Git-keeper: An Automated Assignment Testing Environment Based on Git

CCSC 2018 Midwest Conference

Ben Coleman and Nathan Sommer

This repository contains instructions and example assignments for the Git-keeper tutorial session at the CCSC 2018 Midwest Conference.

Prerequisites

To fully participate in this tutorial, you will need a computer with the following:

- Python 3
- Git
- A UNIX-like command-line environment

Part 1: The Student Role

Part 2: The Faculty Role

In this portion of the tutorial you will experience Git-keeper from the faculty perspective by creating and publishing some assignments. Everyone will form groups of 3-4 and each create classes where the other members of your group are your students. You will then each publish assignments for each other to complete and submit.

Installing the Git-keeper Client

Faculty members interact with Git-keeper via a command line client. The client can be installed using `pip` like this (FIXME: Switch from pypi test. Do we want them to use a virtual environment?):

```
python3 -m pip install --index-url https://test.pypi.org/simple/ --extra-index-url https://pypi.org/sim
```

Configuring the Client

To configure the client, run `gkeep config`. This will prompt you for several configuration options. Enter `ccsc.cs.moravian.edu` for the server hostname and the username portion of your email address for the username. The SSH port and submissions fetch path may be left blank. A configuration file will be created at `~/.config/git-keeper/client.cfg`.

`gkeep` interacts with the server over SSH, and so it is necessary to copy your SSH public key to the server. If you have never created SSH keys on your computer before, run `ssh-keygen` to create a public/private key pair. All of the default options should be fine. For the purposes of this tutorial it would be best to create a key *without* a passphrase. It is possible to use a key with a passphrase with Git-keeper, but it requires that you properly set up `ssh-agent`.

Once you have a key pair, copy the public key to the tutorial server:

```
ssh-copy-id <username>@ccsc.cs.moravian.edu
```

Now test that your configuration works by running `gkeep query classes`. If everything is configured correctly there should be no output from this command since you have not yet created any classes.

Creating Your Class

Now create a class on the Git-keeper server. To create a class you first need to create a CSV file with the names and email addresses of the students in the class.

To create a class you must create a CSV file containing information about each student in the class. Each row of this file must be in the form `Last,First,email@address.edu`, like so:

```
Hamilton,Margaret,mhamilton@example.edu
Hopper,Grace,ghopper@example.edu
Lovelace,Ada,alovelace@example.edu
```

Edit a new CSV file and add the members of your group to the file. For the purposes of this tutorial you can give your class the same name as your username on the server, so you should name this CSV file `<username>.csv`, where `<username>` is your username on the server. Do not add yourself, as you cannot be a member of your own class.

Now you can add the class on the server using `gkeep`:

```
gkeep add <username> <username>.csv
```

Since everyone in your class is participating in this tutorial, everyone already has an account on the server. If you were to add a student that does not already have an acount to a class, an account would be created automatically for that student and they would receive an email with their initial password.

The Structure of an Assignment

A Git-keeper assignment must follow a certain directory structure. The name of the assignment directory will be the name of the assignment on the server. This directory must contain 3 items.

`base_code`

Within the assignment directory there must be a directory named `base_code`. The contents of this directory will be the intial contents of the student's repository for the assignment. Put skeleton code, data files, instructions, etc. in this directory.

`email.txt`

There must also be a file named `email.txt`. The email that students receive when a new assignment is published will always contain a clone URL, and the contents of `email.txt` will be appended to the email after the URL. `email.txt` can be empty but it must exist.

`tests`

Lastly there must be a directory called `tests` which must contain a shell script named `action.sh`. (FIXME mention `action.py`?) The `tests` directory also contains any other code and data files that you will use to test student code.

When a student submits an assignment to the server it creates a temporary clone of the student's repository and and temporary copy of the tests directory. The server then enters the temporary `tests` directory and runs `action.sh` using `bash`, passing it the path to the temporary clone of the student's directory.

This means that before you upload the assignment you can test your tests locally by entering the `tests` directory in the terminal and running `action.sh` with the path to a solution. It is convenient to also store a `solution` directory in the assignment directory for testing. This way you can run the following from inside the `tests` directory to test your tests against your solution:

`bash action.sh ../solution`

The output of `action.sh` (both standard output and standard error) is placed in the email that the student receives as feedback, and is stored in a Git repository of test results that you will be able to fetch from the server.

Creating `action.sh`

For some assignments you can write all your tests in bash and so `action.sh` is the only file you need in the `tests` directory.

For other assignments you may wish to write your tests in another language or to use a testing framework such as JUnit to test the student code. In that case you can simply call your tests from `action.sh`.

Example

Here is an example where the tests are written entirely in `action.sh`.

Let's say you want to create an assignment for which students will write a Python program to print `"Hello, world!"`. You want them to write this program in an initially empty file named `hello_world.py` and for them to receive instructions by email.

You might create an assignment structure like this:

```
hw01-hello_world
   base_code
      hello_world.py
   email.txt
   solution
      hello_world.py
   tests
       action.sh
       expected_output.txt
```

where `base_code/hello_world.py` is an empty file, `email.txt` contains instructions for the assignment, `solution/hello_world.py` contains your solution, and `tests/expected_output.txt` contains `"Hello, world!"`

What's left is to write `action.sh`. You want to run the student's `hello_world.py` and compare the output to `expected_output.txt`. Here is a simple way to go about it:

```
# The path to the student's submission directory is in $1
SUBMISSION_DIR=$1

# Run the student's code and put the output (stdout and stderr) in output.txt
python $SUBMISSION_DIR/hello_world.py &> output.txt

# Compare the output with the expected output. Throw away the diff output
# because we only care about diff's exit code. If we did not throw the output
# away, it would become part of the results that are emailed to the student
diff output.txt expected_output.txt &> /dev/null

# diff returns a non-zero exit code if the files were different
if [ $? -ne 0 ]
then
    echo "Your program did not produce the expected output. Your output:"
    echo
```

```
    cat output.txt
    echo
    echo "Expected output:"
    echo
    cat expected_output.txt
else
    echo "Tests passed, good job!"
fi

# Always exit 0. If action.sh exits with a non-zero exit code the server sees
# this as an error in your tests.
exit 0
```

Time and Memory Limits

Student code containing infinite loops is troublesome. Let's add the use of `timeout` and `ulimit` to the example to catch infinite loops or excessive memory use:

```
# Use a 10-second timeout
TIMEOUT=10
# Limit memory use to 400 MB
MEM_LIMIT_MB=400

# ulimit uses KB
MEM_LIMIT_KB=$(($MEM_LIMIT_MB * 1024))

# Set the memory limit
ulimit -v $MEM_LIMIT_KB

# The path to the student's submission directory is in $1
SUBMISSION_DIR=$1

# Run the student's code and put the output (stdout and stderr) in output.txt.
# If execution takes more than 10 seconds the process will be killed
timeout $TIMEOUT python $SUBMISSION_DIR/hello_world.py &> output.txt

# timeout returns an exit code of 124 on a timeout, otherwise it returns the
# exit code of the process it ran
if [ $? -eq 124 ]
then
    echo "Your program ran for too long. Perhaps you have an infinite loop?"
    exit 0
fi

# Compare the output with the expected output. Throw away the diff output
# because we only care about diff's exit code
diff output.txt expected_output.txt &> /dev/null

# diff returns a non-zero exit code if the files were different
if [ $? -ne 0 ]
then
    echo "Your program did not produce the expected output. Your output:"
    echo
    cat output.txt
```

```
    echo
    echo "Expected output:"
    echo
    cat expected_output.txt
else
    echo "Tests passed, good job!"
fi

# Always exit 0. If action.sh exits with a non-zero exit code the server sees
# this as an error in your tests.
exit 0
```

Uploading an Assignment

Let's assume you have created an assignment named `hw01-hello_world`, as in the example above, which is for the class `CS100`. You can upload the assignment to the server like so:

```
gkeep upload CS100 hw01-hello_world
```

Uploading an assignment does not immediately send it to your students. If the assignment uploads successfully you will receive an email that looks just like the email that the students get when they receive the assignment. You can make sure the assignment works as expected by cloning it and pushing some solutions.

If you discover errors in your assignment you can update it before sending it to the students. You can modify any one of the three components of the assignment individually, or update everything. To update the entire assignment:

```
gkeep update CS100 hw01-hello_world all
```

Run `gkeep update` without additional arguments for more info.

Publishing an Assignment

Once you are satisfied with your uploaded assignment you can publish it, which will send it out to students:

```
gkeep publish CS100 hw01-hello_world
```

Once an assignment is published you cannot update the base code or the email, but you can update the tests.

Fetching Submissions

When you want to grade your students' submissions, you can fetch them with `gkeep fetch`. If you defined `submissions_path` in the `[local]` section of `client.cfg` then `gkeep` can fetch all your assignments into a common directory. Otherwrise you need to specify a directory to fetch to.

This will fetch into `submissions_path`:

```
gkeep fetch CS100 hw01-hello_world
```

Fetch into the current directory:

```
gkeep fetch CS100 hw01-hello_world .
```

The fetched directory for the assigment contains 2 subdirectories: `reports` and `submissions`. `reports` contains the text of the emails that your students received when they submitted. The `submissions` directory contains the code that your students submitted.