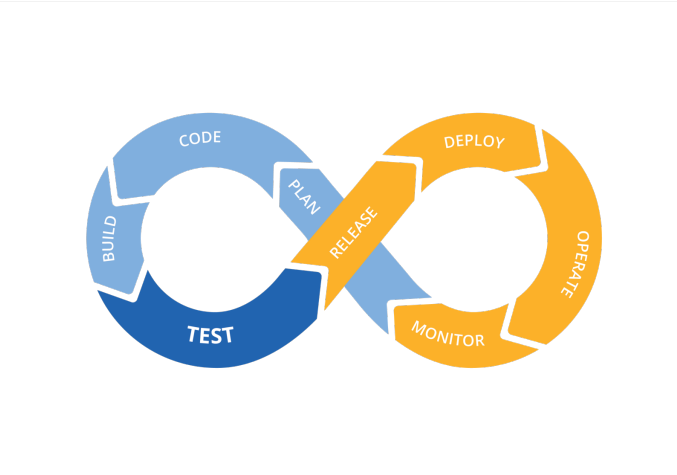


DevOps and Continuous Delivery

CO50007.1 - Second Year Computing Laboratory
Department of Computing
Imperial College London

18th – 22nd January 2021

For this laboratory exercise, you should work in ***groups of 4 people****.*

Aims

- To gain knowledge of and experience with *Continuous Delivery* pipelines;
- To experiment with some popular DevOps tools, like GitLabCI, Heroku and Docker;
- To gain some familiarity with system administration tasks.

A (flash) Introduction to DevOps and Continuous Delivery

In order to make sense of this assignment and its importance, let's try to understand what DevOps and Continuous Delivery actually mean, and how they relate to one another.

DevOps

The term *DevOps* represents the fusion of cultures, practices and tools belonging to Software Development and Infrastructure Operations teams respectively. In the barbaric *pre-agile* era, Software Development was dominated by the Waterfall method[1], and the release of any piece

[1] `https://manifesto.co.uk/agile-vs-waterfall-comparing-project-management-methodologies/`

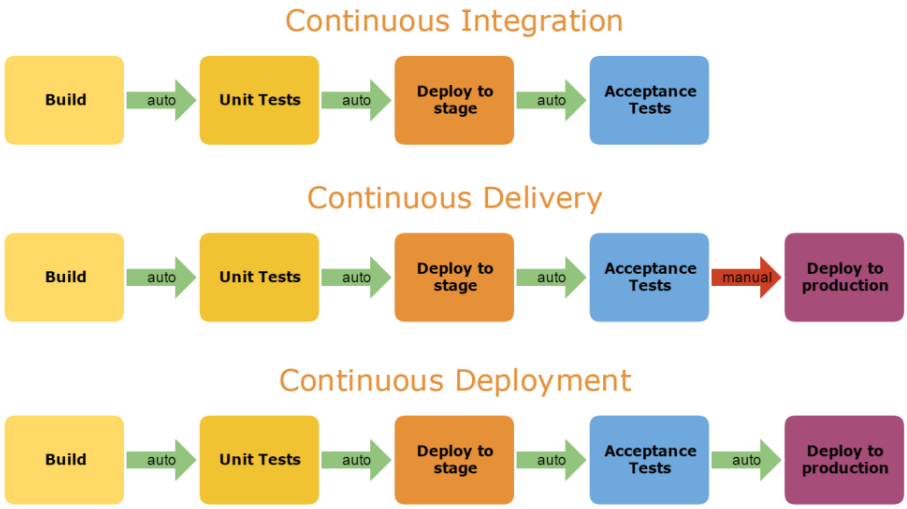


Figure 1: Comparison between Continuous Integration, Delivery and Deployment.

of software usually resulted from the concatenation of *siloed* phases, each being the only/predominant concern of a dedicated team. The different teams (Dev, QA, Tools, Infrastructure, Platform, Operations etc. just to name a few) were in turn orchestrated by the Program/Project Management team(s), which coordinated the succession of responsibilities through to the final delivery of the software to the customer(s).

When the agile approach to development started to make its way in industry[2] at the turn of the new millennium, the need for a **faster**, more reliable, efficient and **systematic** way to deploy software arose. This resulted in an ever closer collaboration between the "makers" of the code (*Dev*) and the "care-takers" for it (*Ops*), eventually leading up to the merging of the two roles into one whose concerns follow the entire service lifecycle, from design, to development, to production support.

The resulting *uber*-developer has an increased awareness of the issues that rolling code into production can bring about, and is therefore in a privileged position to design:

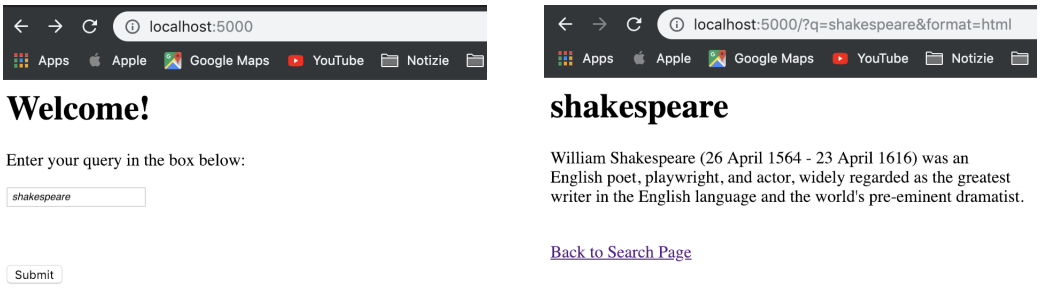
(a) code (or better, code changes) that integrate into or update the production code in place in as seamless a way as possible;

(b) a pipeline of tests of suitable sorts for the new code to go through, being confident enough that whatever code change passes the checks of such a pipeline is ready to be shipped directly to the end-user.

And in a nutshell, this is the objective of *DevOps*, now and back when Patrick Debois coined the term in 2009: *reducing, for any change introduced in a software, the time interval between committing that change (to a Version Control System) and that change being part of the actual software the end-user uses*[3].

[2]For more information about when and why *agile development* came to be: `https://techbeacon.com/app-dev-testing/agility-beyond-history-legacy-agile-development`

[3]Should you be willing to read more about DevOps motivations and history, you might want to have a look at this article:`https://theagileadmin.com/what-is-devops/`



(a) Query for *shakespeare* (b) Result

Figure 2: Sample interaction with the simple web app.

Continuous Delivery

Among the different practices that are associated with the DevOps philosophy, one that has (relatively) recently achieved huge popularity is *Continuous Delivery*. As the name might already suggest, Continuous Delivery naturally "sits" on top of the experience of Continuous Integration (CI)[4], by this meaning that, like CI, it involves (a) frequent commits to a common code repository, (b) automated building and testing of the code, (c) generation of reports on build failures, and crucially adds the deployment of successfully built code to the production environment "at the click of a button". Whenever this last step is also automated, the whole process goes under the revised name of *Continuous Deployment* (see Figure 1). The steps are executed in a sensible succession (usually *build, then run unit tests, run integration tests, deploy to staging environment, run acceptance tests, and finally deploy to production environment*), and, on failure of any of them, the whole pipeline stops. The key objective of Continuous Delivery/Deployment is that of getting code changes (be they features, configuration updates or bug fixes) to the end-users in a quick, predictable and reliable fashion.

Problem Overview

You are provided with a simple, working, Java web application, which accepts a query from a text field and returns a result (if any). Figure 2 shows a sample interaction with a query for "Shakespeare".

You should set up a Continuous Delivery/Deployment pipeline for the application with *GitlabCI*[5], so as to execute the following key stages in an automated fashion:

1. build (i.e. compile, in our case) the code;
2. run all the available unit/integration tests;
3. deploy to your production environment.

GitLabCI, similarly to other Continuous Integration/Delivery tools (modulo different names and naming conventions), has two overall components: a Server, whose responsibility is to

[4]CI can be summarised as the practice of *continuously merging small changes into* `master`. Refer to the slides of 50002, lecture 16: `https://www.doc.ic.ac.uk/~rbc/50002/lecture-notes/16-ContinuousDelivery.pdf`.

[5]`https://about.gitlab.com/product/continuous-integration/`

monitor a VCS repository to detect newly pushed commits (in this case this is the GitLab server itself, which is conveniently also the Version Control server); a *Runner* (one or more instances), that picks up new commits from the Server and executes all the *jobs* declared for the pipeline, in the correct sequence. Jobs (i.e. what a Runner must do whenever it is woken up) are declared in a dedicated configuration file. In the case of GitLabCI, this file is in YAML[6] format and is named `.gitlab-ci.yml`.

Runners can be installed on and execute jobs from any machine, as long as they are registered to track changes on a specific GitLab repository, and the machine they are sitting on has an internet connection. More on this later.

In this lab, you will build a GitLabCI pipeline to deploy the simple web app code first to a DoC cloud virtual machine, then to the *Heroku* Cloud Application Platform. Finally, you will wrap the code in a *docker image*, and deploy that to Heroku instead.

Tasks

- Your first task is to set up a Continuous Deployment pipeline using GitLabCI. In order to do this, you need to:
  1. Create and set up a virtual machine on the DoC cloud, as per instructions further below;
  2. Install `gitlab-runner` on your virtual machine and register a Runner for your repository;
  3. Write, commit and push a `.gitlab-ci.yml` file **to activate your pipeline** for compiling, testing and deploying your Java code on your virtual machine[7].
- You should then set up a free account on Heroku and change the deployment stage in your `.gitlab-ci.yml` to deploy there instead.
- Finally, write a `Dockerfile` to specify how to build an image for packaging and executing your code, and deploy the container on Heroku.

Submit by 19:00 on Friday 17th January 2020

What To Do:

Fork the exercise repository on GitLab from `https://gitlab.doc.ic.ac.uk/261/simplewebapp` (by clicking the *Fork* button on the repository page) to create your own copy, and clone your copy on your machine. ***Note** that only one member of your group should fork the repository.* The other members should be added as collaborators to the forked copy. The repository contains:

- `src` – This directory contains the Java source code and (sample) test suite for a simple web application.

[6]The YAML format is becoming more and more popular in industry. To read more about it, refer to `https://yaml.org/spec/1.2/spec.html`.

[7]You might at this point be wondering why your Build Server and your Production Server happen to be the same machine. You would be right. We are asking you to do so here to keep things simple. You can just as well pretend that you have two separate servers. This will actually be the case further on, when you deploy to Heroku.

- `pom.xml` – *Last term, in Software Engineering Design, you used* ***`gradle`*** *to coordinate building and testing your code. In this lab, you will familiarise yourselves with a similar (yet different) Java build tool:* ***Maven.*** The `pom` file, aka Project Object Model, is Maven's project configuration file, that manages the build and project dependencies. Such dependencies are compatible packages written in Java (or another JVM language).

Pre: Running the App

Before getting started with the GitLabCI pipeline, take some time to familiarise yourselves with the App and, most importantly, with the commands to compile it, test it and run it.

To compile and test the project, from the top directory run `mvn` with the `compile` and `test` commands respectively. To run the application, run `mvn package`. This will create the executable script `target/bin/simplewebapp`. You might have noticed, if you read the code in `WebServer.java`, that the server reads the HTTP connection port from the `PORT` environment variable, which you have to set to a port of your choice. So finally, to start the application, execute the following:

```
> export PORT=5000
> sh target/bin/simplewebapp
```

You can now browse to `localhost:5000` and watch the application come to life!

Setting Up Your Virtual Machine

NOTE: *In order to access cloudstack and any VM on the DoC cloud, you are required to be connected to the College private network. Refer to this page for information on how to set up a VPN to connect off-campus:* ***`https://www.imperial.ac.uk/admin-services/ict/self-service/connect-communicate/remote-access/virtual-private-network-vpn/`****.*

As mentioned previously, the GitLab Continuous Delivery pipeline requires a *runner* to execute build and deployment jobs. During this exercise, your *runner* will live on a virtual machine, in DoC's internal cloud. To set up your VM, go to `https://cloud.doc.ic.ac.uk` and log in with your *group leader's* college credentials. On the left-hand side bar of the landing page, hover the cursor over *Compute* and move, without clicking, first on *Infrastructure* and then on *Virtual Machines*. You should see a service item named `devops_N` where `N` is equal to your WACC group number. Click on the service item, then on the VM item at the bottom of the service page, under the *VMs* section. This will bring you to the summary page for the VM, where you can find -among other pieces of information- the VM's IP address.

There is now some sysadmin work left to finish setting up the environment. You should execute the following from the VM Console (from the *Access* dropdown menu in the topbar, choose *VM Console*).

- **Grant SSH-Access and Root Privileges to the Members of Your Group** - In order to run all the necessary commands described in the following sections, you need to grant yourselves ssh-access and *root* privileges on the VM. To achieve this, select *VM Console* from the dropdown menu of the topbar on your service's page. You should be redirected to a new tab showing a friendly terminal. From the prompt, you should see that you are logged in as `root`. Run the following for each of your usernames:

  ```
  > usermod -a -G ssh <username>  # append <username> to group ssh
  > usermod -a -G sudo <username> # append <username> to group sudo
  ```

Once this is done, you can close the VM Console browser tab.

- **Logging in remotely** - Log in remotely into your virtual machine using `ssh` from your terminal, as so: `ssh <login>@<your-vm-ip>`. Enter your password when prompted to complete the authentication process. If you wish to make future remote logins easier, you can *register on the virtual machine the ssh public key(s) of the user(s) that will access the vm remotely*. Refer to Appendix A for further information.

- **Install and Register the GitLab Runner** - To install `gitlab-runner` at *system-level*, follow the instructions available at `https://docs.gitlab.com/runner/install/linux-repository.html`, without skipping any step and executing the commands with `sudo`. After completing the installation, follow the link at step 4 on the same page to register a `shell` Runner. The gitlab-ci coordinator URL and the gitlab-ci token specific to your project can be obtained from the GitLab repository home page by navigating to `Settings > CI/CD`.

  At this point, you might notice that a new user, `gitlab-runner`, is registered on your machine[8]. This is the UNIX user that will run your CI/CD jobs.

- **Install Maven and Java 11** - Given that you will be dealing with a Java web application, these are needed to compile and execute your code. To perform the installation with the Advance Packaging Tool, execute the following as root[9]:

```
> apt-get install maven
> apt-get install openjdk-11-jre openjdk-11-jdk
```

Building the Application and Deploying to Your VM

Your VM is now well-configured and your runner is ready to start crunching some jobs for you. To *activate* the CD pipeline, there's one last thing to do: writing a `.gitlab-ci.yml` file. The full documentation on syntax and usage of this file is available at `https://docs.gitlab.com/ee/ci/yaml/`. You should define, in the `.gitlab-ci.yml` file, a **three-stage pipeline** (compile-test-deploy) to deploy the application to your VM. Remember that the application reads *from the environment* the port it should listen on, so be sure to specify that in the job associated with the deploy stage.

Before committing and pushing your `.gitlab-ci.yml` file, make sure that *shared runners* are disabled, as they might pick up your jobs instead of *your* runner. To do so, from your GitLab project page go to `Settings > CI/CD`, expand the *Runners* section and, if the *shared runners* are enabled, click on *Disable Shared Runners* (more on *shared runners* later).

Assuming your VM has IP `123.456.78.9` and `5000`[10] is the port you have chosen, on a successful *build* you should be able to access the application from your browser at `123.456.78.9:5000`. You can, at any time, inspect your build logs from your GitLab repo by clicking on the last build-stage node in `CI/CD > Pipelines`.

[8] "How would I notice?", you might rightly ask. Well, there are a few ways to list all the users registered on a machine. An easy one is to explore the content of `/etc/passwd/`, maybe with a little help from the `cut` command!

[9] To execute a command as root, prepend `sudo` to it, e.g. `sudo apt-get update`.

[10] The VMs on the cloud are currently protected by a firewall which prevents any communication between the VM and the outside world. To create an exeption in the firewall for a particular port -say 5000- run 'sudo ufw allow 5000'.

Note that for the pipeline to complete successfully, the *deploy* stage needs to terminate. For this to happen, you need to take care of *starting your web app in a way that allow the gitlab-runner to exit the deployment job.*

Deploying to Heroku

While hosted on your VM, your application is not publically visible, and has limited scaling potential. Let's move the deployment to some more easily accessible place. Heroku is one of the most popular examples of Platform-As-A-Service, i.e. a service offering room for apps to live on the internet. When deploying an application, Heroku executes the command(s) written in a special manifest file, called `Procfile`, on one or more *dynos*, virtualised UNIX containers that provide the environment required by the application. The manifest `Procfile` has recently been superseded by the new `heroku.yml`, which offers increased capabilities for running processes in automation. As part of this lab, you will have the chance to write a simple manifest file for your application as both `Procfile` and `heroku.yml`. By having Heroku take care of the deployment, you do not have to worry about handling the production environment yourselves anymore.

To get started with Heroku, sign up for a free account (`https://signup.heroku.com`). You may select `Java` as your primary development language. Proceed by installing the CLI on your machine (`https://devcenter.heroku.com/articles/heroku-cli`).

NOTE: *The lab machines do not allow you to obtain root privileges, which are necessary for the installation of new packages. Feel free to install and run* ***`heroku`*** *(and, later on,* ***`docker`****) on your own laptop or, alternatively, your DoC virtual machine.*

With the CLI in place, run the following commands from within your project's root folder:

```
> heroku login                          # log in
> heroku create <login>-simplewebapp # create a Heroku App
```

The `.gitlab-ci.yml` file must now be updated to direct the deployment to Heroku. This can be done easily with `dpl`, a deployment tool developed in Ruby by the Travis CI people. To install `dpl` on your VM, run the following:

```
> sudo apt install ruby                 # install ruby
> sudo su - gitlab-runner               # impersonate the gitlab-runner user
> gem install --user-install dpl-heroku # user-level installation of the gem
```

Then, while impersonating the `gitlab-runner` user, paste the following at the bottom of the user's `.profile` file[11]:

```
# set PATH so it includes user's gem dir if it exists
if which ruby >/dev/null && which gem >/dev/null; then
  PATH="$(ruby -r rubygems -e 'puts Gem.user_dir')/bin:$PATH"
fi
```

This will make the local `dpl` gem available as a binary executable to the `gitlab-runner` user. Now update the `.gitlab-ci.yml` file by adding a new deployment stage, as follows:

[11] This is the configuration file executed whenever the `gitlab-runner` picks up a job.

```
# in your .gitlab-ci.yml
...
  script:
    - dpl --provider=heroku --app=<login>-simplewebapp --api-key=<heroku_api_key>
...
```

The `heroku_api_key` is used by Heroku to authorise the deployment of the Heroku Application identified by the name you provided with the `app` flag. The key can be obtained by executing `heroku auth:token`. You can then either paste it directly as the value for the `api-key` flag or add it as a *secret variable*[12] for the gitlab runner (recommended). You should now let Heroku know how to start an application via the `Procfile`[13]. Your `Procfile` should contain a single line defining how to run your `web` process, as so:

```
web: sh target/bin/simplewebapp
```

Pushing these changes to your repo should then trigger the build pipeline that, eventually, will start a deployment on Heroku. By default, Heroku will read your source code, detect a Java project, recognise the `pom.xml` file as your build and dependency manager, and install all the dependencies declared there. You can follow what happens on Heroku by inspecting the generated logs visible on both the Heroku web UI and your GitLab repo's pipelines dashboard[14]. On a successful deployment, you should be able to access the application at `https://<login>_simplewebapp.herokuapp.com/` (or by clicking `Open App` on your Heroku dashboard).

You should now have a four-stage pipeline, where your code gets compiled, tested, deployed to a VM (which you can consider at this point your test or *staging* environment) *and* to Heroku (your production environment).

Extending the Java Web App

Your application is deployed on Heroku, and every new change can be deployed via your build pipeline. It is time to add some new features to your Java web application and assess the impact that some changes and dependencies might potentially have on your deployment environment(s).

- Extend the query processor to add results for new queries, and update the corresponding test suite accordingly. Push and see the new tests pass in the pipeline.
- Extend the application by allowing users to *download* the result of a query in a markdown file[15]. To achieve this, you should *create a (temporary) file and write the result of the query to it (or an appropriate message for the user in case the query did not produce any result; follow the example of `HTMLResultPage`).* You should then serve the file data to the user by *turning the file into an input stream and transfering its bytes to the output stream of the HTTP response.* Note that the HTTP response content type should be set to the

[12] *Secret variables* can be set on GitLab from the **Variables** section in `Settings > CI/CD`, and are automatically injected into the runner(s)' environment.

[13] The full documentation on Procfiles can be found here: `https://devcenter.heroku.com/articles/procfile`.

[14] Chances are Heroku will complain about an '*invalid Java version: 9*' on your first deployment attempt. You might want to have a look at Heroku's Java Support documentation to solve the issue: `https://devcenter.heroku.com/articles/java-support`.

[15] Markdown is the format most notably used to write README files and Wiki pages. It is versatile, rendered automatically by most editors and extremely readable even as plain text. To quickly explore its syntax, have a read here: `https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet`.

appropriate MIME type[16], again in the same fashion of `HTMLResultPage`. Users should be able to choose the format (HTML or markdown) for the result of their query from the index page. A simple radio button should do the job. Finally, deploy the app to Heroku.

- Add support on the server side for returning the result of a query as a PDF file. You should *follow the steps above to create a markdown file, then convert that to PDF using `pandoc`*[17]. `pandoc` should be invoked as a sub-process: investigate how to do this using a Java `ProcessBuilder`. As before, you then should turn the PDF file into an input stream and transfer its bytes to the HTTP response's output stream. For this subtask, you can either update the code written for the previous one or add a third distinct option for the user to choose in the index page. Once again, finish off by deploying your code to Heroku. This deployment should succeed, but (*spoiler alert*) you might find that attempting to use your new PDF generation feature leads to an ungraceful `500 Server Error`.

Adding a dependency like `pandoc` carries one potential complication: in your specific case, `pandoc` is written in Haskell, not Java, and for this reason cannot be included in your `pom.xml`. In similar circumstances, you might rely on the library/tool/binary already being installed on your deployment environments. But what should you do when, like in this instance with Heroku, you have no easy way to perform system administration on the designated deployment environment to install the packages you need? *Containerisation* is a possible answer to this question.

Deploying a Docker Image to Heroku

Docker is a tool that offers special virtualisation functionalities, by enabling users to ship and run their applications, with their application-level- and system-level-dependencies, as single lightweight, portable packages (referred to as *containers*). To better understand Docker and its advantages over virtual machines, refer to the official documentation: `https://docs.docker.com/get-started/`[18]. This part of the exercise requires you to have `docker` installed on your machine (as before, your laptop or your VM). Instructions on how to install the latest version of *Docker CE* can be found here: `https://docs.docker.com/install/`.

Following the changes you have made to your Java source code, your last task is to write a `Dockerfile` to build an image featured with all the necessary system requirements for running the updated application. The `Dockerfile` should be located in the root directory of your project. To learn about the `Dockerfile` syntax, refer to the docs: `https://docs.docker.com/engine/reference/builder/`[19]. The `Dockerfile` should:

1. install maven and pandoc on the container;
2. copy the files that are needed to build the application into the container's file system;
3. run `mvn package` from the correct container's directory;
4. start the application on container execution.

[16] This page offers a detailed explanation of what MIME types are and which ones you can use for your purposes: `https://www.freeformatter.com/mime-types-list.html`.

[17] See the Pandoc documentation: `https://pandoc.org/`.

[18] We also recommend this excellent tutorial for a more hands-on approach: `https://github.com/praqma-training/docker-katas`.

[19] You should understand syntax and semantics at least for *FROM, RUN, CMD* and *COPY*.

Note that your docker image requires Java (specifically a Java version as or more recent than the one specified in your `pom.xml`). Instead of *installing* Java on the container (like you did on your virtual machine), you might want to check Docker Hub for an off-the-shelf *base image* that already includes a suitable JDK.

Experiments with `docker`, including but not limited to *building* your image and *running* it locally, are strongly encouraged. To run your image locally, you will need to (a) set the `PORT` environment variable (to, say, 8080), and (b) perform *port forwarding* to map a port on the host machine (say 5000) to the container's port specified in (a). On the command line, these translate to the following, executed from the root directory of your project:

```
> docker build -t my_image .
> docker run -e PORT=8080 -p 5000:8080 my_image
```

With a correct `Dockerfile`, you should be able to access the web application by browsing to `localhost:5000`.

At this point, if you pushed the `Dockerfile` to your remote repository and the build & deployment ran, Heroku would not know anything about your intentions with it: in fact, it would ignore it altogether. To deploy an application on Heroku *from a container*, you need to:

- set the Heroku *stack* (i.e. the environment it has to use to run the application) to `container`, via the Heroku CLI;
- *write a `heroku.yml` file* to tell Heroku to build and run the docker image specified in your `Dockerfile`, as so:

  ```
  # heroku.yml
  build:
    docker:
      web: Dockerfile # path to your Dockerfile
  ```

  More information on the (brand new) `heroku.yml` manifest file can be found in the docs: `https://devcenter.heroku.com/articles/build-docker-images-heroku-yml`.

The `heroku.yml` file automatically overrides the `Procfile`, so you can confidently retain both to show how you have progressed through these tasks. Heroku should now detect your `Dockerfile`, building and deploying your container to the URL you used for your previous task. You should finally be able to access the web application, submit a query and view (or even download) your search results as a PDF file.

Possible extensions and additional experiments

GitLab CI/CD offers a wide array of features that we did not explore during this Lab exercise. An interesting and generally handy one is, for example, that of *Shared Runners*[20]. These runners are shared across different projects, meaning they are available to run CI/CD jobs issued by anyone who "enables" them for their project(s). The status of shared runners, in relation to your

[20] See the official docs: `https://docs.gitlab.com/ee/ci/runners/`.

GitLab project, can be checked from your GitLab repository, by going to `Settings > CI/CD` and expanding the *Runners* section. The shared runners you see are all *docker executors*[21]. In order to use them, you will have to (a) enable them and (b) update your `.gitlab-ci.yml` specifying a suitable `image` to use for your jobs.

You are also totally free (in fact, encouraged) to extend the provided Java web application at your leisure, and to experiment with other CI/CD technologies like *Circle CI*, *Jenkins* or *Travis*. However, be aware that we will not provide support or aid with those.

Submission

For this exercise, your group should submit a PDF report (`devops-report.pdf`) (2 pages maximum) to CATE by 19:00 on Friday 22nd January 2021. The report must contain:

(a) your names and usernames (e.g. *Alan Turing (amt39)*);

(b) your WACC group number;

(c) the URL to your GitLab repository;

(d) the URL to your Heroku application;

(e) a screenshot of your latest GitLab pipeline dashboard;

(f) a screenshot of the GitLab log of your latest deploy stage.

Assessment

The assessment for this exercise will have the form of a live *review*, where you will be asked to (a) show the full working pipeline [25%], (b) force the pipeline to fail with a bad commit (e.g. by making a test fail) [10%], (c) show the `Dockerfile` and the successful deployment on Heroku [20%], and (d) answer a few questions about the tools you have used for this exercise [30%]. You will also have the chance to discuss any extensions you have worked on [15%].

A Installing an SSH Key on a Server

To install an SSH key on a remote server as an authorised key, have a look at the `ssh-copy-id` command. Should your user not have an ssh key ready to use from the machine you are attempting remote access from, you can always generate one with `ssh-keygen`, like so:

```
> ssh-keygen -o -t rsa -b 4096 -C "email@example.com"
```

The command above generates an OpenSSH-format key (`-o`) of type rsa (`-t rsa`), 4096 bytes long (`-b`), appending as a comment (`-C`) the example email address, that you would replace with your imperial one.

[21] For more information about docker executors, see the documentation: `https://docs.gitlab.com/runner/executors/docker.html`.

B Troubleshooting

- **I can't impersonate the gitlab-runner user** - This is most likely happening because you didn't follow step 1 of `gitlab-runner` installation instructions (`https://docs.gitlab.com/runner/install/linux-repository.html`). You are therefore working with an outdated version of `gitlab-runner`, which does not set up a proper `gitlab-runner` user by default. To fix this, execute the following:

```
> sudo apt autoremove gitlab-runner -y # remove the package
> sudo userdel -r gitlab-runner        # remove the dummy user
> # ==> EXECUTE STEP 1 OF THE INSTALLATION INSTRUCTIONS <==
> sudo apt-get install gitlab-runner   # install latest version
> sudo gitlab-runner unregister --all-runners # prune existing runners
```

  You should now proceed to register a fresh runner for your project, and resume where you left off.

- **My runner is up and, well, running, but the pipeline gets stuck** - This usually happens when you haven't configured your runner to pick up untagged jobs, as likely your `.gitlab-ci.yml` does not (*and should not*) define any tagged jobs. To fix this, go to `Settings > CI/CD`, expand the *Runners* section, scroll down to the "*Runners activated for this project*" section, click the edit button next to your runner and tick the "*Run untagged jobs*" box. Note that the latest version of `gitlab-runner` configures your runners to run untagged jobs by default, so if this is not the case, chances are you need to read the troubleshooting point above too.

C Glossary

C.1 Docker

The following definitions are taken from the official Docker documentation.

- **Image** - An executable package containing everything that is needed to run an application, from code and configuration files to runtime and environment variables.
- **Container** - A running instance of an image.
- **Docker Hub** - A registry of all available Docker images.
- **Dockerfile** - A configuration file defining the environment for an image.

C.2 Heroku

- **Stack** - The environment Heroku uses to run an application (similar to Docker's images).
- **Dyno** - Heroku containers used to run and scale applications.
- **Slug** - Compressed, pre-packaged copy of an application optimized for dyno management.
- **Procfile** - The manifest file specifying the commands (and a process type for each) a Heroku app should run on start up.
- **heroku.yml** - The new Heroku manifest file, that will progressively replace the Procfile.