Butler

Project Testing and Acceptance Plan

Schweitzer Engineering Laboratories

Horizon Group

Brendan Crebs

10/26/23

TABLE OF CONTENTS

I. Introduction

I.1. Project Overview

Butler is a multithreaded continuous integration (CI) tool written in Golang. The intention for Butler is to be a highly flexible application that developers can meld to their development environment. A development team can use Butler for running automated linting, testing, building, publishing, and fuzzing or any combination of those as a part of their build process. Butler is designed to be automatically run on a teams build server and will run whenever a build is triggered i.e., if code is pushed to the remote repository. After Butler has either succeeded or failed it will generate results to inform the user. Butler is a cli tool and can also be used locally. A major feature of Butler is the wide-ranging build in language support. Butler supports repositories containing most major languages without the need for must user contribution. In some cases, Butler may not have the necessary methods for a language or repository configuration. To handle for this, Butler provides a feature that allows users to inject their own methods into Butler's processes so that they can have maximum control over how Butler works in their repository.

In Butler the functionality most testing efforts will be put towards will first be each step of task construction. This includes determining code workspaces and dependencies, determining external third-party dependencies, and creating lint, test, build, publish, and fuzz tasks. Other major functionality to test will be multithreaded task execution and default, language specific built in methods. Lastly, it will be important to test the feature that allows users to supply their own methods. I must make sure that my method is general enough so that the user can supply their methods in whatever form they like.

I.2. Test Objectives and Schedule

The main objectives in testing Butler would include assuring high code quality and assuring that Butler is flexible enough to work in any environment a customer may attempt to use it in. Besides code reviews, assurance of code quality will be accomplished through our unit testing and CI process. The approach to unit tests will be a requirement of 100% test coverage and all tests passing before Butler can be considered for release. This will help assure that every part of Butler is ran and correctly exits with expected outputs. After a working prototype of Butler has been produced, it will be important for a CI process to be set up at this point. This will allow the continuous testing of the code as more features are added to Butler, hopefully catching any bugs when they are introduced. Assuring that Butler is flexible will be partially accomplished with unit tests but will mostly be achieved through functional testing. Functional testing will see us using Butler in a multitude of real and mock repositories to see how well it responds. This will be a highly important step before release.

To accomplish the testing goals, we will utilize several testing technologies. For unit testing we will use multiple third-party testing dependencies for Golang to assist testing. These will be "delve"[1] for debugging, "golangci-lint"[2] for linting, "gotests"[3] for assisting with table driven tests, and "gomock"[4] for mocking. In addition to this, the CI tool Jenkins[5] will be used for our CI process and will be run on a dedicated build server specifically for Butler. At first, we will manually include the testing locations and commands into a Jenkins pipeline. When a working version of Butler has been produced, these manual steps will be removed and Butler itself will be used as our CI tool and included as a step in the Jenkins pipeline.

Major testing deliverables for Butler will first include unit and integrations tests that achieve 100% test coverage over Butler. These tests will be included for the first time with the initial Butler prototype. From there on out, each version of Butler will require that all added features be tested with 100% test coverage before merges to the main branch can occur and a new version released. The next major deliverable will be a markdown document containing the results of functional testing. This document will be pre-prepared with a table of functional tests to run and what the results were. Much like the unit and integration tests, a 100% pass rate will be required with the functional tests for release to be an option. The next deliverable will be a build server and CI pipeline. For the pipeline, Jenkins will be used as discussed in the previous paragraph. The build server will consist of a container run on a virtual machine. The container on the VM will be the same container image we are using for development. The reason for this is to assure consistency between the two environments.

I.3. Scope

The intention behind this document is to outline the plans and processes that will be used in the testing of Butler. This document does not intent to document the results of the tests, nor will it go into exhaustive detail on specific tests. Rather, this document is meant to provide a reasonable strategy for testing Butler properly.

II. Testing Strategy

The approach to testing will have CI and functional testing as the main focus. The latest stable version of Butler will be used in the CI pipeline to test itself. For a branch to be merged to main, and for a new version to be released, Butler must pass all of the conditions for a successful build. These conditions will include, 100% off unit/integration tests passing, 100% code coverage, no linting errors, and a successful build task for Butler before publishing. If any of these conditions fail, then the entire build will fail, and we will be prohibited from merging into main or publishing a new version. For functional testing, all tests will be predetermined, and all tests must pass before Butler can be considered ready for release. These two processes will remain separate and are listed below:

CI Testing Process:

1. **New code is written:** The first step in the process will be writing code instead of writing tests.
2. **Unit tests written for new code:** After new code is produced there should also be new unit tests or updated unit tests to achieve test coverage over the new features. It will be good practice for these tests to include a large set of inputs and edge cases to cover a wide variety of possibilities. We want to avoid merely establishing coverage will bad tests.
3. **Check test results:** The unit tests will be run locally where they should be fixed if they fail or do not produce 100% coverage.
4. **Push to remote and trigger a build:** After a developer pushes code a CI build will be automatically triggered which will lint the code, run all tests including tests for code separate from the new feature, and perform a build task by running a dockerfile which

we will supply. If anything fails on the build server then the developer is to return to step 3 to investigate the issue. Here they will likely need to use the debugger to establish root cause.

5. **Perform Functional testing:** Perform functional tests and document the results. See "Functional Testing Process" for a detailed description of this step.
6. **Merge branch:** After a successful build, functional tests and approval from the reviewers, the branch can be merged into main. When a merge to main happens, this will trigger a special Butler build which will test everything in the repository, lint everything, build everything, and then create a special publish task to publish a new version of Butler. The publish task will be optional and will need to be manually enabled from the Butler config. Success of the merge will hinge on the success of this Butler build. If the build fails, then the branch will not be merged, and the developer will need to look to the Butler results to determine what needs to be fixed before another merge attempt can be made. Success of this step will satisfy the CI testing requirements.

Functional Testing Process:

1. **Get instance of working Butler version:** For functional testing, there first must have been a successful butler build. Functional testing will occur after step 4 in the prior process and before a merge attempt to main is made.
2. **Create functional testing document:** For functional testing, the developer must first create a markdown document that contains a table populated with functional tests in the first column. In the other columns there will be space for a description of the test, observations during the test, any inputs used, and outputs generated, and whether the functional test passed or failed.
3. **Perform functional tests:** The developer must run the functional tests to the specification of the test description. The user should attempt to also do abnormal actions to break Butler so obscure issues can be revealed.
4. **Document results:** The developer should document the results of the functional tests in the empty fields of the table. If any tests fail, then the new version cannot be released and must undergo revision. After cause is determined and a fix is produced, start back at step 1 and reperform functional tests for the new instance of Butler.

III. Test Plans

III.1. Unit Testing

For unit testing we will establish that individual functions will our unit to test. A unit test will be generated for every function, including helper functions. For each file, there will be an adjacent test file which will contain the unit tests for functions in that file. Our goal with unit tests is to provide the functions will a large and diverse set of inputs to detect any edge cases. The unit tests are where 100% test coverage will mostly be established. Here is also where we will check to make sure that Butler errors and returns when it is supposed to.

III.2. Integration Testing

The integration tests will work well with our microservice model approach. With these tests we will check to see how one component flows into the other. For example, we will test the lint component and build component in the same test to make sure nothing failed when they were included together. Since naturally, Butler flows from one component to the next, integration tests will be very helpful and not difficult to implement.

III.3. System Testing

III.3.1. Functional testing:

The functional tests will be performed extensively in a multitude of different repositories with diverse environments to assess how Butler performs. This process I believe will help make Butler much stronger and more flexible. This effort is to expand the usefulness of Butler and reduce the chances that it will fail in an environment which we didn't expect. The steps that will be taken during function testing are listed under the "Functional Testing Process" in Testing Strategy above.

III.3.2. Performance testing:

One of the performance testing practices we will adopt will be fuzzing. Fuzz testing is the practice of feeding an application enormous amount of randomly generated inputs in order to break the application. Other measures will be generation unreasonably large repositories with and unreasonable number of tasks which we will continue to increase until Butler breaks. During this time, we will look for how Butler's performance suffers and how much memory it is using under extreme circumstances.

III.3.3. User Acceptance Testing:

For user acceptance testing, prototype versions of Butler will be distributed to the primary customer which is my team and my manager so they can assess if Butler is adequately meeting their requirements. This practice will occur starting at the first prototype release all the way until the project has entered a maintenance phase.

IV. Environment Requirements

The testing environment will require installation of Golang, all third-party testing tools, and a go.mod file containing the third-party dependencies used in the code. The testing environment will also require a build server and a computer to host said build server. On the build server and locally, a docker container will be used. To use the containers docker and the devcontainer cli will be required. For the CI pipeline Jenkins will also be used. It is expected the all tests are run in a Linux environment. This can be wsl, a VM, or on a machine booted with any major distribution of Linux.

V. Glossary

Continuous Integration: Practice of frequently building and testing code as it's being developed.

Fuzzing: A type of testing that feeds an application enormous amount of random inputs in order to cause bugs or reveal vulnerabilities.

Dockerfile: A file that defines configurations for a docker image.

VI. References

[1] Go-Delve. “Go-Delve/Delve: Delve Is a Debugger for the Go Programming Language.” *GitHub*, github.com/go-delve/delve. Available: https://github.com/go-delve/delve Accessed 26 Oct. 2023.

[2] Golangci. “Golangci/Golangci-Lint: Fast Linters Runner for Go.” *GitHub*, github.com/golangci/golangci-lint. Available: https://github.com/golangci/golangci-lint Accessed 26 Oct. 2023.

[3] “Gotests.” *Gotests Package - Github.Com/Cweill/Gotests - Go Packages*, pkg.go.dev/github.com/cweill/gotests. Available: https://pkg.go.dev/github.com/cweill/gotests Accessed 26 Oct. 2023.

[4] Golang. “Golang/Mock: Gomock Is a Mocking Framework for the Go Programming Language.” *GitHub*, github.com/golang/mock. Available: https://github.com/golang/mock Accessed 26 Oct. 2023.

[5] *Jenkins*, www.jenkins.io/. Available: https://www.jenkins.io/ Accessed 26 Oct. 2023.