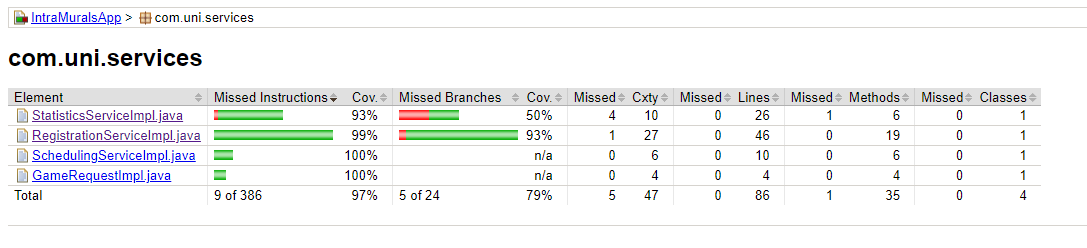
## Defect Summary

|  | Low Severity | Medium Severity | High Severity | Low Priority | Medium Priority | High Priority | Total |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Open | 4 | 2 | 0 | 1 | 2 | 3 | 6 |
| Closed | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Total | 4 | 2 | 0 | 1 | 2 | 3 | 6 |

Table 2: Defect priority and severity overview. For additional details see Defect Report

## Code Coverage:

Unit tests covered 97% of the service layer code with a slight amount of missed branches due to Mockito mocked behavior running in place of the intended functions in the DAO implementations.



# Step #5) Types of testing performed

## End-to-End testing - Java, Selenium, JUnit 4, Cucumber

End-to-End testing was performed using Java and a selection of testing tools to verify that users are able to perform the full range of required functions within the app through a browser (chrome specifically). Java code was written and executed using the Java 19 SDK. Selenium version 4.7.2 was used to automate browser interactions with the server and execute planned tests. JUnit version 4.13.2 was used to add testing-specific functionalities to Java that also integrated with Selenium WebDriver. The Cucumber framework, version 7.10.1, and Gherkin language were used to create feature files for all required feature functionalities and organize test planning and execution of JUnit tests.

## Unit testing - Java, Mockito, JUnit 5

Unit testing was performed using Java and a selection of testing tools to verify that the application’s business logic within the service layer is correct and calls the appropriate methods from each dependency. Java code was written and executed using the Java 19 SDK. Mockito version 4.11.0 was used to isolate the service layer components with mocked objects as dependencies allowing the testing team to determine which methods the service layer objects called from their dependencies. JUnit Jupiter, version 5.9.1, was used to provide testing-specific functionalities from the JUnit framework. DBeaver database manager was used to access the remote-hosted PostgreSQL Relational Database to view database organization.

## Integration - Postman, JavaScript, DBeaver

Integration tests were performed as black-box tests where the only interaction testers had with the application was through using Postman to send server requests to exposed API endpoints. Postman was used for writing, storing and executing http requests to the application hosting site. Postman was also used to store and test server responses to validate returned status codes and information.

Because no DELETE endpoints exist, DBeaver was used to perform pre-emptive database deletions in order for testing to be valid.

## Technology and Test Type Applications

|  | End-to-end | Unit | Integration |
| --- | --- | --- | --- |
| Java | X | X |  |
| Maven | X | X |  |
| Selenium WebDriver | X |  |  |
| Cucumber | X |  |  |
| JUnit 4 | X |  |  |
| JUnit 5 |  | X |  |
| Mockito |  | X |  |
| Postman |  |  | X |
| DBeaver |  |  | X |
| JavaScript |  |  | X |

Table 3: Presence-absence matrix for different test types and their associated tools

# Step #6) Recommendations

* Fix database role Enum to reflect captain functionalities.
  + While the functionality of the captain role exists, the way that captains are defined in the database causes potential complications with testing and debugging. For example, there is no way to make any user a captain without directly interacting with the database.
  + Verify no front-end role functionality is affected negatively by this change.
* **Implement restrictions on user credentials during registration.**
  + As it stands, there are no restrictions, save for the fact that two users cannot have the same username.
* Prevent games from being scheduled for a previous date.
  + Implement the date input in such a way that it prevents past dates from being selectable, and displays an error if past dates are entered manually.
* Implement features for the deletion of games/seasons/venues.
  + As it stands, games, venues, and seasons cannot be deleted without directly interacting with the database. There are no DELETE endpoints to speak of.
* Ensure that all endpoints return a relevant status code.
  + As it stands, the service returns 500 for most invalid requests, which is fairly nonspecific. The service could benefit from more relevant status codes.

# Step #7 Best Practices / Lessons learned

* Have a unanimous selection of ‘dedicated users’ (one for each user type) that are used by all team members
  + This avoids issues where one team member ends up modifying or even deleting a user that another team members needs for their tests to work.
* Establish a stable branching/merging git workflow as early as possible.
  + This minimizes latent merging conflicts and reduces the risk of lost work.
* Establish a general project directory structure as early as possible.
  + This similarly reduces merging conflicts later on, as well as inconsistent project structure.

# Step #8) Exit Criteria

Test Execution phase exit criteria were decided to be the completion of all intended tests, their results logged in the Requirements Traceability Matrix, and the details of discovered defects documented in the defect report.

# Step #9) Conclusion/Sign Off

The testing team has determined this application is **not recommended for production** in its current state.

Critical errors have been discovered that impede the user experience of the application, with code maintainability also called into question if current issues are not resolved. The defects of greatest concern are found in the integration tests, which prevent successful interaction between the back-end code and the database (see defect report for comprehensive details). Specifically the allowed database types should be updated to allow a visible ‘captain’ role for users, and the back-end code updated and regression tested to verify compatibility with the new database structure.

Functional issues have also been discovered in the end-to-end testing whereby one user can register with a blank username, and any user can register with a blank password as the database does not require Not Null values for these fields. The front-end interface also does not enforce validation of user input. Games may also be scheduled for past dates which should be strictly disallowed.

Non-functional tests should also be performed to measure application performance as well as User Acceptance tests. The user interface contains non-functional errors such as two ‘Back’ buttons on the register page and the API design is not entirely RESTful (“.html” endings for resources are common in the URLs). Once these problems are addressed the team believes the software will be nearly ready for production.