Continuous integration (CI) with github actions Icon

Description automatically generated

Getting Started

This guide briefly explains the idea of continuous integration (CI) and gives a step-by-step guide to implement a simple CI setup using git and github actions.

Important resources:

[Github actions reference docs](https://docs.github.com/en/actions/reference/workflow-syntax-for-github-actions) (These are useful for looking up syntax. Not for reading in total)

[Building and testing java applications with maven](https://docs.github.com/en/actions/guides/building-and-testing-java-with-maven) (Read it. Only about one page)

Continuous integration

The idea of continuous integration is simply to merge developing effort into a central, or integrated, environment. The main reason is to avoid heterogeneous environments. Merging code from many developers can be hard because of local dependencies or settings which ‘works on my machine’ (also known as integration hell). In continuous integration, the code will be pushed to a different clean server and tested, before being deployed to production.  
  
In practices like eXtreme Programming (XP) where tests are put in place before code, automated testing (with for instance JUnit) is exploited to act as a ‘guard’ to the production environment. If the tests are green the code can be released. If they are not green, someone needs to cough up a good excuse for ruining production code.  
  
This becomes especially useful if testing can be integrated into the normal development flow. A common method is to use versioning tools like git. You are using this anyway, so in a sense, you can get CI for free. Git and Github is especially good at this because it features commit hooks which can be run whenever someone is committing. By using these hooks, tests can be set up to run automatically on a build server, when someone pushes to a shared repository.  
  
This guide will show you how you can use CI with git and “github actions”.

Setting up a GitHub repository

For this exercise, you should fork and clone the following repository:

* <https://github.com/HartmannDemoCode/ci_actions>

and create a copy under your own GitHub account.

This is a very simple Java project which contains a function which returns the integer 10, along with a single test that verifies that the method actually returns 10. The project uses Maven to run the test and fetch dependencies. After adding the project to your own GitHub account, try to pull it and run the test.

What is “github actions”?  
[From github](https://docs.github.com/en/actions/guides/about-continuous-integration): “Continuous integration (CI) is a software practice that requires frequently committing code to a shared repository. Committing code more often detects errors sooner and reduces the amount of code a developer needs to debug when finding the source of an error. Frequent code updates also make it easier to merge changes from different members of a software development team. This is great for developers, who can spend more time writing code and less time debugging errors or resolving merge conflicts.

When you commit code to your repository, you can continuously build and test the code to make sure that the commit doesn't introduce errors. Your tests can include code linters (which check style formatting), security checks, code coverage, functional tests, and other custom checks.

Building and testing your code requires a server. You can build and test updates locally before pushing code to a repository, or you can use a CI server that checks for new code commits in a repository.”

Using Actions

The goal of this exercise is to tell our CI server (Github actions server) to monitor and watch code being pushed to our repository. Whenever GA figures that out, it should run tests on the code to check whether a developer committed and pushed some bad (untested) code. CI using GitHub Actions offers workflows that can build the code in your repository and run your tests. Workflows can run on GitHub-hosted virtual machines, or on machines that you host yourself.

To figure out which language and code to run, GA uses a configuration file (.yml file) which must be placed in the project root inside ./github/workflows/. Take a look at the file in our project, it is called .github-actions-demo.yml (notice the . in the beginning).

Goto to github actions tab and enable workflows (first time only).

With the cloned project you should add all files to git and make an initial commit, then push your repository to github. On the github repository page click on: “Actions” in the top menu:

A screenshot of a computer screen

Description automatically generated with medium confidence

And see that 1 workflow was run. Click on the commit message and then on the “explore github actions” button to see the details of the workflows that was run based on the yml file in .github/workflows folder.

Next step: testing with maven

We want the build server to run our unit tests to see if the committed code passes the tests. So this time we will create a new yaml file.

A workflow is a configurable automated process made up of one or more jobs. We must create a YAML file to define our workflow configuration. We can create several yaml files on the same repo as long as we put them in the workflows folder.

Create the yaml file in the workflows folder, and call it maven.yml and then take a look at: <https://docs.github.com/en/actions/guides/building-and-testing-java-with-maven>

Copy the first code snippet from the article:

name: Java CI

on: [push]

jobs:

  build:

    runs-on: ubuntu-latest

    steps:

      - uses: actions/checkout@v2

      - name: Set up JDK 11

        uses: actions/setup-java@v2

        with:

          java-version: '11'

          distribution: 'adopt'

      - name: Build with Maven

        run: mvn --batch-mode --update-snapshots verify

Into your file, save it and git commit + push. Then go to the actions tab on the remote repo to see the tasks running. Click on the job: build and scroll to the bottom to see that the test has run successfully.

Next step: auto deploy with maven

1. Make sure that you can access your tomcat manager app from <your digital ocean ip>:8081/manager in a browser (If not log on to digitalocean.com and open up the port 8081 from the EduFirewall settings on your droplet.
2. change the ip address in the POM.xml property: <remote.server> to your own ip.
3. Add 2 lines to your workflow yml file:

- name: Maven deploy

  run: mvn -Dremote.user=${{ secrets.REMOTE\_USER }} -Dremote.password=${{ secrets.REMOTE\_PW }} tomcat7:deploy

1. In your browser on your github page for the repository click on settings. Then click on the Secrets menu item.
2. Add two new secrets: REMOTE\_USER and REMOTE\_PW which must contain your username and password for your tomcat user on your droplet respectively.
3. Now save your changes locally and commit+push to your repo
4. Check your actions tab to see when the task is finished successfully and then go to your tomcat manager app to see that the app has been deployed.