CS 130 Hub

Assignment 3

Your team will build unit tests and integration tests for your webserver. You will learn how to measure test coverage to help you ensure that you are covering as much of your code logic as possible. You will also learn how to set up a continuous build for your repository.

Each student should submit this assignment by 11:59PM on April 25, 2022 into the submission form.

TABLE OF CONTENTS

1 Set up a continuous build
2 Measure test coverage
3 Integrate test coverage into continuous build
4 Write web server unit tests
5 Write web server integration test(s)
6 Grading Criteria
7 Submit your assignment

Set up a continuous build

Before you create tests for your webserver, you should set up a continuous build on Google Cloud Build. This build can be set up to run the build steps of a cloudbuild.yaml configuration every time a change is submitted to a git repository. The built-in triggers support monitoring Cloud Source Repositories, but our code is in our private Gerrit. In order to make the triggers work, you can run a command on Gerrit to mirror your Gerrit repository to a Cloud Source Repository within your project. Let ${PROJECT} be your Project ID used in the Cloud URLs and `gcloud` commands, and ${REPO} be your repository in Gerrit. The following command will create a new Cloud Source Repository in ${PROJECT} and set up mirroring for the repository ${REPO} in Gerrit:

```
$ ssh -p 29418 ${USER}@code.cs130.org gcp-mirror init ${REPO} ${PROJECT}
```

When this is done, you should see a copy of your code in a new repository at Cloud Source Repositories. Going forward, code changes will be mirrored to your Cloud Source repository approximately 15 seconds after submission through Gerrit. Think of this copy as a read-only backup copy. You will *not* be making changes directly to this repository, and any changes you did make would be overwritten on the next mirror from Gerrit. In other words, all code is still submitted through Gerrit.

Once you've set up mirroring, set up a new Cloud Build Trigger to build and test your Docker container every time a change is submitted to your `main` branch:

- From the **Triggers** page, click **Create trigger**.
- **Name** your trigger something like *${REPO}-main-submit*.
- Choose your project repository from the **Repository** dropdown under **Source**.
- Keep *^main$* from the **Branch** dropdown to exactly match the `main` branch.
- For **Cloud Build configuration file location** enter the path in your repository to your `cloudbuild.yaml` (probably *docker/cloudbuild.yaml*).
- Click **Create** to complete creation.

Once the trigger has been created, you can test it by clicking **Run** in the table on the **Triggers** page. If you switch to the History page, you can see a list of all the builds submitted to or triggered by Cloud Build.

Successful builds appear with a green check icon in the table. Clicking on a build hash in the table will show you **Build details** for each build, including the **Build log** from each of the **Steps** of the build. For successful builds, if you look under **Build artifacts** you will also see that new `:latest` images are created and tagged, which means a simple restart of your running server in Compute Engine will pick up the newly built image (assuming it is set to use the `:latest` image in your VM instance settings). If you want to pin your running server to a specific version, such as `:stable` or `:assign3` then you should tag an image in Container Registry, and update your VM instance settings to load the image with that tag instead of `:latest`.

If in the course of development you submit code that breaks your tests, you will see a failure in the test step of the corresponding continuous build and the build will terminate without creating a new `:latest` image. This prevents bad code from getting pushed to your server, and is a major reason why you would setup this continuous build in the first place. The build history should help you pinpoint when your tests began to fail, which in turn should help you figure out which code change broke your tests, so that you can fix them.

Measure test coverage

Next you'll set up testing code coverage, to measure how much of your code is covered by tests. Build your project with test code coverage reporting enabled to generate a test code coverage report.

If you examine the build logs, you will see a couple lines after **Generating coverage report** that show your coverage in relative and absolute numbers for line and branch coverage. These numbers, especially the absolute numbers, will probably be somewhat low.

If you examine `${REPO}/build_coverage/report/index.html`, you will see a more detailed coverage report, including file-by-file, line-by-line coverage information. As you add tests to your codebase, more of your core code will be exercised, and you will see more entries in your generated coverage report with a corresponding higher percentage of test code coverage.

Integrate test coverage into continuous build

Since you already have Cloud Build running your tests on every code submission, it would be helpful to also generate the test coverage report so you can track test coverage over time. While it is a bit complex to extract the actual HTML report files from the continuous build, it's quite easy to just build the coverage report and have the summary appear in the build logs.

To accomplish this, you will create a new Docker container specifically for coverage, and a new cloud build step to build this docker container:

- Create a new file `docker/coverage.Dockerfile` for your coverage build container.
  - The container should resemble the `builder` stage of `docker/Dockerfile`, but should generate the coverage report instead of just building and running tests.
  - Since generating the coverage report requires `gcovr`, and you are inheriting from / building upon your base container, you should add `gcovr` to the packages installed in your `base.Dockerfile`.
- Create a new step in docker/cloudbuild.yaml to build your new container.
  - This step should probably go last, since you only need to measure test coverage if your tests pass, which are running through your main `Dockerfile` in earlier build steps.

You can test this works by submitting a cloud build and checking the output logs in the Cloud Build History. If all goes well, you will see your test coverage summary in the build logs of your final build step.

Write web server unit tests

Now that you can measure your test coverage, you should add or improve unit tests for your web server.

If you haven't already, you should refactor your server into multiple classes, with an `.h` file for each class interface and a `.cc` file for each class implementation. Add a separate `_main.cc` source file containing your `main()` method. Then, for each class, create a `<name_of_class>_test.cc` file in `tests/`, with unit tests for that class. Your `main()` method should be trivial enough to not need unit tests.

General advice for tests:

- Use dependency injection to test code in isolation. (Refer to Lecture 2 and 6)
- If you find your code is difficult to test, refactor it to improve testability.
- Make sure to integrate all your new tests into `CMakeLists.txt`.

When you're satisfied with your refactoring and tests, check your Cloud Build History to see how your test coverage has changed over time. Hopefully, your recent reports should have higher coverage. Groups with very thorough tests may receive extra points.

Write web server integration test(s)

Finally you should write an integration test for your web server in Bash or Python. The integration test script should do at least the following:

- Run the full web server binary with a test config file
  - The config file can be either a static file, or dynamically generated by the script. At this point it will probably only need to configure the port.
  - *Hint:* Learn how to run a background task in `bash` if using Bash.
- Send a request to the web server.
  - You can use `curl` to issue the request, and capture the output. See documentation for the `-o`, `-s`, and `-S` flags.
- Verify that the response from the web server is as expected.
  - The `diff` command helpfully uses exit codes to indicate file equivalency.
  - *Hint:* `$?` is the exit code of the last run command in `bash`.
- Shut down the web server.
  - *Hint:* `$!` is the PID of the last run command in `bash`.
- Exit with an exit code indicating success (0) or failure (1)

You can integrate your test with `ctest` / `make test` by adding an `add_test` rule to `CMakeLists.txt` as suggested in lecture 4. Doing this will also incorporate your integration test into test coverage analysis, since the test will run as part of the normal `ctest` step.

If you want to expand your integration test, you could:

- Test your server's handling of invalid requests, which you can make with `nc`.
- Structure your Bash/Python script to make it easy to add new tests, without duplicating server set up or clean up code.

Things to think about:

- Does your server need to stop and restart between each test case? What are the pros/cons of doing so?
- Is your server getting killed even when there's a test failure? Put another way, is there ever a (normal) situation in which your server will be stuck running in the background after your test is done? Test failure is considered a "normal" situation.

Once you're satisfied with your integration test, and your measured test coverage, you've finished the assignment.

Grading Criteria

The minimum requirements for this assignment are drawn from the instructions above. The team grading criteria includes:

- Deployment of a Repo mirror on GCP
- Setting up build triggers for continuous integration tests
- Good coverage (more coverage translates to more points, bonus accrues above 90)
- Integration tests must be present and correctly detect failures
- Refactoring to make code more testable

Individual Contributor criteria includes:

- Code submitted for review (follows existing style, readable, testable)
- All comments from TL were addressed and resolved

Tech Lead criteria includes:

- Assignment tracker completed and kept up to date

- Maintained comprehensive meeting notes
- A thoughtful (more than just a score) review was given to team members

Additional criteria may be considered at the discretion of graders or instructors.

Submit your assignment

Everyone should fill out the submission form before 11:59PM on the due date. We will only review code that was added to the `main` branch of the team repository before the last change referenced in the submission form. You may update your submission by re-submitting the form, which will update the submission time considered for grading purposes.

"You may think using Google's great, but I still think it's terrible." —Larry Page

Page last modified: April 18, 2023.