Gradescope Autograder Configuration

Matt Russell

May 21, 2022

Introduction

Gradescope is great tool for autograding assignments. However, there is still a substantial amount of infrastructure required to deploy and run an autograder on gradescope. This document provides instructions for both setting up autograders on Gradescope, and for using our in-house autograding framework for `C/C++` code. Setup from start to finish is intended to take roughly 30 minutes. If you have any questions, please reach out to me at `mrussell@cs.tufts.edu`. Thanks!

Infrastructure Background

Gradescope's autograders rely on `Docker` containers which are spun up each time a submission is graded. The default container runs a variant of `Ubuntu 18.04`, coupled with the bare-bones scripts to make the autograding framework function. There are two supported workflows in this document to integrate with this system.

1. The `.zip` method - this workflow is to manually upload a `.zip` file containing two scripts `setup.sh`, which installs dependencies (e.g. `Python`, `clang`, etc.), and a shell script named `run_autograder`, which runs the autograder.

2. The `Docker` method - this workflow is to build the `Docker` container from scratch and upload it to `Dockerhub`.

Pros and cons of these approaches:

- The `.zip` method requires more manual work. You have to upload a new `.zip` file each time you want to update the autograder; the `Docker` container will then be built from scratch on Gradescope, which takes time. However, you don't need `Docker` on your system. If you don't use `Docker`, this workflow is suggested.

- The `Docker` method is more streamlined once it's setup. After uploading the container, for every assignment, you can point Gradescope to the container on `Dockerhub` - no `.zip` file uploading required. And, if you make minor changes to the setup script, usually rebuilding the container is very fast. All of the steps to do the building and deploying of the container are done in a script for you. One drawback of this approach is that you need the `.dockercreds` file - this file contains a deploy key for the `tuftscs` account on `Dockerhub`. The key is not in this repository as it's public; it should be in Tufts Box - if not, email me at `mrussell@cs.tufts.edu`.

Autograding Background

Once the container is built, there is of course the issue of how to run and test student's code. This is no easy task! However, this document includes documentation on an autograding framework we have developed which makes writing tests for student code as easy as possible.

Infrastructure Setup

Autograding .git Repo

Regardless of whether you use the `.zip` method or the `Docker` method, you will need to create a `git` repository for your autograder. This repository will be used by the autograder; each time an autograder instance runs, the code from the repository will be pulled, so the latest version of the grader can run. To that end, if you don't currently have a repository related to course material, please make one. We suggest using `gitlab` for this: go to https://gitlab.cs.tufts.edu, and login with `LDAP`, using your Tufts eecs `utln` and password. You do not need a `README`. The example below will be for `cs 15`, but please follow the instructions for whichever course you're running. Now, in your terminal:

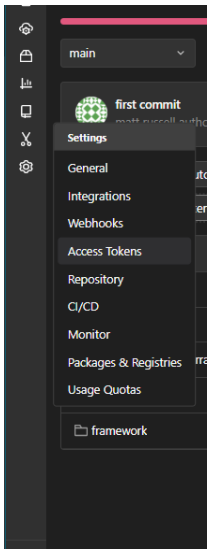
```
mkdir cs-15-autograding
cd cs-15-autograding
git init
git remote add origin git@gitlab.cs.tufts.edu:your_utln/
    path_to_your_repo.git
git switch -c main
```

We have a sample repo for you to start with. This repository contains:

- Files both the `.zip` and `Docker` methods
- A sample autograder for the first cs-15 assignment.
- The autograding framework.

Copy the files as follows:

```
git clone git@gitlab.cs.tufts.edu:mrussell/gradescope-
    autograding
rm -rf gradescope-autograding/.git
mv gradescope-autograding/* .
rm -rf gradescope-autograding
```



Great! Now, you will need an Access Token so your autograder can pull from the repo. To create one, go to `gitlab` in your browser, and navigate to the course repository you just created. Next, hover over the settings cog on the lower left, and select 'Access Tokens'. Create an access token; this will be used by the Gradescope autograder to pull the most recent version of the autograding files for an assignment. We suggest only providing 'read repository' access to the token. Feel free to select whatever you'd like for the name, expiration date, and role (Maintainer is fine). Once the token is created, copy the key. Now, open a file and type the following (we will need it later):

```
https://REPOSITORY-NAME:ACCESS-TOKEN@gitlab.cs.tufts.edu/path/
    to/repository.git
```

For example:

```
https://cs-15-2022uc:glpat-Blah8173Blah8023Blah@gitlab.cs.tufts
    .edu/mrussell/cs-15-2022uc.git
```

Okay! Now, continue with one of either the `.zip` or `Docker` methods below.

.zip Method

As mentioned above, with the `.zip` method, you'll need to upload a `.zip` file for each assignment. To get this working, you'll need to open the file `zipbuild/setup.sh`, and replace the `REPOPATH` string at the top of the file with the path you just build above.

to-dos per assignment with the .zip method

- Zip all of the files in `zipbuild/` - note: don't zip the folder, but the files - i.e. `cd zipbuild && zip Autograder.zip *`
- On gradescope, after creating the programming assignment, upload the `.zip` file in the 'configure autograder' section.
- It should build and be tagged with no errors - if not, check the output of the autograder. Contact me if you run into trouble!

Docker method

If you don't have Docker Desktop, install it: https://www.docker.com/products/docker-desktop/ Then, navigate to your autograding repo, and `cd Dockerbuild`. You will need to add three files here.

.repopath

This file will contain (only) the path you created above.

.dockertag

This will be the tag you'd like to use for your Docker container. Open a file named `.dockertag` and write:

```
tuftscs/gradescope-docker:YOURTAGNAMEHERE
```

Please choose something related to your course for the tag name (e.g. `tuftscs/gradescope-docker:cs-11-2022summer`). Note that `tuftscs/gradescope-docker:` is required.

.dockercreds

We are using a single Dockerhub account for all of the autograding courses. The file `.dockercreds` should be available in the course's Tufts Box folder. If not, reach out to me at `mrussell@cs.tufts.edu` from your Tufts email address; let me know which course you're working on, and I'll send you the file ASAP. Note!! This access token must be kept private; to that end, please keep your course autograding repository private.

Build and upload the container to Dockerhub

Once you've placed the three files in the `dockerbuild` folder, run the commands:

```
cd dockerbuild
./deploy_container
```

The container will be built and uploaded to Dockerhub with the tag you specified. For the future, if you make changes to any of the files in the `dockerbuild` folder, make sure to re-run this script. If you make 'breaking' changes to your autograder, change the tag name in the .dockertag file.

to-dos per assignment with the Docker method

- On gradescope, after creating the programming assignment, select the 'Manual Docker Configuration' option in the 'configure autograder' section; place the contents of the .dockertag file in the box (e.g. `tuftscs/gradescope-docker:cs-11-2022summer`)

Conclusion

Okay, you are ready to begin developing an autograder! Continue to the next section to learn about the autograder, and for a walkthrough to setup an assignment.

Autograding Framework

Introduction

The autograding framework is designed to have you writing and deploying tests as quickly as possible. It supports a variety of options related to test types, etc, however, in general tests will be a set of `.cpp` files. Each one will be compiled and run with the student's submission code, and the output of the test will be `diff`'d against a reference implementation that you provide. `Valgrind` can be run on tests, `stderr` can be `diff`'d. The framework depends on a `testset.toml` file for the configuration.

testset.toml configuration file

`testset.toml` will be configured as follows:

```
[common]
# common test options will go here
# this section can be empty, but is mandatory
# this section must be named 'common'

[set_of_tests]
# subsequent sections will each contain a group of tests to run
# configuration options placed here will override [common]
# test group names (e.g. [set_of_tests]) can be anything
# tests in a section must be placed in a list named 'tests'
# tests = [
      {testname="test0", description="my first test"},
      {testname="test1", description="my second test"},
      ...,
      {testname="testn", description="my nth test"},
]
# each test must have testname and description fields
# you may add any other option to a given test
# test-specific options override any 'parent' options
```

See the section `test .toml configuration options` for details.

Setup Files and Directories

These are all of the possible options, but you may not need many of them depending on your test configuration. [TODO] - ensure that the autograder is 'flexible' - not sure if missing some directories/files will cause an unexpected crash.

```
.
|---canonicalizers.py [opt. file with canonicalization fn(s)]
|---testrunner.sh     [script that runs this file]
|---submission/       [student submission (provided by gs)]
|---testset/          [everything needed to run tests]
|   |---copy/         [files here will be copied to build/]
|   |---cpp/          [.cpp driver files]
|   |---link          [files here will be symlinked in build/]
|   |---makefile/     [contains custom Makefile]
|   |---ref_output/   [output of reference implementation]
|   |---solution/     [solution code]
|   |---stdin/        [files here are sent as stdin]
|---testst.toml       [testing configuration file]
|-
```

Files/Directories Created by the Autograder

```
.
|--- results
|   |--- build      [student submission files]
|   |   |---
|   |   |--- test01 [compiled executables]
|   |   |--- ...
|   |   |--- test21
|   |--- logs
|   |   |--- status
|   |   |--- test01.compile.log
|   |   |--- test01.summary
|   |   |--- ...
|   |   |--- test21.summary
|   |--- output
|       |--- test01.ofile
|       |--- test01.ofile.diff
|       |--- test01.ofile.ccized
|       |--- test01.ofile.ccized.diff
|       |--- test01.stderr
```

```
|       |--- test01.stderr.diff
|       |--- test01.stderr.ccized
|       |--- test01.stderr.ccized.diff
|       |--- test01.stdout
|       |--- test01.stdout.diff
|       |--- test01.stdout.ccized
|       |--- test01.stdout.ccized.diff
|       |--- test01.valgrind
|       |--- ...
|       |--- test21.valgrind
|-
```

Important Notes

- Files in `stdin/` named `<testname>.stdin` (`test01.stdin`) will be sent via `stdin` for that test.
- Files in `.cpp/` named `<testname>.cpp` (`test01.cpp`) will each contain `main()`, and will be compiled and linked with the student's code.
- If you plan to use files in `.cpp`, you must use a custom `Makefile` - see the example:
  `assignments/hw1_ArrayLists/testset/makefile/Makefile`.
- If the students are writing programs which have their own `main()`, then you do not need files in `.cpp` - you may still choose to have your own custom `Makefile` if you wish (otherwise, be sure to set `our_makefile = false` in `testset.toml`).
- The target to build (e.g. `make target`) must be named the same as the program to run (e.g. `./target`).
- Canonicalization functions which are used by the autograder in canonicalizers.py must:
  1. take a single parameter, which is the filename of the student's output
  2. return a string, which contains the canonicalized output
  3. TODO - refactor this to be string input???
- The `.diff`, `.ccized`, and `.valgrind` output files for each test will only be created if your configuation requires them.

- This framework supports `diff`ing against any number of output files written to by the program. Such files must be named `<testname>.ANYTHING_HERE.ofile`. The expectation is that the program will receive the name of the file to produce as an input argument. Then, in the `testset.toml` file, you will add `argv` variable includes #{testname}.ANYTHING_HERE.ofile in the `argv` list. See the `gerp` example. See the `assignments/gerp/testset.toml` file for an example.
- The `summary` files are a 'snapshot' of all of the variables of a test - a summary is created upon initialization of the test, and is overwritten after a test completes with all the information about the test. This is very useful for debugging!

ofiles

How to Build Reference Output

Once you've configured your tests, you can build the reference output as follows:

```
cd ../../framework
python3 build_ref_output.py -p ../../assignments/hwname
```

The reference code will be run as a submission, and the output of the reference will be placed in the `REPO_ROOT/hwname/testset/ref_output/` directory.

Testing an Autograder Locally

After you've produced the reference output, copy a potential submission code to a directory named `submission` in the autograder folder (`REPO_ROOT/hwname/submission/`). Then run

```
python3 ../../framework/autograde.py
```

Parallel Compilation and Parallel Execution

If you would like to enable parallel compilation and parallel execution of tests, instead run

```
python3 ../../framework/autograde.py -j NUMCORES
```

where `NUMCORES` is the number of cores you would like to utilize (`-1` will use all available cores). Note that multiple tests may be run on each core concurrently. The default setting is for one core to be used with no tests running concurrently; that is, only one test will be run at a time (no concurrent tests are run). You can also build the reference output with parallelization by running

```
python3 build_ref_output.py -p REPO_ROOT/hwname -a ../../
    framework -j NUMCORES
```

Note that on gradescope the file `testrunner.sh` is what actually runs the autograder. You can change the command in that file to include `-j NUMCORES` if you'd like, although on gradescope there isn't likely much to be gained from this.

Test .toml Configuration Options

These are the configuration options for a test. You may set any of these in `[common]`, under a test group, or within a specific test.

option	**default**	**purpose**
max_time	30	maximum time (in seconds) for a test
max_ram	-1 (unlimited)	maximum ram (in kb) for a test
valgrind	true	run an additional test with valgrind
diff_stdout	true	test diff of student vs. reference stdout
diff_stderr	true	test diff of student vs. reference stderr
diff_ofiles	true	test diff of student vs. reference output files
ccize_stdout	false	diff canonicalized stdout instead of stdout
ccize_stderr	false	diff canonicalized stderr instead of stderr
ccize_ofiles	false	diff canonicalized ofiles instead of ofiles
ccizer_name	""	name of canonicalization function to use
our_makefile	true	use testset/makefile/Makefile to build tests
pretty_diff	true	use diff-so-pretty for easy-to-ready diffs
max_score	1	maximum points (on gradescope) for this test
visibility	"after-due-date"	gradescope visibility setting
argv	[ ]	argv input to the program
executable	None	executable to build and run