ICT3101 Lab 3

Continuous integration and automated testing

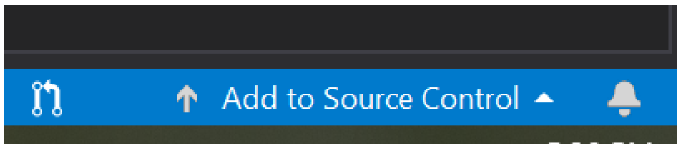
As you well know, being a software engineer commonly involves collaborating as part of a team. In order to harmonize the software development process, we rely on shared repositories and continuous integration management systems to facilitate our collaborative development efforts. This is no different 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.

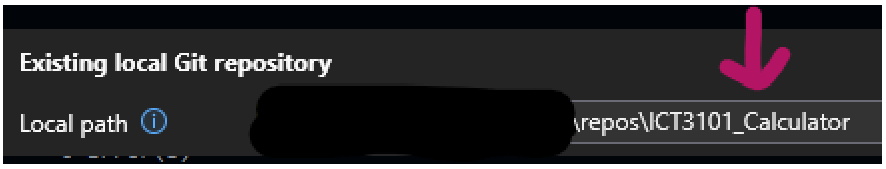
Steps:

1. We'll need to manage our project with Github for the coming labs, so please 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 something like the following, click it (Click this and then select "Git"):



3. Note: when you're adding to Git ensure that you add the folder (where arrow is pointing in image) containing your solution and projects if you want the underlying projects to all be under version control in the same Git repository.


Existing local Git repository

Local path [redacted] \repos\ICT3101_Calculator

4. And then this should be visible in the "Team Explorer" and "Git Changes" tabs. You'll now be able to publish to GitHub and commit + push updates to your code.

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.

Git Actions for CI

6. Next, you're going to set up a Git Actions. Login to your GitHub account, select the repo you have for the Calculator and click on the "Actions" tab. Now set up a "New workflow" for the appropriate .NET application. This should set up a "/.github/workflows" directory and create a "dotnet.yml" file.

7. Okay, so our continuous integration framework is set up, but now we need to configure it. To do this we'll need to edit the given Yaml file for our solution and projects. You should also ensure that you add this Yaml file to your VS solution explorer – so you can edit it locally. Create/get the existing file into your solution (will then become nested under the "Solutions Items" folder). The file should still be called "dotnet-desktop.yml". We'll need to configure this.

8. Thankfully, Git Actions' default file is already mostly good-to-go, but there may be some possible issues depending on our repo structure (**follow the subsequent steps, only if needed**). In the following section of the file ensure that it's configured to run on windows-latest and IF your solution file is not in the main repo folder, then ensure you specify its containing folder's location under working directory.

```
jobs:
  build:

    runs-on: windows-latest

    defaults:
     run:
       working-directory: ./SubFolderThatContainsSolutionFile/
```

9. The dotnot-desktop.yml template will have entries under environment ("`env:`") to edit too. Ensure you add your Solution_Name and both of the Test_Project_Paths for the NUnits and the SpecFlow projects.

10. If you are having issues with the steps, you can manually edit your flow by adding named steps for the Tests as below. Mine looks like the following:

```
    - name: NUnit Tests
      working-directory: ICT3101_Calculator.UnitTests/
      run: dotnet test --no-build --verbosity normal
    - name: SpecFLow Tests
      working-directory: SpecFlowCalculatorTests/
      run: dotnet test --no-build --verbosity normal
```

I've renamed the "test" with NUnit Tests and add the SpecFlow ones as well. These will be run in their respective directories.

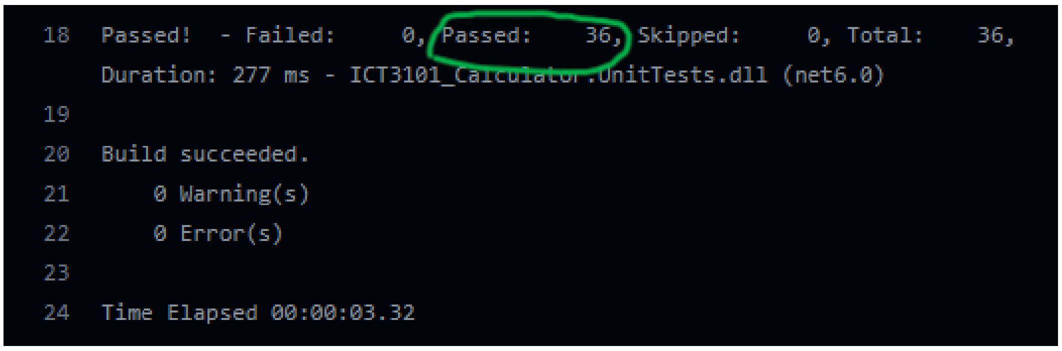
11. Finally, you'll also need to comment out (or delete) everything related to the `Wap_Project_Directory` and `Wap_Project_Path`. These are used in several named steps that we won't be using as we're not making a web application. Depending on your exact file this may vary, but on the template it should be that you comment from (and including):

```
"# Decode the base 64 encoded pfx and save the Signing_Certificate
 #- name: Decode the pfx
 #  run: |
…"
```

and until the end of the file – these flows aren't needed for our build.

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

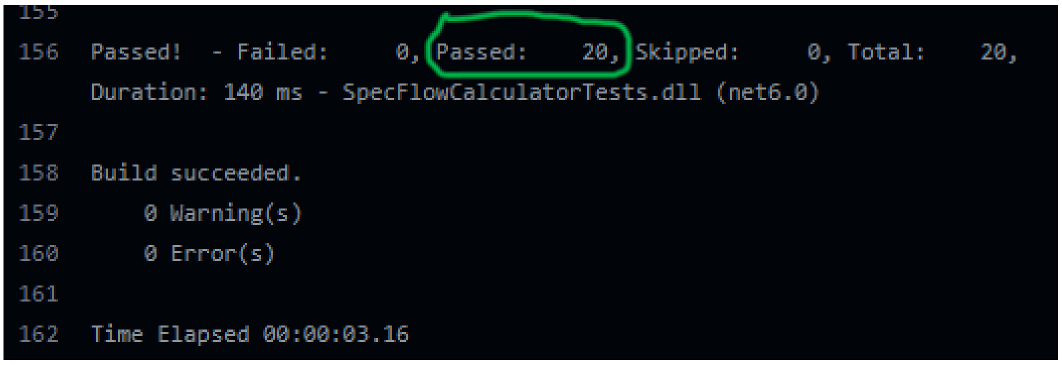
13. Now make another change to your code. Commit and push again and open the Actions tab in your GitHub to see the build take place. It should boot up a VM for your Solution and run everything. This starts the first auto-build and test (commencing the CI flow). You should get something like the following (click on your action workflow run to see your results):



```
Passed!  - Failed:     0, Passed:    36, Skipped:     0, Total:    36,
Duration: 277 ms - ICT3101_Calculator.UnitTests.dll (net6.0)

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:03.32
```



```
Passed!  - Failed:     0, Passed:    20, Skipped:     0, Total:    20,
Duration: 140 ms - SpecFlowCalculatorTests.dll (net6.0)

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:03.16
```

Good job! You've just set up an automated regression testing process.

—The end of steps/questions—