ENG1 Assessment 2 Report

Cohort 3, Group 23

Deliverable #6: Continuous Integration

Oliver Dixon
od641@york.ac.uk

George Cranton
gc1263@york.ac.uk

Praj Dethekar
pd910@york.ac.uk

Denys Sova
ds1855@york.ac.uk

Shivan Ramharry
sr1907@york.ac.uk

Albara Shoukri
aams508@york.ac.uk

Rafael Duarte
rd1395@york.ac.uk

Department of Computer Science
University of York

Semester 2, 2023/24

Overview of CI Methods and Approaches

Throughout the Integration and Extension component of the Assessment 2 cycle, numerous Git branches diverging from the mainline were maintained, in which new features could be developed and tested in isolation from any other unrelated development work; for example, a branch “leaderboard” was used to prototype and implement the Leaderboard requirement, and a branch “score-achievements” was used to prototype and implement the Scores and Achievements/Streaks requirements; these are henceforth referred to as *feature branches*.

To prevent feature branches diverging to such an extent that they can no longer be easily integrated into the code base, each core developer was instructed to periodically commit to the mainline branch, minimally once-per-day. Automated testing and build automation were key tenets in the CI processes: a single, isolated, and centralised *build machine* [1] (detailed in the following section) was used to ensure reliable building and automatic testing of the project in a clean environment, that is nominally a clone of the production environment.

Given the principle of mandatory regular commits to the mainline, it was deemed unnecessary to run automated build and test processes on the feature branches, as this would increase the amount of billable time spent on the virtual build machine and would not aid integration; Martin Fowler refers to this semi-integration as *Continuous Build Services* [2]. Conversely, a full build, test, and style-check routine was run on every push to the mainline, ensuring that it is kept in a suitable state with reference to the build machine’s environment. Caching of toolchain components and build-time dependencies was used on the build machine to ensure a fast build, in line with classical Extreme Programming (and hence CI) principles [3].

Automated deployment and packaging was also used to generate JAR files, including any run-time dependencies, upon the pushing of a tag to mark a new release according to the established versioning schema. This was appropriate for the project to ensure that recent binaries are available to interested parties, but are not generated on every commit; given the frequent and regular daily mainline commits (on which build and test actions are always executed), generated binaries would cause unnecessary strain on the storage and processing capabilities of the infrastructure hosting the build machine.

CI Infrastructure Review

GitHub Actions was selected as the CI system due to its integration with the existing version-control system coupled with the extensive documentation and options for a GitHub-supported hosted build machine [4]. Discrete CI servers were considered, but due to the existing GitHub framework being in place with sufficient IDE integration, *Actions* were deemed as the most appropriate hosted CI model. Furthermore, due to the limited execution time available on the cloud-based supported build machines, a custom build environment was configured, deployed, and isolated on a University of York server (`csteach0`) on which unlimited GitHub Action build processes could be run.

There are two major GitHub Actions workflows being used on the repository:

1. The 'push' workflow, used to check-out the HEAD (latest commit) of the mainline branch and execute a full Gradle build in the isolated environment, having cached build dependencies from previous Action runs as appropriate. All automated tests are executed against the build project, and a Checkstyle report is generated to identify any potential style violations, determined by a modified variant of the canonical Google Java coding standard. The reports from JUnit testing and Checkstyle are then attached to the GitHub Action window; if an automated test fails, then the entire run fails and the Software Manager (Oliver Dixon) is notified via e-mail.

   If the push was on a regular branch, in this case restricted to the mainline, then the above process is performed and Action run completes or fails as appropriate. If the push was on a tag, indicating a new release, a full JAR is packaged and attached to the tag release page. The following GitHub Action packages were used in this workflow:

   - `actions/checkout@v4`: Checkout sources
   - `actions/setup-java@v4`: Setup a JDK (11) building environment
   - `gradle/actions/setup-gradle@v3`: Download Gradle for the wrapper
   - `actions/upload-artifact@v4`: Upload 'build artefacts' to the Action report
   - `lcollins/checkstyle-github-action@v2`: Upload Checkstyle reports
   - `softprops/action-gh-release@v2`: Upload the JAR to the tag release

2. The 'pull request' workflow, used to assess the quality of the commits in a feature branch requesting to be merged into the mainline. This workflow builds and tests the project akin to the 'push' workflow, but also uses JaCoCo and its associated Gradle plugin to generate a coverage report of the tested code according to the changes made in the pull request. For this, the `Vadgyik/jacoco-report@use-node20` workflow typesets and attaches the JaCoCo results as a comment to the pull request.

   Due to the rapid rate of non-PR-initiated commits to the mainline, it was deemed unnecessary to generate coverage reports for all mainline pushes, as features would often be integrated before having achieved comprehensive testing. However, baselines of coverage are enforced before creating a new tag and subsequent packaged release.

References

[1] P. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*. Upper Saddle River, New Jersey, U.S.: Addison-Wesley, 2007, p. 12.

[2] M. Fowler, "Continuous Integration," *martinFowler.com*, Jan. 18, 2024. https://martinfowler.com/articles/continuousIntegration.html (accessed Apr. 30, 2024).

[3] K. Gallaba, Y. Junqueira, J. Ewart, and S. Mcintosh, "Accelerating Continuous Integration by Caching Environments and Inferring Dependencies," *IEEE Transactions on Software Engineering*, vol. 48, no. 6, Dec. 2020, doi: https://doi.org/10.1109/tse.2020.3048335.

[4] T. Kinsman, M. Wessel, M. Gerosa, and C. Treude, "How Do Software Developers Use GitHub Actions to Automate Their Workflows?," *IEEE Xplore*, pp. 420–431, May 2021, doi: https://doi.org/10.1109/MSR52588.2021.00054.