RANCHER LABS

March 2017

Continuous Integration and Deployment with Docker and Rancher

2nd Edition

Usman Ismail and Bilal Sheikh

Introduction

As Docker projects and their surrounding ecosystem matures, containers are being used for larger scale projects. As a result, coherent processes and workflows are needed to streamline deployment for such projects. In this guide, we will cover a workflow for code development, continuous integration and deployment, as well as zero-downtime updates. Such workflows are fairly standard in large organizations; however, we cover how to replicate some of these workflows for Docker-based environments. We also detail how you can leverage Docker and Rancher to automate such workflows. Throughout this paper, we provide detailed examples of each step necessary to implement your own CI system.

We hope that by following this guide, you will be able to apply some of these ideas and make use of tools such as Docker and Rancher to create continuous integration and deployment pipelines onto which you can graft custom process as they make sense for your organization.

Before we begin, a note of caution: Both Docker and Rancher are evolving rapidly and therefore we expect some API and implementation inconsistencies with different versions of these platforms. For reference, we're working with Golang 1.8, Docker 1.13.1+, Jenkins Version 2.32.2, docker-compose 1.11.1+ and Rancher 1.4.1+ for this guide.

Part 1: Continuous Integration

To kick things off, we start at the pipeline ingress, i.e., building source code. When any project starts off, building/compilation is not a significant concern as most languages and tools have well-defined and well documented processes for compiling source code. However, as projects and teams scale, and the number of dependencies increase, ensuring a consistent and stable build for all developers while ensuring code quality becomes a much bigger challenge. In this section, we will cover some of the challenges, best practices and how Docker can be used to implement them.

1.1 Challenges of Scaling Build Systems

Before we get into best practices, let's look at some of the challenges that arise in maintaining build systems. The first issue that your project will face as it scales out is *Dependency Management*. As developers pull in libraries and integrate source code against them it becomes important to track versions of each library being used by the code, ensure the same version is being used by all parts of your project, test upgrades to library versions and push tested updates to all parts of your project.

A related but slightly different problem is to manage environment dependencies. This includes IDE and IDE configurations, tools versions (e.g. Maven version, Python version) and configuration e.g. static analysis rule files, code formatting templates. Environmental dependency management can get tricky because sometimes different parts of the project have conflicting requirements. Unlike conflicting code level dependencies, it is often not possible or easy to resolve these conflicts. For example, in a recent project we used fabric for deployment automation and s3cmd for uploading artifacts to Amazon S3. Unfortunately, the latest version of fabric required Python2.7 whereas s3cmd required Python2.6. A fix required us to either switch to a beta version of s3cmd or an older version of fabric.

Lastly, a major problem that every large project faces is build times. As projects grow in scope and complexity, more and more languages get added. Tests get added for various components which are all interdependent. For example, if you have a shared

database then tests which mutate the same data cannot be run at the same time. In addition, we need to make sure that tests setup expected state prior to execution and clean up after themselves when they finish. This lead to builds that can take anything from minutes to hours which either slows down development or leads to a dangerous practice of skipping test runs.

1.2 Solutions and Best Practices

To solve all these problems a good build system needs to support the following requirements (among others):

1. Repeatability
    - We must be able to generate/create similar (or identical) build environments with the same dependencies on different developer machines and automated build servers.
2. Centralized Management
    - We must be able to control the build environment for all developers and build servers from a central code repository or server. This includes setting up the build environment as well as updates overtime.
3. Isolation
    - The various sub-components of the project must be built in isolation other than well-defined shared dependencies.
4. Parallelization
    - We must be able to run parallel builds for sub-components.

To support the Repeatability requirement, we must use centralized dependency management. Most modern languages and development frameworks have support for automated dependency management. Maven is used extensively with Java and a few other languages, Python uses pip and Ruby has Bundler. All these tools have a very similar paradigm, where you would commit an index file (pom.xml, requirements.txt or gemfile) into your source control. The tool can then be run to download dependencies onto the build machine. We can manage the index files centrally after testing them and then push out the change by updating the index in source control. However, there remains the issue of managing environmental dependencies. For example, the correct version of Maven, Python and Ruby must be installed. We also need to ensure that the tools are run by developers. Maven automates the check for

dependency updates but for pip and Bundler we must wrap our build commands in scripts which trigger a dependency update run.

To setup the dependency management tools and scripts, most small teams just use documentation and leave the onus on developers. This however, does not scale to large teams especially if the dependencies are updated over time. Further complicating matters is the fact that installation instructions for these tools can vary by platform and OS of the build machines. You can use orchestration tools such as Puppet or Chef to manage installation of dependencies and setting up configuration files. Both Puppet and Chef allow for central servers or shared configuration in source control to allow centralized management. This allows you to test configuration changes ahead of time and then push them out to all developers. However, these tools have some drawbacks, installing and configuring Puppet or Chef is non-trivial and full featured versions of these tools are not free. In addition, each has its own language for defining tasks. This introduces another layer of management overhead for IT teams as well as developers. Lastly, orchestration tools do not provide isolation hence conflicting tool versions are still a problem and running parallel tests is still an open problem.

To ensure component isolation and reduce build times, we can use an automated virtualization system such as Vagrant. Vagrant can create and run virtual machines (boxes) which can isolate the build for various components and allow for parallel builds. The Vagrant configuration files can be committed into source control and pushed to all developers when ready to ensure centralized management. In addition, boxes can be tested and deployed to an Atlas for all developers to download. This still has the drawback that you will need a further layer of configuration to setup Vagrant and that virtual machines are a very heavy weight solution for this problem. Each VM runs an entire OS and network stack just to contain a test run or compiler. Memory and Disk resources need to be partitioned ahead of time for each of these VMs.

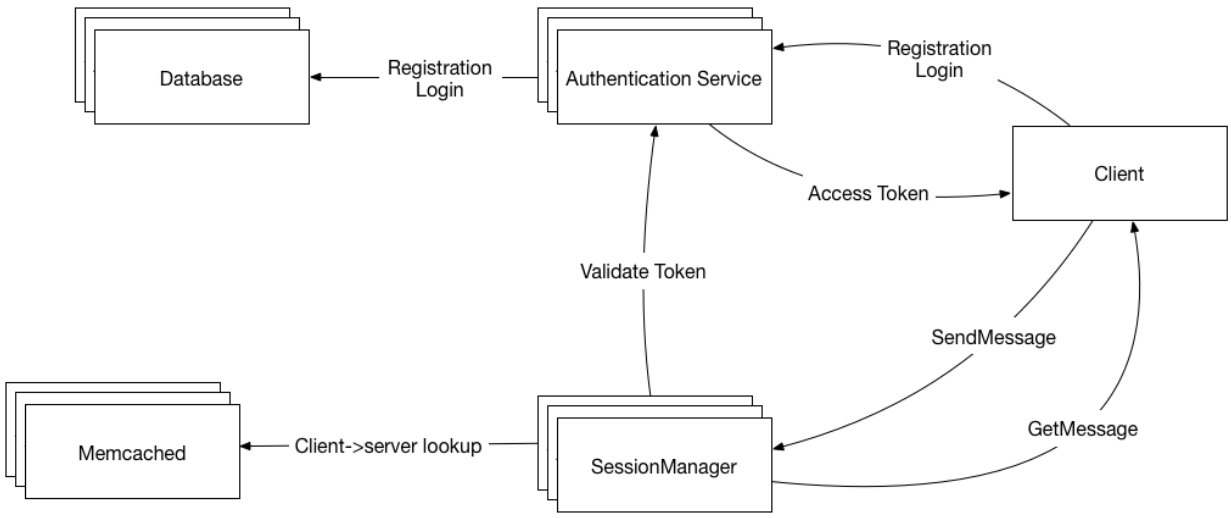
Despite the caveats and drawbacks, using Dependency Management (Maven, pip, Bundler), orchestration (Puppet, Chef) and virtualization (Vagrant), we *can* build a stable, testable centrally managed build system. Not all projects warrant the entire stack of tool; however, any long-running large project will need this level of automation.

1.3 Leveraging Docker for Build systems

Docker and its ecosystem of tools can help us target the requirements above without the large investment of time and resources to support all the tools mentioned above. In this section, we'll go through the steps below for creating containerized build environments for applications.

1. Containerizing your build environment
2. Packaging your application with Docker
3. Using Docker Compose for creating build environments

To illustrate the use of Docker in build pipelines, for this (and subsequent chapters) we'll be using a sample application called go-messenger. To follow along you can fetch the application from Github. The major data flows of the system are shown below. The application has two components; a RESTful authentication server written in Golang and a session manager which accepts long running TCP connections from clients and routes messages between clients. For the purposes of this paper, we will be concentrating on the RESTful Authentication Service (go-auth). This sub-system consists of an array of stateless web-servers and a database cluster to store user information.



2.3.1 Containerizing your build environment

The first step in setting up the build system is to create a container image with all tools required to build the project. The Docker file for our image is shown below and is also available here. Since our application is written in Go, we are using the official golang image and installing the govendor dependency management tool. Note that if you are using Java for your project, a similar "build container" can be created with Java base image and installation of Maven instead of govendor.

```
from golang:1.8
# Install govendor
RUN go get -u github.com/kardianos/govendor
Add compile.sh /tmp/compile.sh
CMD /tmp/compile.sh
```

We then add a compile script which puts all the steps required to build and test our code in one place. The script shown below downloads dependencies using govendor restore, standardizes formatting using the go fmt command, runs tests using the "go test" command and then compiles the project using go build.

```
#!/bin/bash
set -e
# Set directory to where we expect code to be
cd /go/src/${SOURCE_PATH}
echo "Downloading dependencies"
govendor sync
echo "Fix formatting"
go fmt ./...
echo "Running Tests"
go test ./...
echo "Building source"
go build
echo "Build Successful"
```

To ensure repeatability, we can use Docker containers with all tools required to build a component into a single, versioned container image. This image can be downloaded

from Dockerhub or built from Dockerfile (`docker build -t go-builder:1.8 .`). Now all developers (and build machines) can use the container to build any go project using the following command:

```
docker run --rm -it \
        -v $PWD:/go/src/github.com/[USERNAME]/[PROJECT]/[SUB-CDIRECTORY]/ \
        -e SOURCE_PATH=github.com/[USERNAME]/[PROJECT]/[SUB-CDIRECTORY]/ \
        usman/go-builder:1.8
```

In the above command, we are running the usman/go-builder image version 1.8, and mounting our source code into the container using the -v switch and specifying the SOURCE_PATH environment variable using the -e switch. To test the go-builder on our sample project, you can use the commands below to run all the steps and create an executable file called go-auth in the root directory of the go-auth project.

```
git clone git@github.com:usmanismail/go-messenger.git
cd go-messenger/go-auth
docker run --rm -it \
        -v $PWD:/go/src/go-messenger/go-auth/ \
        -e SOURCE_PATH=go-messenger/go-auth/ \
        usman/go-builder:1.8
```

An interesting side-effect of isolating all source from build tools is that we can easily swap out build tools and configuration. For example, in the commands above we have been using golang 1.8. By changing **go-builder:1.8** to **go-builder:1.5,** you can test the impact of using golang 1.5 on the project. In order to centrally manage the image used by all developers, we can deploy the latest tested version of the builder container to a fixed version (i.e. **latest**) and make sure all developers use **go-builder:latest** to build the source code. Similarly, if different parts of our project use different versions of build tools we can use different containers to build them without worrying about managing multiple language versions in a single build environment. For example, our earlier Python problem could be mitigated by using the official Python image which supports various Python versions.

2.3.2 Packaging your application with Docker

If you would like to package the executable in a container of its own, add a dockerfile with the content shown below and run "**docker build -t go-auth .**" In the dockerfile we are adding the binary output from the last step into a new container and exposing the 9000 port for application to accept incoming connections. We also specify the entry point to run our binary with the required parameters. Since Go binaries are self-contained, we're using a stock Ubuntu image, however, if your project requires run time dependencies they can be packaged into the container as well. For example, if you were generating a war file you could use a tomcat container.

```
FROM ubuntu
ADD ./go-auth /bin/go-auth
EXPOSE 9000
ENTRYPOINT ["/bin/go-auth","-l","debug","run","-p","9000"]
```

2.3.3 Using Docker Compose for build environments

Now that we have our project building repeatably in a centrally managed container which isolates the various components, we can also extend the build pipeline to run integration tests. This will also help us highlight the ability of Docker to speed up builds using parallelization. One major reason that tests cannot be parallelized is because of shared data stores. This is especially true for integration tests where we would not typically mock out external databases. Our sample project has a similar issue, we use a MySQL database to store users. We would like to write a test which ensures that we can register a new user. The second time a registration is attempted for the same user we expect a conflict error. This forces us to serialize tests so that we can cleanup our registered users after a test is complete before starting a new one.

To setup isolated, parallel builds we can define a Docker Compose template (docker-compose.yml) as follows. We define a database service which uses the MySQL official image with required environment variables. We then create a GoAuth service with the container we created to package our application and link the database container to it. Note that we use a variable substitution for GO_AUTH_VERSION. If this variable is

specified in the environment then compose will use that as the tag for the go-auth image, otherwise it will use the default value *latest*.

```
version: "3"
services:
  Goauth:
    image: usman/go-auth:${GO_AUTH_VERSION:-latest}
    ports:
      - "9000:9000"
    command:
      - "-l"
      - "debug"
      - "run"
      - "--db-host"
      - "db"
      - "-p"
      - "9000"
  db:
    image: mysql
    environment:
      MYSQL_ROOT_PASSWORD: rootpass
      MYSQL_DATABASE: messenger
      MYSQL_USER: messenger
      MYSQL_PASSWORD: messenger
```

With this docker-compose template defined we can run the application environment by running **docker-compose up**. We can then simulate our integration tests by running the following curl command. It should return *200 OK* the first time and *409 Conflict* the second time. The `service_ip` parameter is either localhost if you are running on linux or the IP of the Docker virtual machine if you are running on OSX. You can lookup the service_ip by running:

```
service_ip=$(docker-machine url $(docker-machine active) | cut -d : -f 2 | cut -c 3-)
curl -i -silent -X PUT -d userid=USERNAME -d password=PASSWORD ${service_ip}:9000/user
```

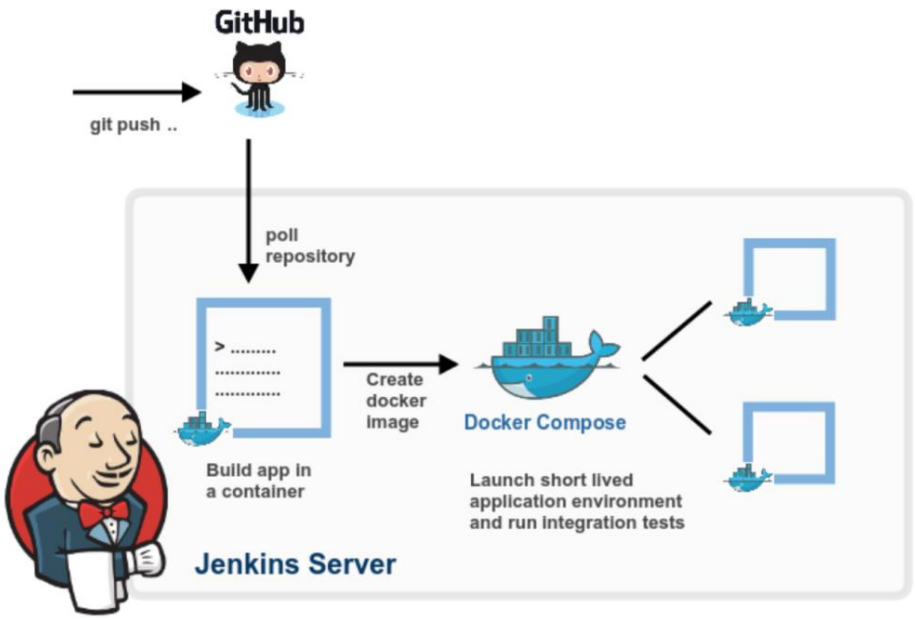
Lastly, after running tests, we can run **docker-compose rm** to clean up the entire application environment.

To run multiple isolated versions of the application, we need to update docker-compose template to add the service database1 and goauth1 with identical configurations to their counterparts. The only change is that in Goauth1 we need to change the ports entry from 9000:9000 to 9001:9000. This is so that the publicly exposed port of the application does not conflict. The complete template is available here. When you run **docker-compose up** now you can run the two integration test runs in parallel. Something like this can be effectively used to speed up builds for a project with multiple independent sub-components, e.g., a multi-module Maven project.

```
curl -i -silent -X PUT -d userid=USERNAME -d password=PASSWORD ${service_ip}:9000/user
... 200 OK
curl -i -silent -X PUT -d userid=USERNAME -d password=PASSWORD ${service_ip}:9001/user
... 200 OK
curl -i -silent -X PUT -d userid=USERNAME -d password=PASSWORD ${service_ip}:9001/user
... 409 Conflict
curl -i -silent -X PUT -d userid=USERNAME -d password=PASSWORD ${service_ip}:9000/user
... 409 Conflict
```

1.4 Creating a Continuous Integration Pipeline

Now that we have the build setup, let's create a continuous integration pipeline for our sample application. This will help ensure that best-practices are followed and that conflicting changes are not acting together to cause problems. However, before we get into setting up the continuous integration for our code we will spend a little time talking about how to partition out code into branches.



1.4.1 Branching Model

As we automate our continuous integration pipeline, an important aspect to consider is the development model followed by the team. This model is often dictated by how the team uses the version control system. Since our application is hosted in a git repository, we're going to use git-flow model for branching, versioning and releasing our application. It's one of the most commonly used models for git based repositories. Broadly, the idea is to maintain two branches; a develop(ment) branch and a master branch. Whenever we want to work on a new feature, we create a new branch from develop and when the feature work is complete, it is merged back into it. All feature branches are managed individually by developers working on those features. Once code is committed to the develop branch, CI servers are responsible for making sure the branch always compiles, passes automated tests and is available on a server for QA testing and review. Once we're ready to release our work, a release is created from the develop branch and is merged into the master branch. The specific commit hash being released is also tagged with a version number. Tagged releases can then be pushed to Staging/Beta or Production environments.

We are going to be using the gitflow tool to help manage our git branches. To install git-flow, follow the instructions here. Once you have git-flow installed you can

configure your repository by running the *git flow init* command as shown below. Git flow is going to ask a few questions and we recommend going with the defaults. Once you execute the git-flow command, it will create a develop branch (if it didn't exist) and check it out as the working branch.

```
$ git flow init
Which branch should be used for bringing forth production releases?
 - master
Branch name for production releases: [master]
Branch name for "next release" development: [develop]
How to name your supporting branch prefixes?
Feature branches? [feature/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? []
```

Now, let's create a new feature using git flow by typing *git flow feature start [feature-name]*. It's a common practice to use ticket/issue id as the name of the feature. For example, if you are using something like Jira and working on a ticket, the ticket Id (e.g., MSP-123) can become the feature name. You'll notice that when you create a new feature with git-flow, it will automatically switch to the feature branch.

```
git flow feature start MSP-123
Switched to a new branch 'feature/MSP-123'
Summary of actions:
- A new branch 'feature/MSP-123' was created, based on 'develop'
- You are now on branch 'feature/MSP-123'
Now, start committing on your feature. When done, use:
git flow feature finish MSP-123
```

At this point you can do all the work needed for the feature and then run your automated suite of tests to make sure that everything is in order. Once you are ready to ship your work, simply tell git-flow to finish the feature. You can do as many commits

as you need for the feature. For our purposes, we're just going update the README file and finish off the feature by typing "git flow feature finish MSP-123".

```
Switched to branch 'develop'
Updating 403d507..7ae8ca4
Fast-forward
 README.md | 1 +
 1 file changed, 1 insertion(+)
Deleted branch feature/MSP-123 (was 7ae8ca4).
Summary of actions:
- The feature branch 'feature/MSP-123' was merged into 'develop'
- Feature branch 'feature/MSP-123' has been removed
- You are now on branch 'develop'
```

Note that git flow merges the feature in 'develop', deletes the feature branch and takes you back to the develop branch. At this point you can push your develop branch to remote repository (*git push origin develop:develop*). Once you commit to the develop branch the CI server takes over to run the Continuous integration pipeline. Note, for a larger team, an alternative and a more suitable model would be to push feature branches to remote before finishing them off, getting them reviewed and using Pull requests to merge them into develop.

1.4.2 Creating CI pipeline with Jenkins

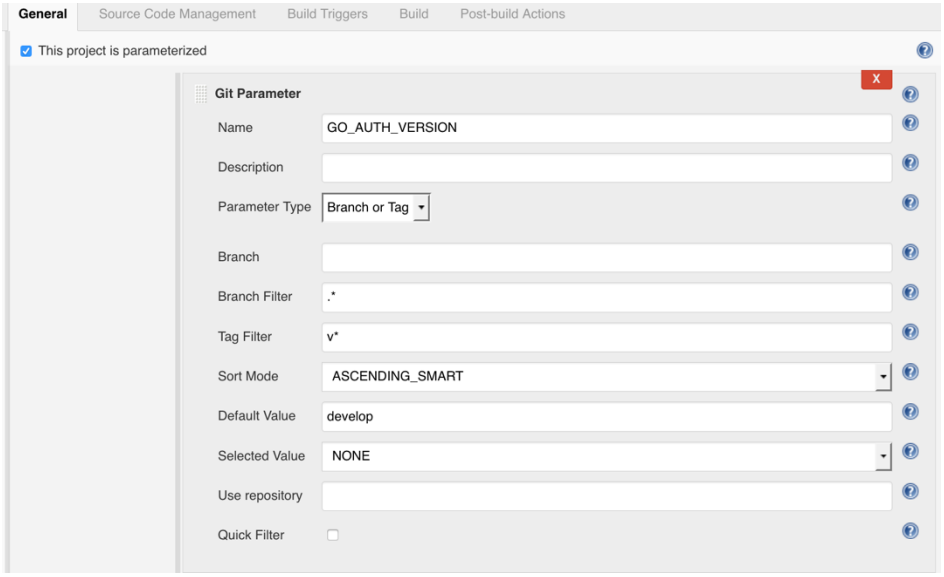
For this section, we assume that you have gotten a Jenkins cluster up and running. If not, you can use the official Jenkins image from Here or read more about setting up a scalable Jenkins cluster here. Once you have Jenkins running, we need the following plugins and dependencies installed on your Jenkins server:

- Jenkins Plugins 2.32.2+
    - Git Parameter Plugin 0.8.0+
    - Parameterized Trigger Plugin 2.33+
    - Copy Artifact Plugin 1.38.1+
    - Build Pipeline Plugin 1.5.6+
    - Mask Passwords Plugin 2.9+
- Docker 1.13.1+
- Docker Compose 1.11.1+

Once you have setup the requisite plugins we can create the first three jobs in our Build Pipeline: compile, package and integration test. These will serve as the starting point of our continuous integration and deployment system.

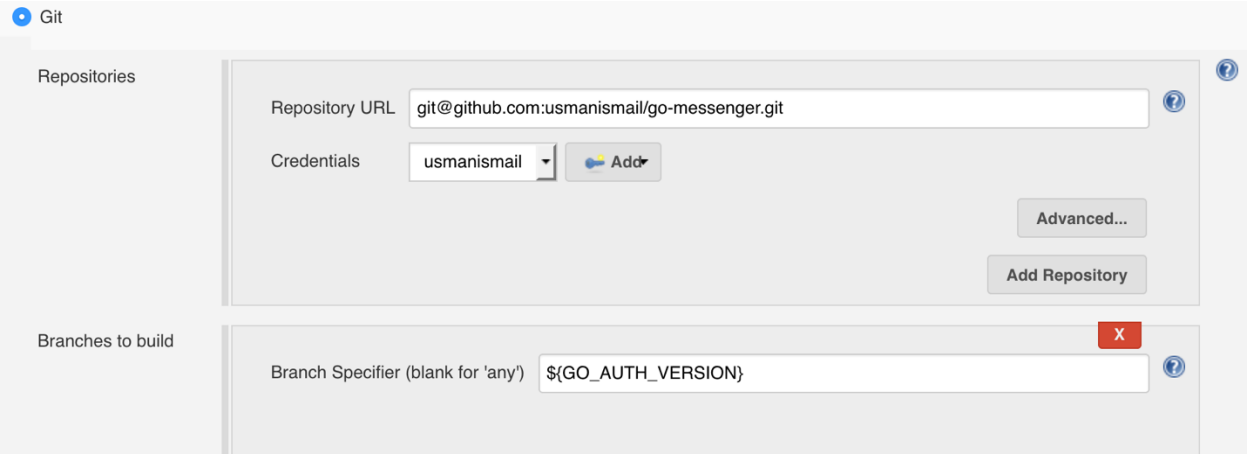
1.4.2.1 Building application

The first job in the sequence will checkout the latest code from source control after each commit and ensure that it compiles. It will also run unit tests. To setup the first job for our sample project select *New Item* > *Freestyle Project*. Once in the project configuration view select the *General* tab and the "This project is parameterized" option. Add a git parameter called GO_AUTH_VERSION, set the parameter type to *Branch or Tag*. Next select *Advanced* and configure the parameter to pick up any tags matching "v*" (e.g., v2.0) using the *Tag Filter* setting. Set the *Default Value* setting to *develop* (branch). This is quite useful for getting a list of version tags from Git and populating a selection menu for the job. If the job is automatically triggered and no value is specified, the value of GO_AUTH_VERSION defaults to develop.



Next, In the *Source Code Management section* add the *repository url*, specify the branch as ${GO_AUTH_VERSION} so that manual builds will use the git parameter to select branch or tag to build and set the polling interval. With this, Jenkins will keep tracking our develop branch for any changes to automatically trigger the first job in our CI (and

CD) pipeline. Note that default value of the GO_AUTH_VERSION e.g. develop branch will be used for auto-detected changes.

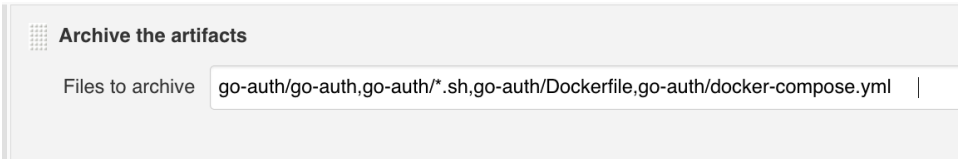


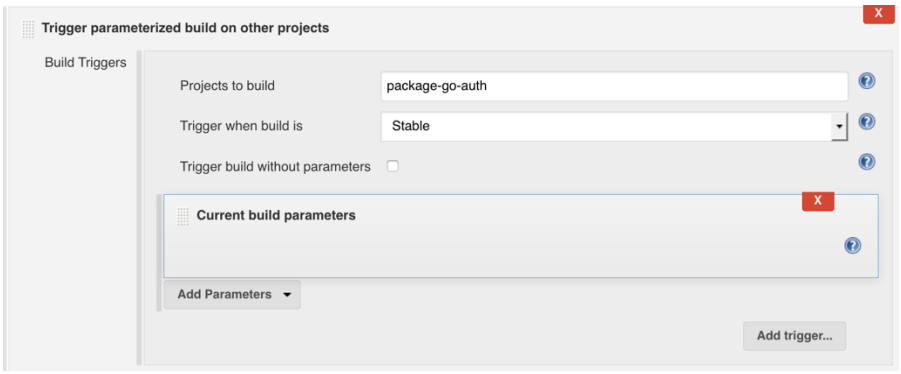
Now in the *Build* section select *Add Build Step* > *Execute Shell* and copy the docker run command from earlier in the chapter. This will get the latest code from Github and build the code into the go-auth executable. Note that this requires docker to be installed and running. If you are on a linux server you may also need to add sudo to the command for the docker client to be able to access the docker daemon.

Following the build step, we need to add two post-build steps, *Archive the Artifacts* to archive the go-auth binary and the helper scripts in the project. We need the specify the following artifacts to be archived for this job.

```
go-auth/go-auth,go-auth/*.sh,go-auth/Dockerfile,go-auth/docker-compose.yml,deploy/*
```

Then use *Trigger parameterized builds* to kick off the next job in the pipeline as shown below. When adding the *Trigger parameterized build* action, make sure to add *Current build parameters* from *Add Parameters*. This will make all the parameters (e.g., GO_AUTH_VERSION) for the current job available for the next job. Note the name to use for the downstream job in the trigger parameterized build section as we'll need it in the following step.





The log output form the build job should look something like following. You can see that we use a dockerized container to run the build. The build will use go fmt to fix any formatting inconsistencies in our code and to run our unit tests. If any tests fail or if there are compilation failures, Jenkins will detect the failure. Furthermore, you should configure notifications via email or chat integrations (e.g. Hipchat or Slack) to notify your team if the build fails so that it can be fixed quickly.

```
Started by an SCM change
Building in workspace /var/jenkins/jobs/build-go-auth/workspace
 > git rev-parse --is-inside-work-tree # timeout=10
Fetching changes from the remote Git repository
 > git config remote.origin.url https://github.com/usmanismail/go-messenger.git # timeout=10
Fetching upstream changes from https://github.com/usmanismail/go-messenger.git
 > git --version # timeout=10
 > git -c core.askpass=true fetch --tags --progress https://github.com/usmanismail/go-
messenger.git +refs/heads/*:refs/remotes/origin/*
 > git rev-parse refs/remotes/origin/develop^{commit} # timeout=10
 > git rev-parse refs/remotes/origin/origin/develop^{commit} # timeout=10
Checking out Revision 89919f0b6cd089342b1c5b7429bca9bcda994131 (refs/remotes/origin/develop)
 > git config core.sparsecheckout # timeout=10
 > git checkout -f 89919f0b6cd089342b1c5b7429bca9bcda994131
 > git rev-list 7ae8ca4e8bed00cf57a2c1b63966e208773361b4 # timeout=10
[workspace] $ /bin/sh -xe /tmp/hudson1112600899558419690.sh
+ echo develop
```

```
develop
+ cd go-auth
+ docker run --rm -v /var/jenkins/jobs/build-go-auth/workspace/go-auth:/go/src/go-
messenger/go-auth/ -e SOURCE_PATH=go-messenger/go-auth/ usman/go-builder:1.8
Downloading dependencies
Fix formatting
Running Tests
?       github.com/usmanismail/go-messenger/go-auth [no test files]
?       github.com/usmanismail/go-messenger/go-auth/app    [no test files]
?       github.com/usmanismail/go-messenger/go-auth/database      [no test files]
?       github.com/usmanismail/go-messenger/go-auth/logger [no test files]
ok      github.com/usmanismail/go-messenger/go-auth/user   0.328s
Building source
Build Successful
Archiving artifacts
Warning: you have no plugins providing access control for builds, so falling back to legacy
behavior of permitting any downstream builds to be triggered
Triggering a new build of package-go-auth
Finished: SUCCESS
```

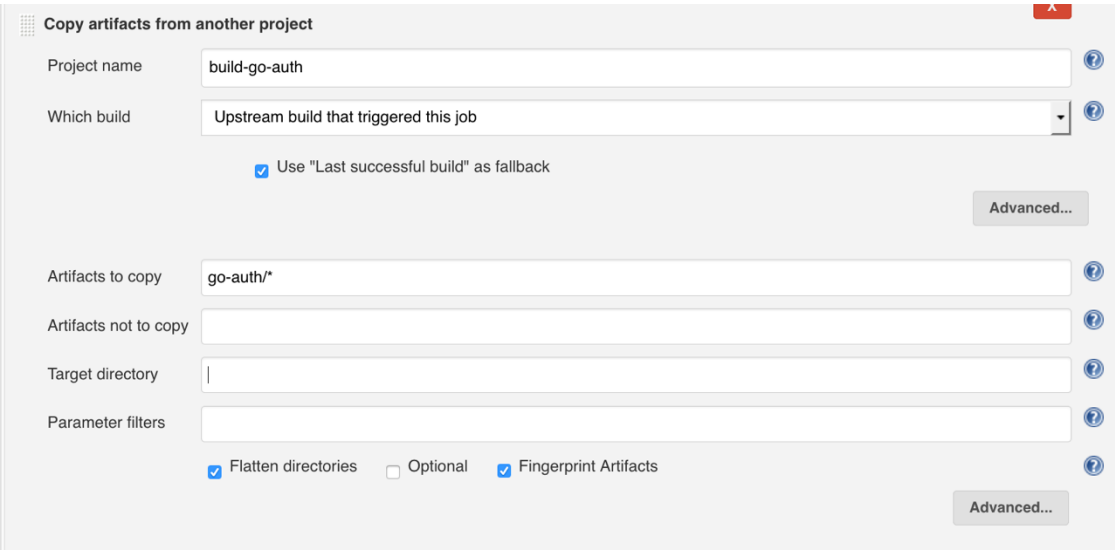
1.4.2.2 Packaging application

Once we have compiled the code, we can package it into a Docker container. To create the package job select, *New Item* > *Freestyle Project* and give your second job a name matching what you specified in the previous job. As before, this job is also going to be a parameterized build with a GO_AUTH_VERSION parameter. Note that for this and all subsequent jobs, the GO_AUTH_VERSION is simply a string parameter with a default value of "develop". The expectation here is that this value would be coming from the upstream.

As before add a build step to execute shell. Note that you do not have specify SCM settings as we are going to pull the required binary and scripts from the artifacts generated in the previous build. Note that we build as the untagged usman/go-auth and then retag so that we can run integration tests later bring up the env with an untagged container.

```
echo ${GO_AUTH_VERSION}
chmod +x go-auth
docker build -t usman/go-auth:${GO_AUTH_VERSION} .
```

To build the Docker container, we also need the executable we built in the previous step. To do this we add a build step to copy artifacts from the upstream build. This will ensure that we have the executable available for the Docker build command which can be packaged into a Docker container. Note that we select flatten directories to make sure that all artifacts are copied into the root of the current project.



Note that we're using the GO_AUTH_VERSION variable to tag the image we're building. By default, for changes in develop branch, it would always build usman/go-auth:develop and overwrite the existing image. In the next chapter, we'll revisit this pipeline for releasing new versions of our application.

As before use the *Trigger parameterized builds* (with *Current build parameters)* post-build action to trigger the next job in the pipeline which will run integration tests using

the Docker container we just built and the Docker Compose template that we detailed earlier in the chapter.

1.4.2.3 Running integration tests

To run integration tests, create a new job. As with the package job, this job needs to be a parameterized build with the GO_AUTH_VERSION string variable. Next copy artifacts from the build job. This time we will use the Docker Compose template above to bring up a multi-container test environment and run integration tests against our code. Integration tests (unlike unit tests) are typically kept entirely separate from the code being tested. To this end, we will use a shell script which runs http queries against our test environment. In your execute shell command, change the directory to go-auth and run *integration-test.sh*.

```
echo ${GO_AUTH_VERSION}
chmod +x integration-test.sh
./integration-test.sh
```

The contents of the script are available here. We use Docker Compose to bring up our environment, then use curl to send http requests to the container we brought up. The logs for the job will be similar to the ones shown below. Compose will launch a database container, and link it to the goauth container. Once the database is connected you should see a series of "*Pass: ...*" as the various tests are run and verified. After the tests are run, the compose template will clean up after itself by deleting the database and go-auth containers.

```
./integration-test.sh
Creating goauthintegrationtest_Goauth_1
Creating goauthintegrationtest_db_1
Using Service IP 172.19.0.2
Pass: Register User
Pass: Register User Conflict
Stopping goauthintegrationtest_db_1 ...
Stopping goauthintegrationtest_Goauth_1 ...
```

```
Stopping goauthintegrationtest_Goauth_1 ... done
Removing goauthintegrationtest_db_1 ...
Removing goauthintegrationtest_Goauth_1 ...
Removing goauthintegrationtest_db_1 ... done
Removing goauthintegrationtest_Goauth_1 ... done
Going to remove goauthintegrationtest_db_1, goauthintegrationtest_Goauth_1
Finished: SUCCESS
```

With the three jobs now setup, you can create a new Build Pipeline view by selecting the + tab in the Jenkins view and selecting the build pipeline view. In the configuration screen that pops up, select your compile/build job as the initial job and select OK. You should now see your CI pipeline take shape. This gives a visual indication of how each commit is progressing through your build and deployment pipeline.

When you make changes to the develop branch, you'll notice that the pipeline is automatically triggered by Jenkins. To manually trigger the pipeline, select your first (build) job and run. It would ask you to select the value of the git parameter (e.g., GO_AUTH_VERSION). Not specifying any will result in the default value and run the CI pipeline against the latest in the develop branch. You can also just click 'Run' in the pipeline view.

Let's quickly review what we've done so far. We created a CI pipeline for our application with the following steps:

1. Use git-flow to add new features and merge them into develop
2. Track changes on develop branch and build our application in a containerized environment
3. Package our application in a docker container
4. Spin up short-lived environments using Docker Compose
5. Run integration tests and tear down environments

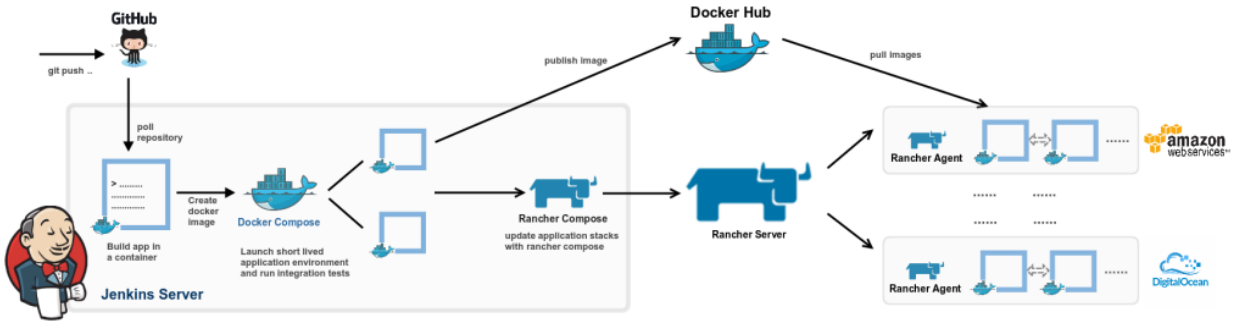
With the above CI pipeline, every time a new feature (or fix) is merged into the develop branch, all of the above steps are executed by our CI pipeline to create the "usman/go-auth:develop" Docker image. Further as we build out a deeper pipeline in upcoming chapters which integrates deployment. You will also be able to use this view to promote application versions to various deployment environments as they clear testing phases.

1.5 Summary

In this chapter, we've seen how to leverage Docker to create a continuous integration pipeline for our project which is centrally managed, testable, and repeatable across machines and in time. We were able to isolate the environmental dependencies for various components as needed. This forms a starting point to a longer Docker-based build and deployment pipeline which we'll continue to build and document in the next chapter. The next step in our pipeline is to setup continuous deployment. We will show how to use Rancher to deploy an entire server environment to run our code. We will also cover best-practices for how to setup a long running testing environment and deployment pipeline for large scale projects.

Part 2: Continuous Deployment

In the previous section, we have seen how to leverage Docker to create a continuous integration pipeline. As part of that, we used Docker to create a centrally managed build environment which can be rolled out to any number of machines. We then setup the environment in Jenkins CI and automated the continuous building, packaging and testing of the source. In this chapter, we will take the pipeline further (shown below) and see how we can continuously deploy the project to a long-running testing environment. This will allow manual human testing of the code in addition to automated acceptance testing. This environment will also allow you to get your customers' or QA's eyes on the latest changes before they hit production. Further, this will give you a good idea of how to build and deploy to production environments which we will cover in the next chapter.



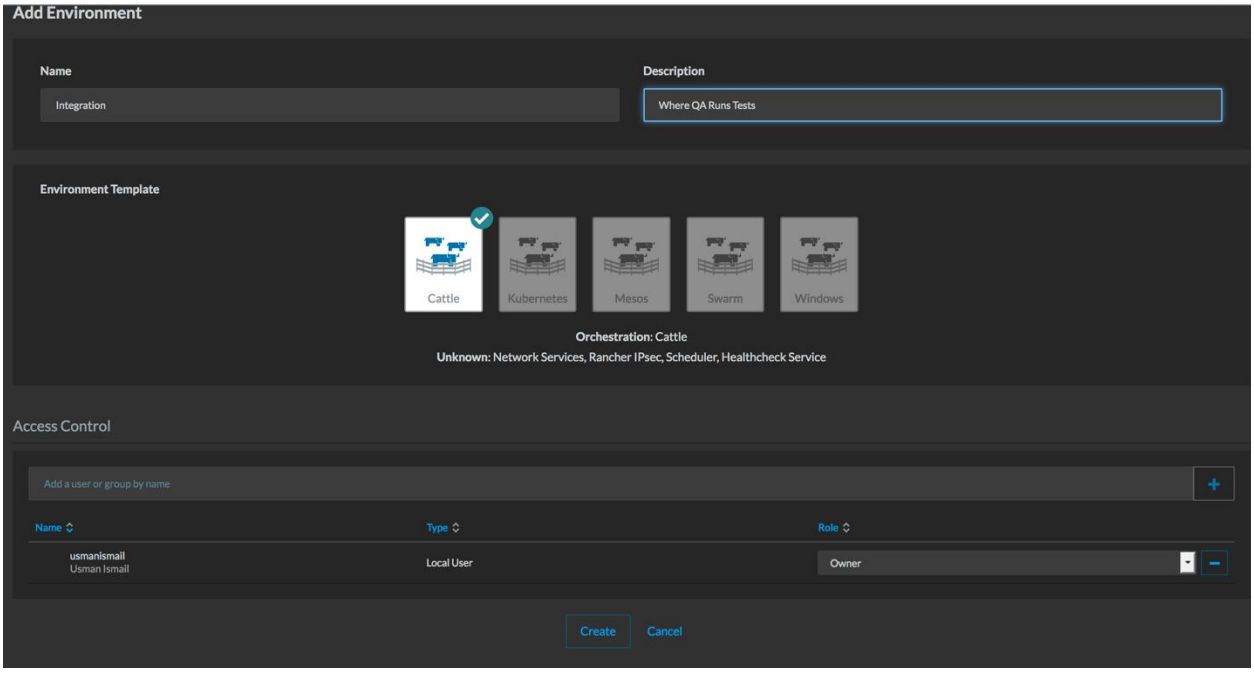
2.1 Creating long-running application environments

After we've built and tested our application, we can now deploy it to a long-running, potentially externally facing environment. This environment will allow Quality Assurance (QA) or customers to see and test the latest changes before they make their way to production. This environment is an important step on the road to production as it allows us to unearth bugs that are only seen with real world use and not automated integration tests. We normally call this environment the QA or Integration environment however, you may name your environments as you wish. We'll go through the steps below for creating our integration environment:

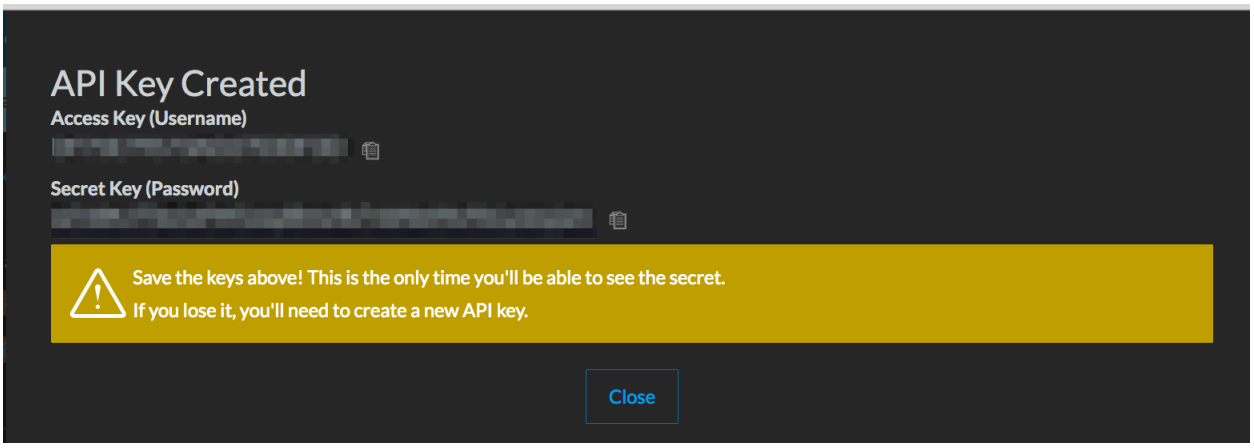
1. Create an Integration environment in Rancher
2. Define Docker Compose and Rancher Compose templates
3. Create application stack with Rancher
4. Manage DNS records with Rancher and AWS Route53
5. Add support for HTTPS

2.1.1 Creating an Integration environment in Rancher

In the Rancher UI, go to the top left corner and select *Manage Environments* and *Add Environment*. In the resulting screen (shown below) add the Name (Integration) and optionally a description for each of the environments. You also need to select a list of users and organizations that have access to the environments.



Once you have your environment setup, select the *Integration* environment from the drop down in the top left corner of the screen. We can now create the application stack for the integration environment. Before we do so let's get our Rancher deployment's API keys. Select *API* > *Keys* from the top menu and *Add Account API Key*. This will load a pop-up screen which allows you to create a named API Key pair. We need the key in subsequent steps to use Rancher Compose to create our test environments. We will create key pair named *JenkinsKey* to run rancher compose from our Jenkins instance. Copy the key and secret for use later as you will not be shown these values again. Note that API keys are specific to the environment and hence you will have to create a new key for each environment.



2.1.2 Defining Compose templates

In the previous chapter, we created a Docker Compose template to define the container types required for our project. The compose template (docker-compose.yml) is shown below. We will be using the same Docker Compose template as before but with the addition of auth-lb service. This will add a load-balancer in front of our go-auth service and split traffic across all the containers running the service. Having a load balancer in front of our service is essential to insure availability and scalability, as it continues to serve traffic even if one (or more) of our service containers die. Additionally, it also spreads the load on multiple containers which may be running on multiple hosts. Rancher does not yet support compose file version 3, hence we have to set the version to 2 and remove the environment variable comprehension and fix the version of go-auth.

```
version: "2"
services:
  Goauth:
    image: usman/go-auth:${auth_version}
    ports:
      - "9000:9000"
    command:
      - "-l"
      - "debug"
      - "run"
      - "--db-host"
      - "db"
      - "-p"
      - "9000"
  db:
    image: mysql
    environment:
      MYSQL_ROOT_PASSWORD: rootpass
      MYSQL_DATABASE: messenger
      MYSQL_USER: messenger
      MYSQL_PASSWORD: messenger

  auth-lb:
    image: rancher/lb-service-haproxy:v0.6.2
    ports:
```

```
    - 9000:9000/tcp
    labels:
      io.rancher.container.agent.role: environmentAdmin
      io.rancher.container.create_agent: 'true'
```

We are using Rancher Compose to launch the environment in a multi-host environment, this more closely mirrors production and also allows us to test integration with various services, e.g. Rancher and Docker Hub etc. Unlike our previous Docker Compose based environment which was explicitly designed to be independent of external services and launched on the CI server itself without pushing images to dockerhub.

Now that we are going to use Rancher Compose to launch a multi-host test environment instead of Docker Compose, we also need to define a rancher compose template. Create a file called *rancher-compose.yml* and add the following content. In this file, we are defining that we need two containers of the auth service, one container running the database and another running the load-balancer container.

```
Version: “2”

auth-service:
  scale: 2
mysql-master:
  scale: 1
auth-lb
  scale: 1
```

Next, we will add a health check to the auth-service to make sure that we detect when containers are up and able to respond to requests. For this we will use the /health URI of the go-auth service. The auth-service section of rancher-compose.yml should now look something like this:

```
auth-service
  scale: 1
  start_on_create: true
    health_check:
```

```
      response_timeout: 2000
      healthy_threshold: 2
      port: 9000
      unhealthy_threshold: 3
      initializing_timeout: 60000
      interval: 2000
      strategy: recreate
      request_line: GET "/health" "HTTP/1.0"
      reinitializing_timeout: 6000
```

We are defining a health check on port 9000 of the service container which is run every 2 seconds (2000 milliseconds). The check makes a http request to the /health URI and 3 consecutive failed checks mark a container as unhealthy whereas 2 consecutive successes mark a container as healthy.

Similarly, we need to add the lb_config section to the load balancer rancher-compose.yml file so that it knows which service to target (GoAuth), which ports need to be exposed and how to verify the health of instances behind the load-balancer. The auth-lb section of rancher-compose.yml should look like this now:

```
auth-lb:
  scale: 1
  start_on_create: true
  lb_config:
    port_rules:
    - priority: 1
      protocol: http
      service: Goauth
      source_port: 9000
      target_port: 9000
  health_check:
    response_timeout: 2000
    healthy_threshold: 2
    port: 42
    unhealthy_threshold: 3
    interval: 2000
```

2.1.3 Creating an application stack with Rancher CLI

Now that we have our template defined we can use Rancher Compose to launch our environment. To follow along, simply checkout the go-messenger project and download the rancher CLI from rancher UI. To setup rancher-compose on your development machine, follow the instructions here. Please run ./rancher config to specify the rancher url, access key, secret key and default environment for all subsequent commands.

Once you have rancher CLI setup you can use the create command shown below to setup your integration environment. Note that this assumes that the docker-compose.yml and rancher-compose.yml that we create above are in the current directory.

```
git clone https://github.com/usmanismail/go-messenger.git
cd go-messenger/deploy

#replace rancher-compose with the latest version you downloaded from rancher UI
rancher --debug stack create messenger-int
```

In the UI, you should now be able to see the stack and services for your project. Note that "create" command creates the stack and also starts all the services.

To make sure everything is working, head over to the public IP for the host running the "auth-lb" service and create a user using the command shown below. You should get a

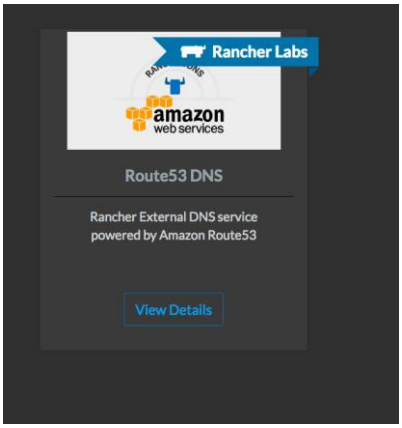
200 OK. Repeating the above request should return a 409-error indicating a conflict with an existing user in the database. At this point we have a basic integration environment for our application which is intended to be a long running environment.

```
curl -i -silent -X PUT -d userid=<TEST_USERNAME> -d password=<TEST_PASS>
<HOST_IP_FOR_AUTH_LB>:9000/user
```

2.1.4 Managing DNS records

Since this environment is meant to be long running and externally facing, we are going to be using DNS entries and HTTPS. This allows us to distribute the application outside corporate firewalls securely and also allows more casual users to rely on persistent DNS rather than IPs which may change. You may use a DNS provider of your choice; however, we are going to illustrate how to setup DNS entries in Amazon Route53. To do so, go to AWS Console > Route 53 > Hosted Zones and *Create Hosted Zone*. In the hosted zone, you will have to specify a domain name of your choice (e.g. gomessenger.com). While you are in the AWS console you can also create a user for Rancher to use to make Route53 updates. Go to AWS Console > IAMS > Users and select *Create New Users*. Keep the Access Key and Secret Key of this user handy as you will need later on. Once you have created the user account, you must add the AmazonRoute53FullAccess policy to the user so that it can make updates to route53.

Now that we have our Hosted Zone and IAMs user setup, we can add the Route53 integration to our Rancher Server. The detailed instructions on how to do so can be found here. In short, you need to browse to *Catalog* > *Library* on your rancher server and select Route 53 DNS. You will be asked to specify the Hosted Zone that you setup earlier, as well as the AWS Access and Secret Keys for you Rancher IAMs user with Route53 access. Once you enter the required information and click create, you should see new stack created in your environment with a service called route53.



This service will listen for Rancher events and catch any load balancer instance launches and terminations. Using this information, it will automatically create DNS entries for all the Hosts on which your load balancer containers are running. The DNS entries are of the form [Loadbalancer].[stack].[env].[domain], e.g.

```
goauth.integration.testing.gomessenger.com.
```

As more containers are launched and taken down on your various Rancher compute nodes the Route53 service will keep your DNS records consistent. This is essential for our integration test environments because as we will see later, we need to relaunch the environment containers in order to push updates as part of continuous deployment. With Route53 DNS integration we do not have to worry about getting the latest hostnames to our clients and testers.

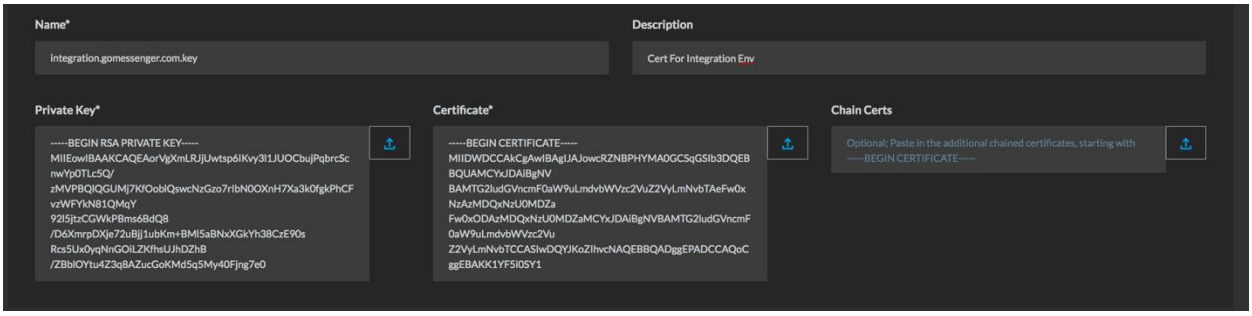
2.1.5 Enabling HTTPS

After creating DNS records for our environment, it is a good idea to support HTTPS. To do that, first, we need an SSL certificate for our domain. You can purchase a root SSL certificate for your domain from one the many trusted certificate authorities. If you don't have a certificate you can generate a self-signed certificate to complete the setup and replace it with a trusted one at a later time. The implication of a self-signed certificate is that any user will get a "This connection is untrusted" warnings in browsers, however, the communication is still encrypted. To generate the self-signed certificate, you will first need to generate the ssl key which you can do using the genrsa command of openssl. Then you can use the key file to generate the certificate using the req command. The steps to do so are listed below. It is also a good idea to print

and store the sha256 fingerprint of the certificate so that you can manually ensure that the same certificate is presented to you when making HTTPS requests. In the absence of a trusted certificate, manually matching fingerprints is the only way to ensure that there aren't any man-in-the-middle attacks.

```
openssl genrsa -out integration.gomessenger.com.key 2048
openssl req -new -x509                    \
    -key integration.gomessenger.com.key   \
    -out integration.gomessenger.com.crt  \
    -days 365 -subj /CN=integration.gomessenger.com
openssl x509 -fingerprint -sha256 -in integration.gomessenger.com.crt
SHA256 Fingerprint=E2:E5:86:09:F0:91:……
```

Now that you have the certificate and the private key file, you'll need to upload these into Rancher. We can upload certs by clicking the *Add Certificate* button in the *Certificates* Section of the *Infrastructure* tab in the Rancher UI. You need to specify a meaningful name for your certificate and optionally a description as well. Copy the contents of *integration.gomessenger.com.key* and *integration.gomessenger.com.crt* into the *Private Key* and *Certificate* fields respectively (or select Read from File and select the respective files). Once you have completed the form click save and wait a few moments for the certificate to become active.



Once the certificate is active we can add the HTTPS endpoint to our environment. In order to do so we have to modify our rancher-compose file to include the SSL port configuration. We add a second port (9001) to the *ports* section to make it accessible outside the load balancer container and we use the *io.rancher.loadbalancer.ssl.ports* label to specify that '9001' will be the public load balancer port with SSL termination.

Furthermore, since we are terminating SSL at the load balancer we can route requests to our actual service container using plain HTTP over the original 9000 port. We specify this mapping from 9001 to 9000 using the *io.rancher.loadbalancer.target.auth-service* label.

```
  auth-lb:
    image: rancher/lb-service-haproxy:v0.6.2
    ports:
    - 9000:9000/tcp
    - 9001:9001/tcp
    labels:
      io.rancher.container.agent.role: environmentAdmin
      io.rancher.container.create_agent: 'true'
  - auth-service:auth-service
  stdin_open: true
mysql-master:
  environment:
  ...
  ...
```

We also need to update the rancher-compose file to specify the SSL certificate we should use in the load balancer service for SSL termination. Add the *default_cert* parameter with the name of the certificate we uploaded earlier. After these changes you will need to delete and recreate your stack as there is currently no way to add these properties to a deployed stack.

```
auth-lb:
  scale: 1
  start_on_create: true
  lb_config:
    default_cert: integration.gomessenger.com.key
    port_rules:
    - priority: 1
      protocol: https
      service: Goauth
      source_port: 9000
      target_port: 9000
  health_check:
```

```
    response_timeout: 2000
    healthy_threshold: 2
    port: 42
    unhealthy_threshold: 3
    interval: 2000
```

Now to make sure everything is working, you can use the following curl command. When you try the same command with the https protocol specifier and the 9001 port you should see a failure complaining about the use of an untrusted certificate. You can use the *--insecure* switch to turn of trusted certificate checking and use https without it.

```
# Http Request
curl -i -silent -X PUT            \
    -d userid=<TEST_USERNAME>     \
    -d password=<TEST_PASS>       \
    http://integration.gomessenger.com:9000/user

# Https Request with secure checking
# Note Http(s) and 900(1)
curl -i -silent -X PUT            \
 -d userid=<TEST_USERNAME>        \
 -d password=<TEST_PASS>          \
 https://integration.gomessenger.com:9001/user
curl: (60) SSL certificate problem, verify that the CA cert is OK. Details:
error:14090086:SSL routines:SSL3_GET_SERVER_CERTIFICATE:certificate verify failed

# Https Request with insecure checking
curl -i -silent -X PUT            \
 --insecure                       \
 -d userid=<TEST_USERNAME>        \
 -d password=<TEST_PASS>          \
 https://integration.gomessenger.com:9001/user
```

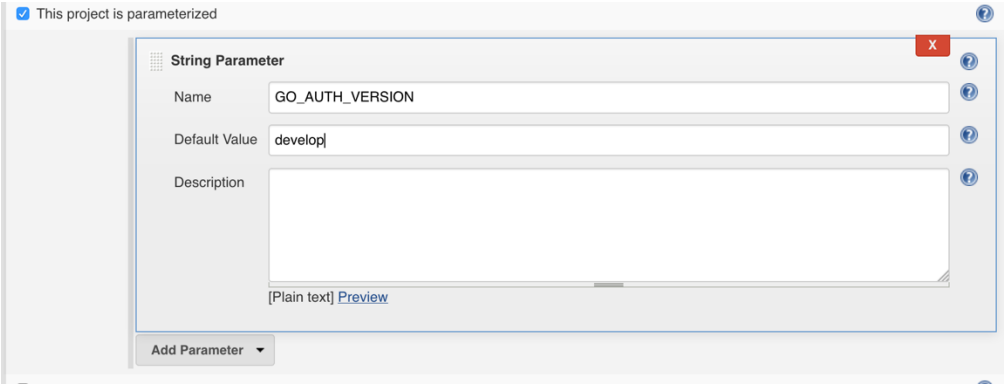
2.2 Creating a Continuous Deployment Pipeline

Now that we have created our test environment we can finally get back to the original intent of this chapter and build out a continuous deployment pipeline by extending our CI pipeline (from previous chapter) which built the application, packaged it into a container and ran integration tests against it.

2.2.1 Publishing Docker images

We're going to start by publishing the packaged image to a Docker repository. For simplicity, we are using a public DockerHub repository, however, for actual development projects you would want to push your Docker images to a private repository. Let's create a new *Free Style Project* job in Jenkins by clicking the *New Item* button and name our job *push-go-auth-image*. Once you do so, you will be taken to the Jenkins job configuration page where you can define the steps required to push your go-auth image up to Dockerhub.

Since this is a continuation of the pipeline we built in our previous chapter, the job will have similar configuration to *go-auth-integration-test* job. The first setting you need is to make it *parameterized build* and adding the GO_AUTH_VERSION parameter.



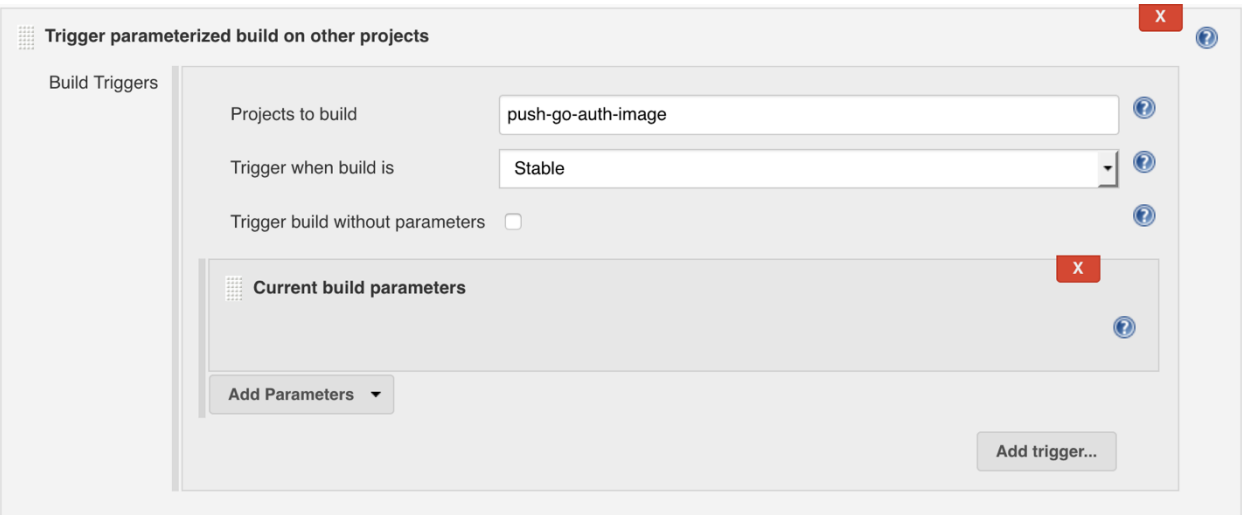
In order to actually push the image, we will select the *Add build step* drop down and then the *Execute shell* option. In the resulting text box add the commands shown below. In the commands, we are going to log in to DockerHub and push the image we built earlier. We're pushing to the *usman/go-auth* repository, however, you will need to push to your own DockerHub repository.

As covered in the previous chapter, we're using git-flow branching model where all feature branches are merged into the 'develop' branch. To continuously deploy changes to our integration environment we need a simple mechanism to generate the latest image based off of develop. In our package job we tagged the docker container using the GO_AUTH_VERSION (e.g., docker build -t usman/go-auth:${GO_AUTH_VERSION} ....). By default, the version will be develop, however, later in this chapter we'll create new releases for our application and use the CI/CD pipeline to build, package, test and deploy them to our integration environment. Note that with this scheme, we're always overwriting the image for our develop branch (usman/go-auth:develop) which prevents us from referencing historical builds and do rollbacks. One simple change that you can make to the pipeline is to attach the Jenkins build number to the version itself, e.g., usman/go-auth:develop-14.

Note that you will need to specify your DockerHub username, password and email. You can either use a parameterized build to specify these for each run or use the Jenkins Mask Passwords Plugin to define these securely, once in the main Jenkins configuration and inject them into the build. Make sure to enable 'Mask passwords (and enable global passwords)' under *Build Environment* for your job.

```
echo ${GO_AUTH_VERSION}
docker login -u ${DOCKERHUB_USERNAME} -p ${DOCKERHUB_PASSWORD} -e ${DOCKERHUB_EMAIL}
docker push usman/go-auth:${GO_AUTH_VERSION}
```

Now we need to make sure that this job is triggered after our integration test job. To do that we need to update our integration test job to trigger *parameterized build* with *current build parameters*. This means that after each successful run of the integration test job we will push the tested image up to DockerHub.



Lastly, we need to trigger the deployment job once the image is successfully pushed to DockerHub. Again, we can do that by adding a post-build action as we did for other jobs.

2.2.2 Deploying to Integration environment

For this we will use the Rancher Compose CLI to; stop the running environment, pull latest images from DockerHub, and restart the environment (A brief word of caution, the Updates API is under heavy development and may change. There will certainly be newer features added in the coming weeks and months so check the Documentation to see if there are updated options). Before we create a Jenkins job to achieve continuous deployment let's first go through the steps manually.

A simple approach would be to stop all services, (auth service, load balancer and mysql), pull the latest images and start all services. This however would be less than ideal for long running environments where we only want to update the application. To update our application, we're first going to stop auth-service. You can do this by using the stop command with Rancher Compose.

```
# If you not have already done so
# git clone https://github.com/usmanismail/go-messenger.git
# cd go-messenger/deploy
rancher --debug stop –type service Goauth
```

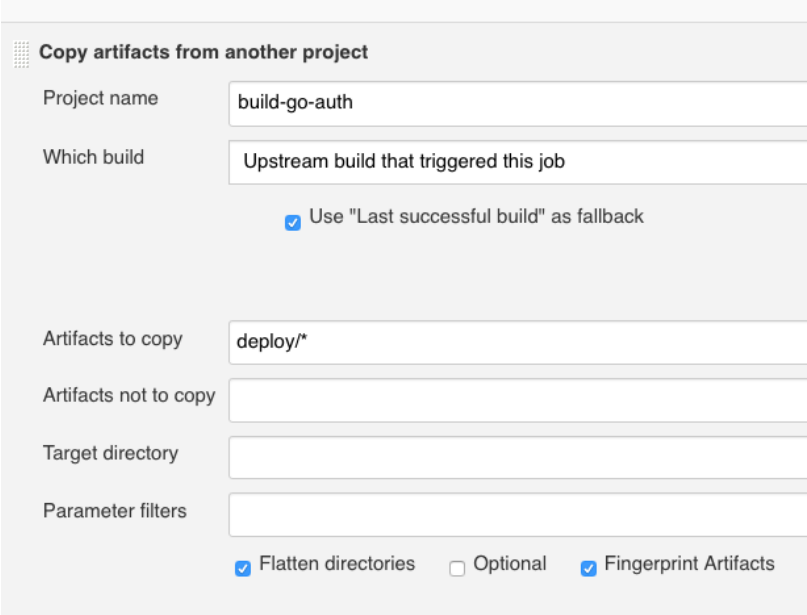
This will stop all containers running for goauth service which you can verify by opening the stack in the Rancher UI and verifying that the status of the service is set to *Inactive*. Next, we'll tell Rancher to pull the image version we want to deploy. Now we can dynamically specify the version we want to run without updating the template every time. If you push the same image version multiple times then add the *pull* switch to make sure we are using the latest copy of the image version.

```
auth_version=develop-14 rancher --debug up --pull –d -s go-messenger-int Goauth
```

We can also use the upgrade function to do rolling updates of our environment without stopping using the following command. We will be taking a more about rolling updates in the next section. Once the upgrade is complete you can use the --rollback command or the --confirm-upgrade to confirm the change or rollback to the preview state.

```
auth_version=develop-14 ./rancher --debug up -d --upgrade --pull -s go-messenger-int Goauth
```

Now that we know how to run our update lets create a Jenkins job in our pipeline to do so. As before create a new freestyle project and name it *deploy-integration*. As with all other jobs, this will also be a parameterized build with GO_AUTH_VERSION as a string parameter. Next, we need to copy over artifacts from the upstream *build-go-auth* job.



Lastly, we need to add the *Execute Shell* build step with the Rancher Compose up command that we specified earlier. Note that you will also need to setup Rancher CLI on Jenkins ahead of time, and make it available to your build on the system path. The contents of the execute shell step looks like the snippet shown below. Note that if you have multiple Rancher Compose nodes the load balancer containers may launch on different hosts and your Route 53 record-set may need to be updated.

```
auth_version=${GO_AUTH_VERSION} rancher --debug up -d --upgrade --pull \
   -s go-messenger-int Goauth

# Optionally Run Validation against environment
if [[ "$status_first" -ne 200 ]]; then
       echo "Fail: Expecting 200 OK for first user register got $status_first"
       # Or rollback if verification failed
       auth_version=${GO_AUTH_VERSION} rancher --debug up -d \
           --rollback -s go-messenger-int Goauth
       quit 1
else
       echo "Pass: Register User"
       # Confirm upgrade
       auth_version=${GO_AUTH_VERSION} rancher --debug up -d \
           --confirm-upgrade -s go-messenger-int Goauth
fi
```

With our two new Jenkins jobs the pipeline we started in the previous chapter now looks like the image shown below. Every check-in to our sample application now gets compiled to make sure there are no syntax errors and that the automated tests pass. That change then gets packaged, and tested with integration tests and finally deployed for manual testing. The five steps below provide a good baseline template for any build pipeline and helps predictably move code from development to testing and deployment stages. Having a continuous deployment pipeline ensures that all code is not only tested by automated systems but is available for human testers quickly. It also serves as a model for production deployment automation and can test the operations tooling and code to deploy your application on a continual basis.

2.2.3 Releasing and deploying a new version

Once we have deployed our code to a persistent testable environment we will let the QA (Quality Assurance) team test the changes for a period of time. Once they certify that the code is ready, we can create a release which will subsequently be deployed to production. The way releases work with git-flow is similar to how feature branches (which we talked about in the previous chapter work. We start a release using the *git flow release start [Release Name]* command (shown below). This will create a new named release branch. In this branch, we will perform housekeeping actions such as incrementing version numbers and making any last-minute changes.

```
git flow release start v1
Switched to a new branch 'release/v1'
Summary of actions:
- A new branch 'release/v1' was created, based on 'develop'
- You are now on branch 'release/v1'
Follow-up actions:
- Bump the version number now!
- Start committing last-minute fixes in preparing your release
- When done, run:
git flow release finish 'v1'
```

Once done, we can run the *release finish* command to merge the release branch into the master branch. This way master always reflects the latest released code. Further, each release is tagged so that we have a historical record of what went into each release. Since we don't want any other changes to go in, let's finalize the release.

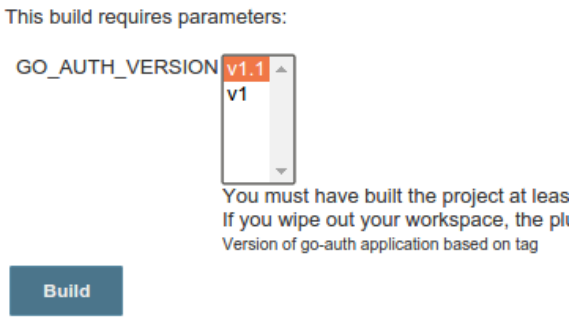
```
Switched to branch 'master'
Merge made by the 'recursive' strategy.
 README.md | 1 +
 1 file changed, 1 insertion(+)
Deleted branch release/v1 (was 7ae8ca4).
Summary of actions:
- Latest objects have been fetched from 'origin'
- Release branch has been merged into 'master'
- The release was tagged 'v1'
- Release branch has been back-merged into 'develop'
- Release branch 'release/v1' has been deleted
```

The last step here is to push the release to remote repository.

```
git push origin master
git push --tags //pushes the v1 tag to remote repository
```

If you're using Github to host your git repository, you should now have a new release. It is also a good idea to push images to DockerHub with a version that matches the release name. To do so, let's trigger our CD pipeline by running the first job. If you

recall, we setup Git Parameter plugin for our CI pipeline to fetch all the tags matching our filter from git. This normally defaults to *develop* however, when we trigger the pipeline manually we can choose from git tags. For example, in the section below, we have two releases for our application. Let's select one of them and kick off the integration and deployment pipeline.



This will go through the following steps and deploy our application with version 1.1 to our long running integration environment all with a couple of clicks:

1. Fetch the selected release from git
2. Build the application and run unit tests
3. Create a new image with tag v1.1 (e.g., usman/go-auth:v1.8)
4. Run integration tests
5. Push the image (usman/go-auth:v1.1) to DockerHub
6. Deploy this version to our integration environment

2.3 Deployment Strategies

One of challenges when managing long running environments is to ensure minimal or zero downtime during releases. Doing so predictably and safely, takes quite a bit of work. Automation and quality assurance can go a long way to make releases more predictable and safe. Even then, failures can and do happen, and for any good ops team the goal would be to recover quickly while minimizing impact. In this section, we'll cover a few strategies for deploying long running environments and their trade-offs.

2.3.1 In-place updates

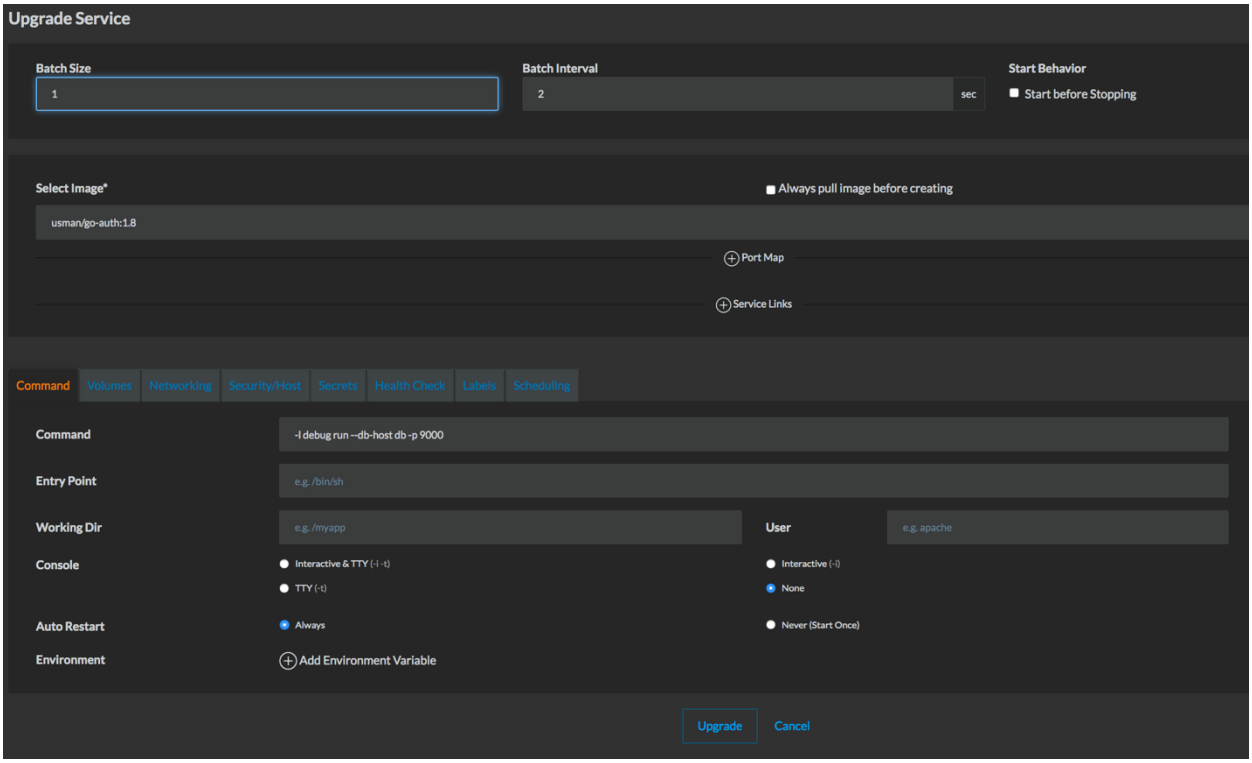
The first strategy is called *in-place update*, as the name suggests, the idea is to reuse the application environment and update the application in-place. These are also sometimes referred to as *Rolling Deployments*. We're going to work with our sample application (go-auth) we covered so far. Further, we're going to assume that you have the service running with Rancher. To do an in-place update, you can use the upgrade command:

```
auth_version=develop-14 rancher --debug up –batch-size 1 --pull –d -s go-messenger-int
Goauth
```

Behind the scenes, Rancher agent fetches the new image on each host running an auth-service container (because of the ---pull the image is downloaded again even if it exists). The agent then stops the old containers and launches new containers in batches. You can control the size of the batch by using the *--batch-size* flag. Additionally, you can specify a pause interval *(--interval)* between batch updates. A large enough interval can be used to allow you to verify that the new containers are behaving as expected, and on the whole, the service is healthy. By default, old containers are terminated and new ones are launched in their place. Alternatively, you can tell rancher to start the new containers before stopping the old containers by setting the *start_first* flag in your rancher-compose.yml.

```
auth-service:
  upgrade_strategy:
    start_first: true
```

If you are not happy with the update and want to rollback, you can do so with the rollback flag for the upgrade command. Alternatively, if you want to proceed with the update, simply tell rancher to complete the update by specifying the confirm-update flag. You can also do these in one step by specifying the confirm-update flag in the original up command. You can also perform these updates using the rancher UI, by selecting "upgrade" from a service's menu (shown below).



In-place updates are quite simple to perform and don't require the additional investment to manage multiple stacks. There are, however, downsides to this approach for production environments. First, it is typically difficult to have fine-grained control over rolling updates, i.e., they tend to be unpredictable under failure scenarios. For example, dealing with partial failures and rolling back a rolling update can get quite messy. You'll need to know which nodes to which changes were deployed, which failed to deploy and which are still running the previous revision. Second, you need to make sure all updates are not only backwards compatible, but also forward compatible as you will have old and new versions of your application running concurrently in the same environment. Last, depending on the use case, in-place updates might not be practical. For example, if legacy clients need to continue to use the old environment while newer clients roll forward. In this case ,separating client requests is much easier with some of the other approaches we are going to list today.

2.3.2 Blue-Green Deployments

Lack of predictability is a common problem with in-place updates. To overcome that, another strategy for deployments is to work with two parallel stacks for an application: one active and the other in standby. To run a new release, the latest version of the

application is deployed to the standby stack. Once the new version is verified to be working, a cut-over is done to switch traffic from the active stack to the standby stack. At that point the previously active stack becomes the standby and vice versa. This strategy allows for verification of deployed code, fast rollbacks (switching standby vs active again) and also extended concurrent operation of both stacks if needed. This strategy is commonly referred to as blue-green deployments. To accomplish such deployments through Rancher for our sample application, we can simply create two stacks in Rancher: go-auth-blue and go-auth-green. Further, we're going to assume that the database is not part of these stacks and is being managed independently. Each of these stacks would just run *goauth service* and *auth-lb* services. Assuming that go-auth-green stack is live, to perform an update, all we need to do is to deploy the latest version to the blue stack, perform validation and switch the traffic over to it.

2.3.2.1 Traffic Switch

There are a couple of options for doing the traffic switch, changing DNS records to point to the new stack or using a proxy or load-balancer and routing traffic to the active stack. We cover both options in detail below.

DNS record update

A simple approach is to update a DNS record to point to the active stack. One advantage of this approach is that we can use weighted DNS records to slowly transition the traffic over to the new version. This is also a simple way to do canary releases which are quite useful for safely phasing in new updates on live environments or for doing A/B tests. For example, we can deploy an experimental feature to its own feature stack (or to the in-active stack) and then update the DNS to forward only a small fraction of the traffic to the new version. If there is an issue with the new update, we can reverse the DNS record changes to roll back. Further, it is much safer than doing a cut-over where all traffic switches from one stack to another, which can potentially overwhelm the new stack. Although simple, DNS record update is not the cleanest approach if you want all your traffic to switch over to the new version at once. Depending on the DNS clients, the changes can take a long time to propagate, resulting in a long tail of traffic against your old version instead of a clean switch over to the new version.

Using a reverse proxy

Using a proxy or load-balancer and simply updating it to point to the new stack is a cleaner way of switching over the entire traffic at-once. This approach can be quite useful in various scenarios, e.g., non-backwards compatible updates. To do this with Rancher, we first need to first create another stack which contains a load-balancer only.

Next, we specify a port for the load-balancer, configure SSL and pick the load-balancer for the active stack as the *target service* from the drop-down menu to create the load-balancer. Essentially, we are load balancing to a load-balancer which is then routing traffic to actual service nodes. With the external load-balancer, you don't need to update the DNS records for each release. Instead, you can simply update the external load-balancer to point to the updated stack.

2.4 Summary

In this chapter, we covered creating a continuous deployment pipeline which can put our sample application on an integration environment. We also looked at integrating DNS and HTTPS support to create a more secure and usable environment with which clients can integrate. In the subsequent work, we'll look at running production environments. Deploying to production environments presents its own set of challenges as we will be expected to deploy under-load, with little (ideally zero) downtime. Furthermore, production environments present challenges as they have to scale out to meet load while also scaling back to control cost. Lastly, we take a more comprehensive look at DNS management in order to provide automatic failover and high availability. We'll also look at operations management of Docker environments in production as well as different types of workloads for example state-full connected services.

Conclusion

This document is necessarily a limited look at a few approaches for implementing a complete turnkey CI/CD pipeline using containers. We've tried to cover common use cases, provide detailed examples and share some of the best practices we've learned from years of working in DevOps at web services companies. We hope to follow-up this e-book with a companion volume that will look in more depth at running services in production using containers. As always, we will be posting our latest work on the Rancher blog, and we welcome any feedback you may have about this paper to info@rancher.com.