Continuous Integration

James Smith
Amanda Ling
Fran Medland
Hannah Vas
James Kellett
Malik Tremain
Mischa Zaynchkovsky

Methods & Approaches

Continuous Integration is a software development practice where each member of a team merges their changes into a codebase ... daily. Each of these integrations is ... to detect integration errors as quickly as possible.[1].

A large amount of the application of valuable continuous integration comes from using the appropriate and best considered practices, firstly I will outline the inferential approaches we implemented in the ENG1 assessment 2 as were outlined by our team:

1. Team wide visibility of the complete code base
2. Simple availability of the latest version both for the development codebase and for the latest release
3. Agreement upon codestyle and architecture
4. Broken build countdowns

These simple methods can be justified as follows: 1.complete visibility of the code base affords those working on the code base the opportunity to verify state and implementation details of any feature they will be integrating alongside. 2. This covers the side of continuous integration with relation to delivery of a product as stable and dev releases allow for the development team to test holistic features and the end user to view the status of the product that they are funding. 3. This is a vital feature in all continuous integration pipelines as it leads to an increase in code consistency and therefore reduces the time spent having to integrate features, the number of issues that show up when doing so and comes at little cost in a team that is already implementing a form of team workflow such as agile scrum, which our team has chosen. 4. Policy wise if a build is committed but found to be broken, for example if it makes some tests fail, then it should be reverted or fixed within 10 minutes to prevent others checking out any non functional code.

The theory behind continuous integration is that such practices will reduce time spent and issues faced in the integration process. The traceable continuous integration methods we have used are as follows:

1. Automatic builds on commit and push to main
2. Automatic upload of build artifacts on commit and push to main
3. Testing automation on commit and push to main
4. Replication on clean environment on commit and push to main
5. Releases on push to main with a tag of the form: ‘v[0-9].[0-9].[0-9]’

Each of these that we have used are represented in the github actions section and are covered in the infrastructure section. These methods are justified by the small scale of the project allowing for constant artifact releases and across the board it is best practice to do ‘clean’ tests on a non development environment.

I would put forth that the pipelines and approaches that we have deployed go beyond that of continuous integration and reach the realms of continuous delivery, this is because the main branch always remains merged with latest commits and beyond that we have pipelines in place that mean broken code or code that does not pass tests never remains on main leading it to be in a deployable state at all times. Where continuous delivery is defined as follows:

Continuous Delivery is the ability to get changes of all types ... into production, or into the hands of users, safely and quickly in a sustainable way ... We achieve all this by ensuring our code is always in a deployable state. [2]

CI Infrastructure

The one prerequisite for a developer committing to the mainline is that they can correctly build their code. This, of course, includes passing the build tests. As with any commit cycle the developer first updates their working copy to match the mainline, resolves any conflicts with the mainline, then builds on their local machine. If the build passes, then they are free to push to the mainline. [1]

The major implementation of our continuous integration pipelines is through the github actions features. Such pipelines are permitted by our approaches and the implemented systems such as ‘jacoco’ for testing and 'gradle’ for build purposes. The pipelines are as follows:

1. Pushes to the main branch with tags of the form ‘v[0-9].[0-9].[0-9]’ represent a version for release using the typical semantic versioning system of v representing a version, major versioning number which implements incompatible API changes, minor version number which implements backward compatible changes, patch version which implements bug fixes [3]. Any push to main with a tag of this form will create a release on the main branch, the release will contain a runnable jar file of the codebase in the state it is in upon push. This is an example of automatic deployment.

2. Gradle based build: we run our build system, including testing, on a clean ubuntu based hermetic environment (an ‘integration machine’[1]) which is closed and separate from that of any development environment which allows us to sidestep the debate of ‘well it runs on my machine’ if we encounter bugs or errors. This is another automatic deployment methodology.

3. Gradle runnable jar distribution. For each push to the main branch the gradle build also creates a runnable distribution of the version which is important as it makes production builds readily available without the need to clone the repository and locally build the same file. The relevant output is uploaded to the actions tab as an artifact readily downloadable.

4. Per commit to main testing readouts, we decided to include 2 testing reports, that being the default one produced by a gradle build and the more detailed top down view provided by JaCoCo. This is because the grade test report is more simple to view and consistent commits to main should only require you to view this to debug however upon the occasion that a larger set of errors occur then the JaCoCo report can provide an in depth testing report including code snippets related to test failures. Both of these reports are uploaded to the actions tab as separate artifacts making them accessible at any point.

5. We maintain a homogenous codebase on the main branch containing every configuration file, piece of code and class file including everything that would be needed to make a clean build of the most recent codebase from scratch. This is the concept of homogeneity which is very important to all continuous integration.

The most current representation of these workflows can be found using the repository gradle.yml file which can be found in the references section [4].

References

[1]https://martinfowler.com/articles/continuousIntegration.html as of 02/05/24
[2]https://continuousdelivery.com/ as of 08/05/24
[3]https://semver.org/ as of 10/05/24
[4]https://github.com/WaddleWareStudios/HeslingtonHustlePrivate at .github/workflows/gradle.yml