ICT3101 Lab 3

Continuous integration and automated testing

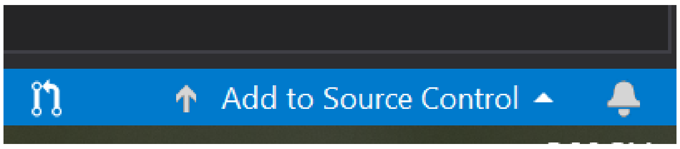
As you well know, being a software engineer commonly involves collaborating as part of a team. In order to harmonize the software development process, we rely on shared repositories and continuous integration management systems to facilitate our collaborative development efforts. This is no different when it comes to developing tests. In fact, we can go one step further by automating our software testing in the continuous integration process: enter Regression testing.

Whenever a teammate wishes to commit and push their code to a shared repository, it's important to ensure that this commit doesn't break the current version. That is, we need the system to still be able to build, but also ensure it passes all the requisite tests. By automating our testing to be run on each new build of the system we help ensure no new bugs have been created. This is the process of Regression testing.

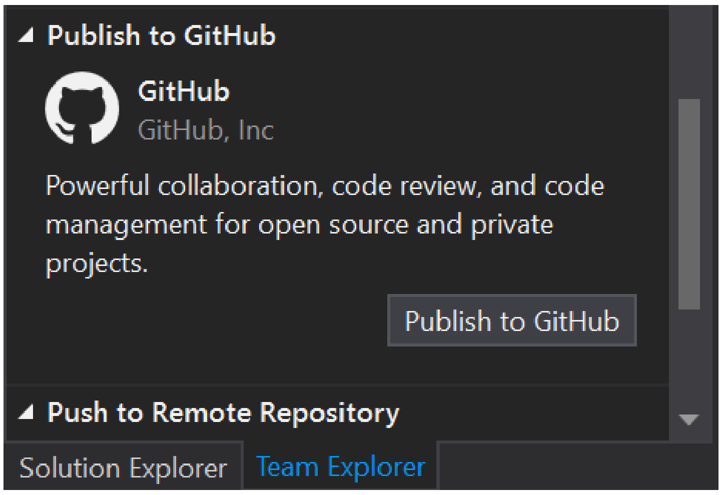
Steps:

1. Ensure that you've installed the Git Extension (Lab 2.1), we'll need this to manage our project. Also ensure you have a Github account (would be strange if you didn't have one already, but… there's no time like the present).

2. Right, so now open your Solution Explorer and click on the solution. At the bottom right of the screen you should see the following:



3. Click this and then select "Git".
4. And then go to the "Team Explorer" tab and click on "Publish to GitHub, as below.


Publish to GitHub
GitHub
GitHub, Inc
Powerful collaboration, code review, and code management for open source and private projects.
Publish to GitHub
Push to Remote Repository
Solution Explorer
Team Explorer

5. Open up your Github account in a browser and ensure that the repo is set up. Fingers-crossed all is good.

Test Automation (this section is adapted from *Software Testing Primer v2, Nick Jenkins 2017*)

The field of automated testing is born from the goals to reduce cost and to expedite the process of testing. Generally, this is a boon to the software development testing process.

However, there are some myths about automated test tools that need to be dispelled:

- *Automated testing does not find more bugs than manual testing:* experienced manual testers, familiar with the system, will find more new defects than a suite of automated tests.
- *Automation does not fix the development process*: as harsh as it sounds, testers don't create defects, and developers do. Automated testing does not improve the development process although it might highlight some of the issues.
- *Automated testing is not necessarily faster:* the upfront effort of automating a test is much higher than conducting a manual test, so it will take longer and cost more to test the first time around. Automation only pays off over time. It will also cost more to maintain.
- *Everything does not need to be automated:* some things don't lend themselves to automation, some systems change too fast for automation, some tests benefit from partial automation.

The Hidden Cost

The hidden costs of test automation are in its maintenance. An automated test asset which can be written once and run many times pays for itself much quicker than one which must be continually rewritten to keep pace with the software. And there's the rub. Automation tools, like any other piece of software, talk to the software-under-test through an interface. If the interface is changing all the time then, no matter what the vendors say, your tests will have to change as well.

What is Automated Testing Good For?

Automated testing is particularly good at:

- Smoke testing (Primary system flow and functionality testing): a quick and dirty test to confirm that the system 'basically' works. A system which fails a smoke test is automatically sent back to the previous stage before work is conducted, saving time and effort.
- Regression testing: testing functionality that *should not have changed* in a current release of code. Existing automated tests can be run, and they will highlight changes.

Pitfalls of Test Automation

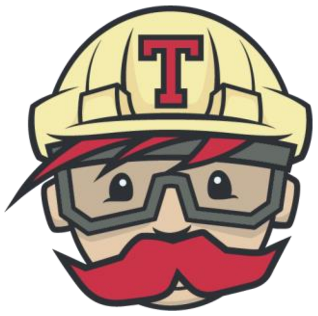
Automating is like a software project in its own right. It must have clearly defined requirements and specify what is to be automated and what isn't. It must design a solution for testing and that solution must be validated against an external reference. Consider the situation where a piece of software is written from an incorrect functional spec. Then a tester takes the same functional spec. and writes an automated test from it. Will the code pass the test? Of course. Every single time. Will the software deliver the result the customer wants? Is the test a valid one? Nope.

Manual testing/testers could fall into the same trap, but Automated tests only do what they're told, and they won't ask questions—unlike human testers. Further, automated tests should be designed with maintainability in mind. They should be built from modular units and designed to be flexible, parameter driven and follow rigorous coding standards. There should be a review process to ensure standards are implemented.

NOTE: this section includes verbatim content from the Software Primer book (Jenkins 2017), with minor edits. Please consult the book for a more expansive discussion.

Travis CI

6. Next you're going to need to set up a Travis CI account. Go to https://travis-ci.com/ and sign up, if you haven't already. Next link it to your Github account (the signup process should guide you through this).



7. Okay, so our continuous integration framework is set up, but now we need to configure it. To do this we'll need to add a Yaml file to our project. Create a file in your solution (will automatically be nested under the "Solutions Items" folder). Call the file ".travis.yml". This is the configuration file.

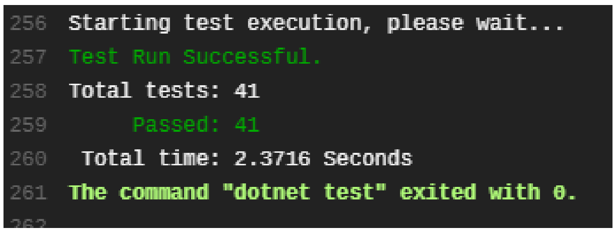
8. One of the great things about .Net Core is that they're streamlining a lot of testing and configuration. Hence, your yaml file can be as simple as follows (be precise with spacing, spelling, and the file name):

```
language: csharp
solution: ICT3101_Calculator.sln
dotnet: 3.1
mono: none

script:
  - dotnet restore #this will restore are settings and get all our NuGet packages
  - dotnet test #this will run all our tests
```

9. Once you've saved this file, you'll need to go to Changes: stage, commit, sync, and then push to your Github repo (make sure you re-familiarize with this Git process).

10. Now make another change to your code. Commit and push again and go to your Travis account to see the build take place. It should boot up a VM for your Solution and run everything. You should get something like the following:



```
Starting test execution, please wait...
Test Run Successful.
Total tests: 41
     Passed: 41
 Total time: 2.3716 Seconds
The command "dotnet test" exited with 0.
```

Good job! You've just set up an automated regression testing process and hence you've become 10-20% more employable!

—The end of questions—