David Ziegelbauer

ID: 000496966

Testing Summary

D424 Task 3

Table of Contents

[Test Plan 2](#_Toc157170589)

[Unit Test Scripts 3](#_Toc157170590)

[The Pipeline 3](#_Toc157170591)

[The Unit Tests 4](#_Toc157170592)

[Unit Test Results 7](#_Toc157170593)

[Summary of Changes 8](#_Toc157170594)

[Unit Testing 8](#_Toc157170595)

[User Testing 8](#_Toc157170596)

# Test Plan

For this project I’ve used a two-pronged testing approach. First, I’ve implemented automated unit testing to ensure that changes do not have unintended consequences and that code performs the functions it is expected to. Second, I’ve enlisted the aid of test users to try out the application and point out flaws I haven’t noticed.

My unit testing strategy was to do as little manually as possible. I leaned heavily on automation built into my IDE, as well as a custom pipeline I built in AWS, to make it difficult to make changes without testing them. My workflow for changes has been as follows:

1. Make changes and/or refactor code.
2. Run unit tests locally in my IDE.
3. Make any corrections needed to fix failed tests.
4. Repeats steps 2 and 3 until all tests are passing.
5. Commit and push changes to my remote repository.
6. Create a pull request to merge the changes to the main branch.
7. Merge the pull request.
8. At this stage the automated pipeline runs the unit tests again and, if they all pass, deploys a docker image to my container repository.
9. Deploy the docker image to my cloud environment.

My second line of defense was user testing. Once the application was deployed to my cloud environment, I notified my testers that there was a new build ready. The testers logged into the application and worked through prescribed workflows such as creating a ticket, updating a ticket, and deleting a ticket. They logged and reported any issues they encountered including both technical (“things just don’t work”) issues and ergonomic (“I don’t like how this works”) issues. I then added these items to the backlog for the next build.

# Unit Test Scripts

## The Pipeline

The TicketPro pipeline is comprised of three stages. Each stage performs a distinct function:

A screenshot of a computer

Description automatically generated

Figure 1: Stage 1 - Source Code

Stage 1 obtains the application source code from the CodeCommit repository’s main branch.

A screenshot of a computer

Description automatically generated

Figure 2: Stage 2 - Unit Testing

Stage 2 launches a docker container with the source code and executes the *dotnet test* command to build and run the test project.

A screenshot of a computer

Description automatically generated

Figure 3: Stage 3 - Container Build

Stage 3 launches a docker container with the source code and runs through the buildspec.yaml build script to create a docker image and load it into the Amazon Elastic Container Repository, making it available for deployment.

## The Unit Tests

Inside the unit test project, there is a test class holding tests to run against the application business logic. In this project, the business logic is represented by the ITicketService interface and its implementation, the TicketService class.

A screenshot of a computer

Description automatically generated

Figure 4: Test Project

Within the TicketServiceTests class, tests are defined as public methods, as well as environment setup tasks in the Setup method.

A screenshot of a computer program

Description automatically generated

Figure 5: Partial View of TicketServiceTests Class

These tests can be run from within the IDE and are also executed when the *dotnet test* command is invoked.

A screen shot of a computer program

Description automatically generated

Figure 6: Example Test

In this sample test, we attempt to invoke the *CreateTicketAsync* method on the TicketService and validate that it correctly throws a NullUserException.

# Unit Test Results

The unit test project is run every time the application is changed before pushing a commit and making a pull request. Additionally, once the pull request is merged, the tests are run again for validation before deployment. As such, there is not one definitive set of test results to include here. The following figure is an example of the results feedback a developer would receive in the IDE following a test run:

A screen shot of a computer

Description automatically generated

Figure 7: Example Test Results

# Summary of Changes

## Unit Testing

An exhaustive summary of specific changes made on account of the automated unit tests would be of little value in understanding how the unit testing process results in application changes. Instead, I’ll describe it in a way generally applicable to all the changes.

Many times, throughout development, I found it necessary or useful to refactor code to make it more readable or support SOLID principles. While these changes seemed great in a vacuum, they occasionally broke functionality elsewhere in the application. The presence of these unit tests allowed me to identify these breaking changes before committing them. In turn, I was able to implement fixes and submit functionally correct commits.

## User Testing

Through user testing, I received direct feedback on the usability of the application’s UI and functionality. Changes made based on user feedback included:

|  |  |
| --- | --- |
| Feedback | Change |
| It doesn’t feel right to click <back> when editing a ticket and return to a fresh search page or the management list at page 1. | A session state functionality was implemented to allow the user to return to where they previously left off. |
| Test users identified a bug in the ticket editing functionality where they could not save an edited ticket. | I corrected a bug in the *UpdateTicketAsync* method on the TicketService and fixed the edge case in the unit test project to account for it. |
| Test users identified it was possible to create a ticket with an assigned technician but in an unassigned state and vice versa. | Business logic was adjusted to ensure this edge case was appropriately corrected. |