Canary: An end to end testing framework for Minestom

A Major Qualifying Project (MQP) Report
Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for the Degree of Bachelor of Science in

Computer Science

By:

Alexander Kinley

Matthew Worzala

Project Advisors:

Professor Joshua Cuneo

Date: April 2022

Abstract

Type abstract here.

Second paragraph of abstract

Acknowledgements

Begin list of acknowledgements

1. First Acknowledgement
2. Second Acknowledgement

- First Acknowledgement
- Second Acknowledgement

Contents

List of Tables

List of Figures

1 Introduction

Minecraft is a popular video game developed by Mojang studio and officially released in 2011. The basic premise is that the world is made up of many cube shaped blocks, and the player has the ability to break and place these blocks. Since its first release, Minecraft has grown into something much larger than this basic premise. One part of this growth is through custom multiplayer servers that allow for entirely new ways to play the game. Traditionally these custom servers have been accomplished by in some way modifying the default multiplayer server. This has been a very fruitful approach, and there are a massive amount of servers that take this approach.

Minestom is an open source library that re-implements the basic functionalities of a Minecraft server. The goal is to provide a strong foundation for people to build custom Minecraft servers on top of. Minestom is more performant, and far easy to extend than the default server, because it was built from the ground up with those goals. People who want to create a custom multiplayer gamemode or any other sort of custom behavior, can choose to build it on top of Minestom instead of using other tools that work by modifying the default multiplayer server. Since Minestom is a library that is aimed at people who want to implement custom functionality, providing a robust software development framework is important. Part of this framework could include test driven development.

Test driven development is a software development methodology where the software requirements are first encoded in test cases, before they are implemented. This means that you can continually test your software using the test cases, and have some reassurance that it is working correctly. This is commonly done using unit tests, which are tests of specific parts of the code like a class or function. This specificity is great for making sure that particular parts of a larger system are working properly. They are something that are simple to integrate into a java codebase using libraries such as JUnit. For when you want to test the behavior or functionality of an entire system, integration tests, or end to end testing are the solution. In other contexts this may take the form of automatically clicking through buttons on a form to ensure that it reaches the correct end state, or doing a predetermined set of inputs into a video game, and verifying that you end up where you expect to.

Minecraft presents a particularly interesting situation with regard to end to end testing, in that everything is modular and intended to behave the same in many different situations. You can manually test a feature by building some test structure, and then taking steps like spawning entities, or interacting with blocks, to test the feature. You would look for the expected behavior, and then be able to determine if the

feature works as expected. This is a workflow that can be automated. You can build your test structure and then save it. Then you can use the test structure along with programmatic equivalents of the additional steps like spawning entities, or interacting with blocks. Then you can have assertions about the behavior of the situation under test.

2 Background

Test driven development has become a popular technique in recent years, and as such this project is not the first to attempt to bring testing into the Minecraft world. Mojang itself has created an internal tool (named GameTest) for testing their game, and it serves as a great inspiration for this project. According to Henrik Knilberg (2020), a member of the gameplay team at Mojang, "". Ultimately, while Mojang's work serves as a good inspiration for creating tests based inside the game, it has some significant flaws for widespread use. It is not made for Minestom, and therefore makes a number of assumptions about Mojang features being implemented. Furthermore, the framework is closed source, so modifications are challenging and ineffective in a different environment, and cannot be posted publicly. In the category of modifications to the Mojang server, there have been two major attempts at testing: MockBukkit and McTester. Both take a programmatic approach to testing, and target different use cases. MockBukkit provides the user with an API for creating mocked resources such as players, worlds, and Bukkit plugins. This approach does allow for a variety of testing methods, however, since the tests are created programmatically, the setup can be expensive even for simple tests. McTester works by running a headless client and allowing the test to submit commands to the client, which can be asserted on server side. For example, a test could include a client right clicking a Command Block and asserting that the GUI was only opened if the player had appropriate permission. This method of end-to-end testing is effective for testing a server implementation (such as testing Minestom internally), however, it introduces unnecessary overhead for testing applications built on top of Minestom because the internals are expected to be tested and working independently. Finally, Minestom has seen its own attempt to create a testing framework internally. The test framework involved a mocked client connection to the server, from which a test could submit packets and make assertions on the received packets. Similar to McTester, this can be effective for testing internal features, however, it becomes cumbersome to work on a packet level when testing in userland.

Custom Minecraft servers have evolved throughout the years, starting from simple survival servers with extra enchantments, to game modes which look and feel like a new game completely.

3 Requirements

3.1 Objectives and Constraints

As previously discussed, this project aims to fill a perceived void in the minestom ecosystem for end to end testing solutions. We hope to create a library to be the de facto standard for end to end testing in the Minestom ecosystem. In order to meet this goal, we need to consider two main factors, ease of use and coverage of all major use cases. Ease of use means that developers using Canary shouldn't feel like they have to fight with the library in order to accomplish what they want. The library should be internally consistent and predictable, while handling the needs of developers. To cover all major use cases, Canary should be applicable to the scenarios where developers would want end to end testing.

To figure out what standard usage might look like, we can look towards the greater Minecraft server modding community to see what sort of features or functionality people have implemented within a Minecraft server.

Although the functionality and features that people add to Minecraft servers are broad and diverse, we can break them down into a few general categories based on what they require from an end to end testing library. The main factor in determining what a feature requires to be able to be tested, is the types of effects on the world it can have. We will look at three broad categories of functionality that can be added to a minecraft server.

Custom entity or block behavior

This category refers to most world behavior that is not linked to the player. This includes entity AI, entity properties, block properties, and how blocks interact with other blocks, and more. Generally these features only affect a small area around the block or entity, and happen based on the state of the world, and not players in the world.

Manually testing these types of features would generally involve building some sort of setup that will cause the desired behavior to occur, and then watching for the behavior to happen as expected. If testing, for example, you want to test that minecarts roll down hills correctly. You would first build a track down a hill, and then place a minecart on that track. Then you would watch for the minecart to end up at the bottom of the track.

Canary should allow you to define the setup for a test, as well as the expected outcome, and then execute the test on its own. The required functionality for this is: Be able to define the starting state of a

test, including the blocks, block properties, and entities. Be able to check if expected outcomes happened.

Determining if the expected outcome happened, in general, is a challenging problem. There is a wide variety of things that someone might look for to determine if a test ran correctly. Examples include, looking for an entity to get to a location, looking for a block to be in a certain state, making sure some condition did not happen, or checking for things to happen in a certain order. There is a further discussion of assertions in the needs section.

This category of feature is likely the one that benefits the most from end to end testing. Because most of the behaviors only cause changes in a small area, tests can be run simultaneously without having to reset the entire world between tests. On top of the functionality requirements, the process of making, running, and debugging these tests should be simple and straightforward to align with the overall goal of ease of use.

Custom server commands

Server commands are a way to use player messages like a command line or terminal. Custom commands are a common way to implement features like switching between worlds, primitive UI for server interaction, or a way for players to modify various aspects of the world. Server commands, unlike entity and block behavior, frequently affect things far beyond the player, including internal server state. To test server commands we should be able to Simulate a player from a known starting state in a known context executing a command Check for the expected behavior of the command

Server commands present a particular challenge in that their behavior is not bound in any way. A server command can do anything from changing a block near the player, to teleporting the player, to affecting other players, worlds, or the entire server. This broad scope makes it challenging to be able to effectively test all varieties of server commands, because creating a known starting state could potentially require restarting the entire server between every test, which would seriously impact the ability to quickly run tests.

Customizing player interaction

As discussed, changing what happens when a player interacts with the world is quite a common thing for Minecraft server mods to do. The types of changes that might be caused by these custom player behaviors frequently cover the full range of possible server behavior. Anything from combat, to inventories, to manipulating custom guis. Similar to server command, testing player interactions requires Simulate a player from a known starting state in a known context interacting with the world in some particular way Check for the expected behavior of the interaction

Player interaction often causes changes that can be challenging to test. A player clicking on an item in their inventory might teleport them to a different world, or a player clicking on a block with a certain item could cause that player to be sent a chat message.

3.2 Final Requirements

Over all, there are many challenges to creating a test system that covers all expected use cases. In particular, being able to test every possible server behavior in a consistent and reliable manner would force the overall library to be much less useful. For this reason, we chose to focus mainly on testing custom entity and block behavior features which can be tested in a finite and confined area. These features can require elaborate setups to fully test, and stand to benefit the most from automated end to end testing. That being said, we have not entirely ignored other feature types, and they could be the focus of future work, or other projects.

From this, we can create a list of the high level requirements of Canary. We have split the requirements into two categories, based on our original goals of ease of use, and full feature coverage.

Ease of Use

- Canary must be easy for developers to use, including creating tests, running tests, and debugging why a test might have failed.
- When possible, Canary should allow developers to create tests in a way similar to how they would when manually testing
- The code API used when writing code for tests should be understandable and abstract away common operations
- When a test case fails, Canary should provide error messages that help debug the problem
- Tests should be fast to run on a local machine
- Tests should be able to run in CI/CD servers in the same way other types of test might
- Any data for tests that is not code should work nicely with version control

Feature Coverage

- Canary must be applicable to all major use cases, in particular custom block and entity behavior should be able to be tested by Canary.

- Defining the expected outcome of a test should be able to express complex scenarios
- The things that can be asserted about should by default cover most common usage, and allow for easy developer extension for rare or custom properties

4 Needs

With the high level requirements determined, we now have to translate them into specific requirements to be implemented in software. To do this, we first need to decide on our overall approach to end to end testing. For this, we looked at prior work in end to end testing, as well as how developers approach manual testing. Our main inspiration was the work that Majang themselves had done. The key concept from this work is the combination of code with other data to create the "structure" associated with the test, and to use code to define any additional starting setup, as well as to create the assertions for the test.

This approach allows for the creation of the test structures to be done in Minecraft, the same way that someone would when manually testing. Along with this, it should reduce the amount of code needed for each test, further improving ease of use. The actual functionality of Canary can be split up into 3 broad categories that are responsible for all of the requirements.

4.1 Test Builder

As discussed, Canary tests have a test structure associated with them. This structure is not defined by code. Instead, it is other data that is loaded into the world when a test is being run. To make Canary easy to use for developers, we need to allow them to create these structures in the same way that they would when manually testing, by building them in Minecraft. The test builder is the subsystem of Canary responsible for letting users create and manipulate these structures. The test builder is a feature accessible on servers running Canary, and allows users to build structures in game, that will then be saved in a form that can be ingested when running tests.

The requirements of the test builder in general are to allow for creating all the necessary starting conditions for tests. The test builder should also be easy to use, and allow for all of the types of operations that would frequently be needed when making end to end tests. Such as making tests that have similar test structures. The created structures should also allow users to edit the properties of the blocks, either to change how they behave, or to mark them so their position can be referenced from the actual code associated with the test (mark a block with a name that can be used to easily get the position of the block).

Once a user has built a test structure in game, they should be saved in a format that can be read back in when running tests. Additionally, this format should have some amount of compatibility with git. Structure files should be text based so that minor changes to the structure will not cause git to see the whole fill as changed, and instead see a few lines out of the entire structure as changed.

4.2 Assertions

Assertions, along with the other API's involved in the code for each test, are perhaps the most integral part of Canary. The test builder is how a user defines the starting block configuration for their test, and assertions are how a user defines the expected behavior of the test. This system bears the brunt of the complexity in Canary with respect to being fully featured and easy to use.

As discussed previously, most other end to end testing systems require the programmer to handle the fact that their tests are most likely not fully deterministic due to factors such as frame rate. Minestom falls into this category, it's very possible to write code that produces slightly different output depending on things like exact frame timing. For Canary to be useful, we need to provide developers a way of specifying the expected behavior of their tests. To be reliable, this needs to be done without using methods such as directly comparing runs, or letting users prescribe an exact amount of time that something will take.

The solution proposed in this report is to allow for developers to create small programs that define the expected state of the world during the test. The types of things that can be expected would be that an entity makes it to a certain location, or that there is not a block at a certain position. Additionally, these sorts of assertions can be combined in various ways, such as with a logical and, or by saying that one should happen before the other. This solution creates many challenges, but ultimately it is the only approach that offers the needed capabilities while still fulfilling the other goals of the project.

4.3 Test Executor

The test executor is the core of Canary. Similar to other systems, test execution comes with its own set of challenges. These mainly have to do with the need for tests to not affect each other, while wanting the user experience of Canary to be as good as possible. So the main requirement is that a test cannot inadvertently change the results of another test. This would be a risk if tests were run together in the same world such that an entity might react to other blocks outside of the test that it is a part of.

While we want tests to be isolated, we also want it to be easy for a user to see groups of related

tests at the same time. This is a valuable feature both for development when you would want to make sure that all of the tests for a feature are working as expected, but also for when you want to see if groups of related tests are failing in the case of a bug. The way that we have decided to accomplish this is by grouping tests from the same test class together in rows in the world.

The final feature of the test executor is that it should be able to run the tests both locally, as well as on a CI/CD server. In both cases, there should be output that is helpful in the case of test failure.

5 Implementation

As established, writing assertions about events which have not happened, nor will happen in a deterministic manner creates a significant challenge. The solution proposed in this report is split into two distinct segments, roughly modeled after a typical interpreted programming language: assertions themselves (syntax), and the backing assertion nodes (AST). A typical assertion in Canary might look like the following:

```
// Listing AE.1
expect(myZombie).toBeAt(3, 1, 3).and().toHaveHealth(20.0);
```

From this list, the engine generates a tree of nodes, each of which has the ability to report whether it passed or failed depending on the condition (eg ‘subject.pos == ¡3,1,3¿‘) or the children. For example, the ‘AND‘ node has the following simple logic:
1. If any child is a `FAIL`, return ‘FAIL‘ 2. Return ‘PASS‘ This system is analogous to an Abstract Syntax Tree (AST) in an interpreter, and allows the test executor to simply check the root node to determine the result of the assertion.

5.1 Soft Pass

5.2 Assertion Specification

TODO: Finish in iA

5.3 Player Interaction Testing

TODO

5.4 Test Executor

The Canary test executor is responsible for managing the testing environment and complying with the JUnit test engine specification. When the engine is invoked, it first discovers every potential test case (method annotated with '@InWorldTest'). Each test is loaded into the appropriate Java ClassLoader (TODO: Talk about class loaders. It is relevant because we are targeting mixin Minestom. Could put this in an appendix and talk about the ClassLoader troubles as well as the compiler plugin that ensures nothing bad is happening), and a Minestom "instance" is created. An "instance" in Minestom is analogous to a Minecraft world, and allows the test to be completely isolated from all others. When the tests are executed, each test structure is placed into its respective instance, the assertion engine parses the step lists (Listing AE.2) into assertion trees. Finally, the test instances are ticked until it receives a definitive result. The stop conditions for a running test are as follows:

5.5 Sandbox Instance

Writing tests inside the environment they are testing provides the user a huge benefit, including real-time feedback. The user can enter the game and watch their tests running in real time, making it extremely simple to determine what is wrong. Canary facilitates this using the "sandbox instance", a Minestom instance containing every loaded test. This world allows the user to visibly see which tests are passing or failing, and fly around to view them.

5.6 Sandbox vs. Headless Mode

Canary has a strong focus on ease of use and visual feedback through the sandbox mode of the test engine. That, however, is not the only responsibility of a test engine. Test engines must be able to execute remotely in a Continuous Integration/Continuous Development (CI/CD) pipeline. Canary refers to this execution mode as "headless" mode, and it differs from the sandbox mode in a few ways. The running test server is not available to join, and the debug information is not loaded including commands and the sandbox instance. To ensure that all tests remain isolated and predictable, each test is still loaded into its own instance.

5.7 Isolation

It is critical to create the same exact environment for every execution of a test both locally on multiple distinct machines, or remotely as a part of a CI/CD pipeline. This task is non-trivial in the Minestom environment, since a test may access and use arbitrary server data, or interact with arbitrary blocks around the test environment. The most obvious solution is to use a unique Minestom server for each test which solves the global state and nearly interaction issues simply. Unfortunately, this solution has significant drawbacks in terms of speed and usability. Minestom makes use of some global (‘static‘ in Java) state for running server information, which means that you cannot easily have two Minestom servers running at the same time. It is possible to start a Java Virtual Machine (JVM) subprocess for each test, or to load each test into its own ClassLoader. Both results incur a large initialization overhead and memory increase for each individual test because every class must be reloaded each time. Regarding usability, separating each test into its own JVM or ClassLoader means that it is no longer possible to create the aggregate sandbox instance, destroying the ease of creating tests. Instead, Canary handles isolation by placing each individual test (and associated structure) into its own unique Minestom instance. This allows the sandbox instance to remain by forwarding actions from each sub instance into the sandbox. This solution, however, does not impose strict isolation for global server state. In the end, this compromise was worth the benefit provided by the sandbox instance. One of the benefits of thorough testing is that it forces users to write code which can be used in isolation, so this requirement is helpful in some ways.

6 Conclusion

Input your conclusion here.