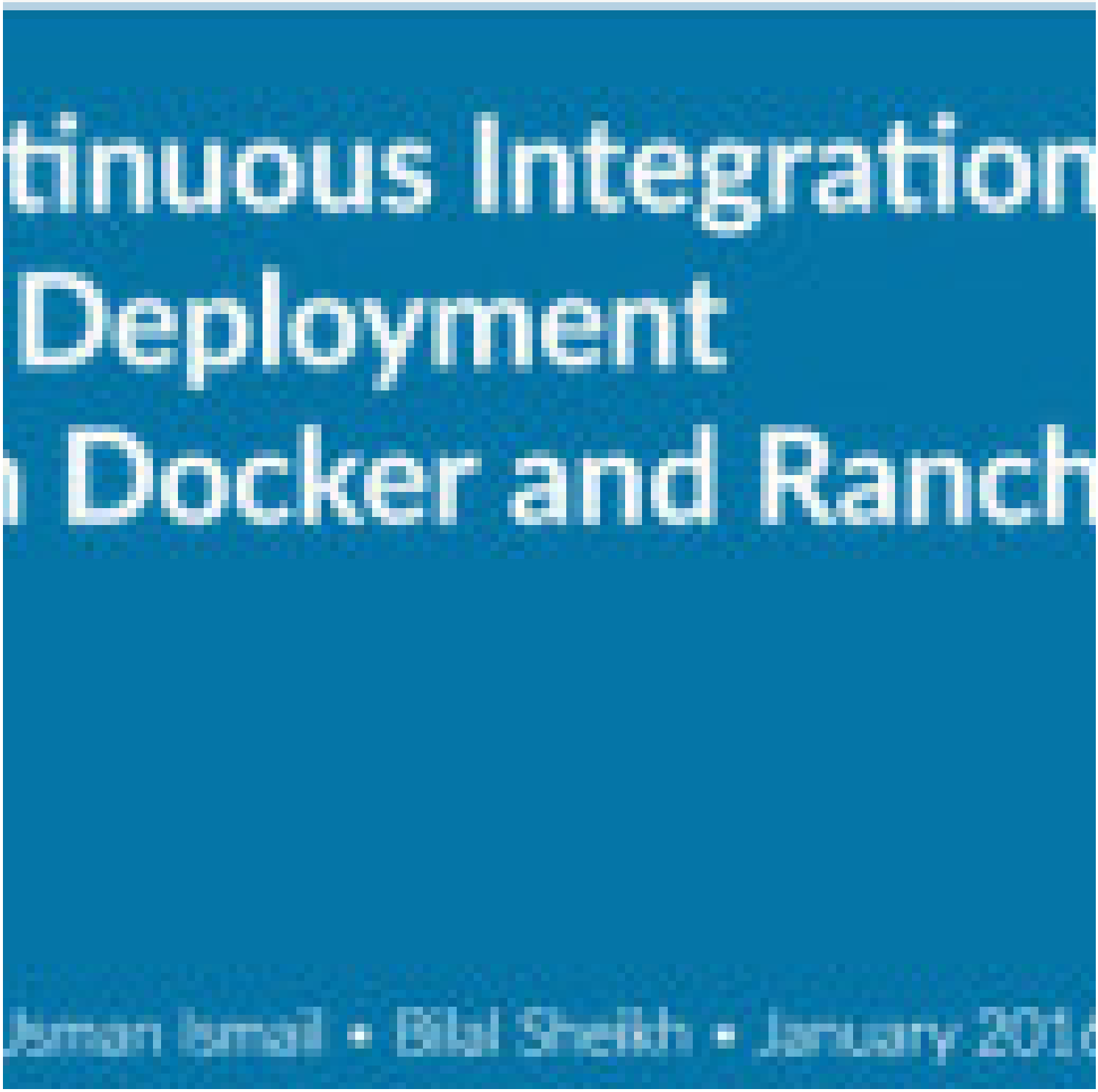
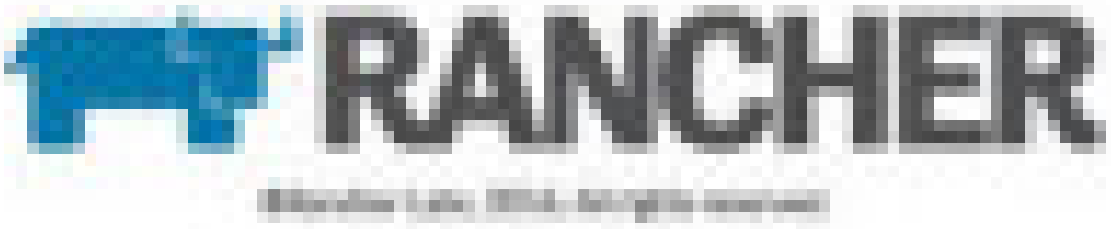
Docker-based build pipelines (Part 1) - Continuous Integration and Testing

JANUARY 6, 2016





ation stacks running on top of docker, e.g. Magento, Jenkins, Prometheusand so forth. However, containerized deployment can be useful for more than just defining applicati

Docker and Rancher in its' various stages. Specifically, we're going to cover; building code, running tests, packaging artifacts, continuous integration and deployment, as well

g 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 nsistent and stable build for all developers while ensuring code quality becomes a much bigger challenge. In this post we will cover some of the challenges around CI and te

s

lenges 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 inte ne 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.

ent dependencies. This includes IDE and IDE configurations, tools versions (e.g. maven version, python version) and configuration e.g. static analysis rule files, code formatti e 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 fab ric required python2.7 where as s3cmd required python2.6. A fix required us to either switch to a beta version of s3cmd or an older version of fabric.

mes. As projects grow in scope and complexity, more and more languages get added (my current project uses Java, Groovy, python and Protocol Buffers IDL). Tests get add e 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 cleanup after themselves when the ls to a dangerous practice of skipping test runs.

ort the following requirements (among others):

milar (or identical) build environments with the same dependencies on different developer machines and automated build serv

nvironment for all developers and build servers from a central code repository or server. This includes setting up the build envir

oject must be build in isolation other than well defined shared dependencies.

for sub-components.

ized dependency management. Most modern languages and development frameworks have support for automated dependency management. Maven is used extensively in J index file (pom.xml, requirements.txt or gemfile) is committed to your control. The tool can then be run to consume the file and download dependencies onto the build mach control. However, there remains the issue of managing environmental dependencies. For example the correct version of maven, python and ruby have to be installed. We als undler we must wrap our build commands in scripts which trigger a dependency update run.

s most small teams just use documentation and leave the onus on developers. This however, does not scale to large teams specially if the dependencies are updated over ti 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 all st 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 nc 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

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 also a en 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 yo his 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 c

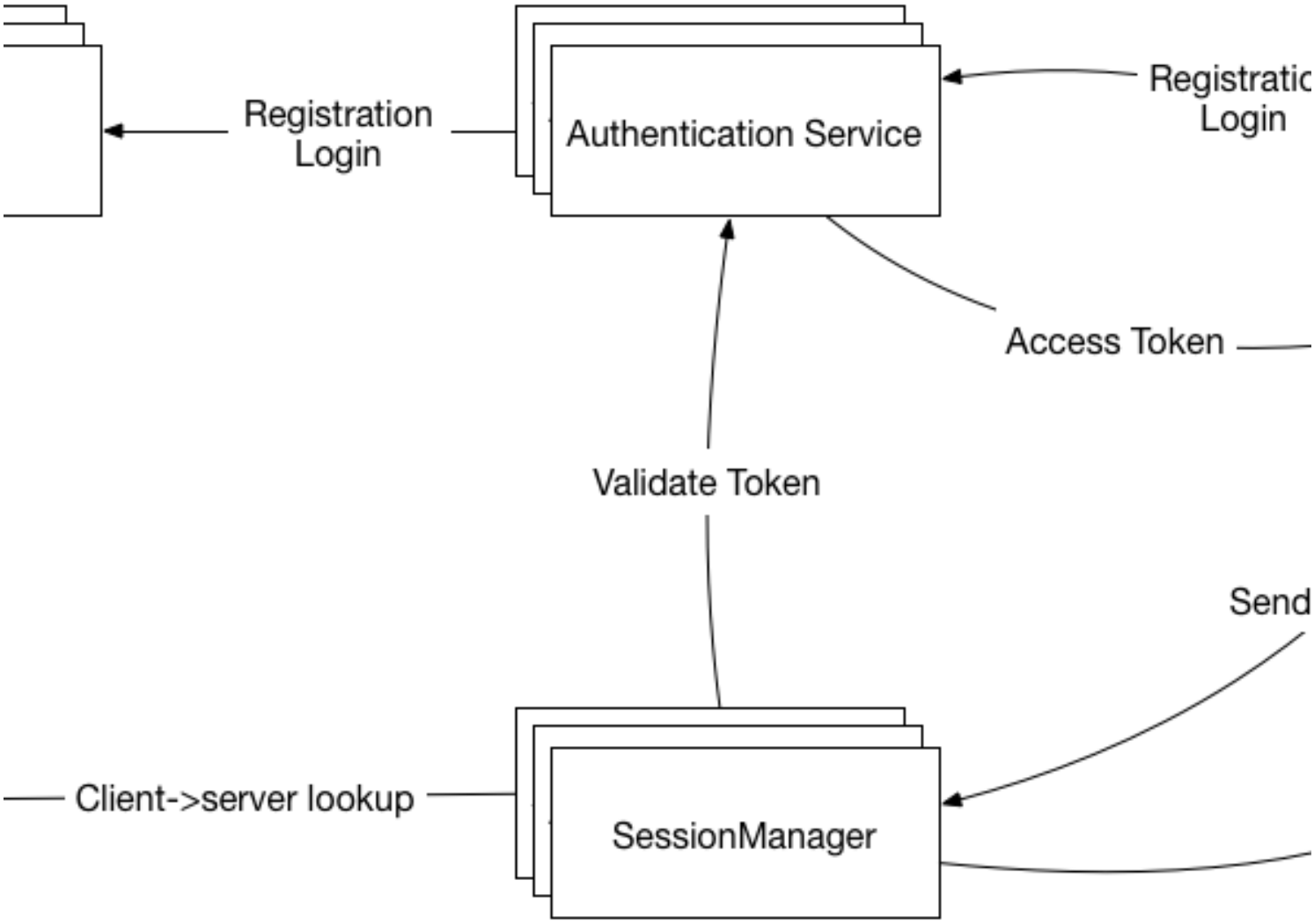
ement (maven, pip, bundler), orchestration (puppet, chef) and virtualization (vagrant), we *can* build a stable, testable centrally managed build system. Not all projects warran

s

ements 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 con

nments

(and subsequent articles) 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 s
and a session manager which accepts long running TCP connections from clients and routes messages between clients. For the purposes of this article, we will be concentr
a database cluster to store user information.



nent

ainer 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 u
for your project, a similar \"build container\" can be created with Java base image and installation of Maven instead of godep.

to build and test our code in one place. The script shown below downloads dependencies using godep restore, standardizes formatting using the go fmt command, runs tes

be

ools required to build a component into a single, versioned container image. This image can be downloaded from Dockerhubor built from Dockerfile. Now all developers (and

```
[PROJECT]/[SUB-CDIRECTORY]/ \
?ROJECT]/[SUB-CDIRECTORY]/ \
```

nage version 1.4 and mounting our source code into the container using the -v switch and specifying the SOURCE_PATH environment variable using the -e switch. In order to
file called go-auth in the root directory of the go-auth project.

```
ssenger.git
```

```
/go-messenger/go-auth/ \
go-messenger/go-auth/ \
```

s is that we can easily swap out build tools and configuration. For example in the commands above we have been using golang 1.4. By changing **go-builder:1.4** to **go-builde**
ge 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
nt containers to build them without worrying about managing multiple language versions in a single build environment. For example, our earlier python problem could be miti

Docker

s 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 con
ur 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

```
un","-p","9000"]
```

ng build environments

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 docke
es. 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
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 startir

se 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 Go/

lication 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 tim on environment.

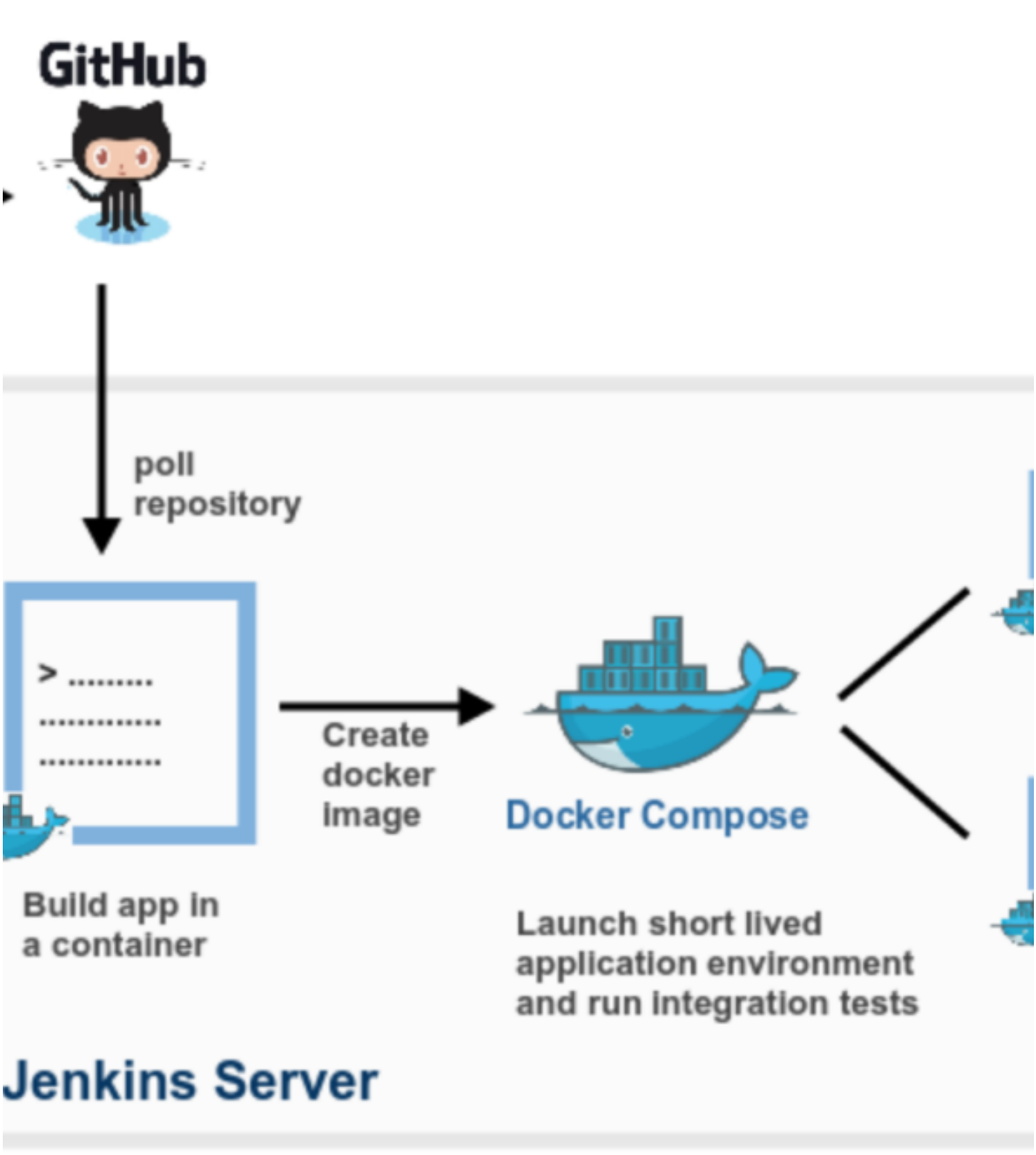
```
-d password=PASSWORD ${service_ip}:9000/user
```

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 lication 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 lik e maven project.]{.pl-s}

```
-d password=PASSWORD ${service_ip}:9000/user

-d password=PASSWORD ${service_ip}:9001/user

-d password=PASSWORD ${service_ip}:9001/user

-d password=PASSWORD ${service_ip}:9000/user
```

Pipeline with Docker and Jenkins

egration 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. Ho t how to partition out code into branches.



ant 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 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 wa ack 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

l 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 i

t 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 show
low command, it will create a develop branch (if it didn't exist) and check it out as the working branch.

```
orth production releases?

er]
: [develop]
s?
```

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 tick
: will automatically switch to the feature branch.

```
ed, based on 'develop'

n done, use:
```

l 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
ure by typing \"git flow feature finish MSP-123\".

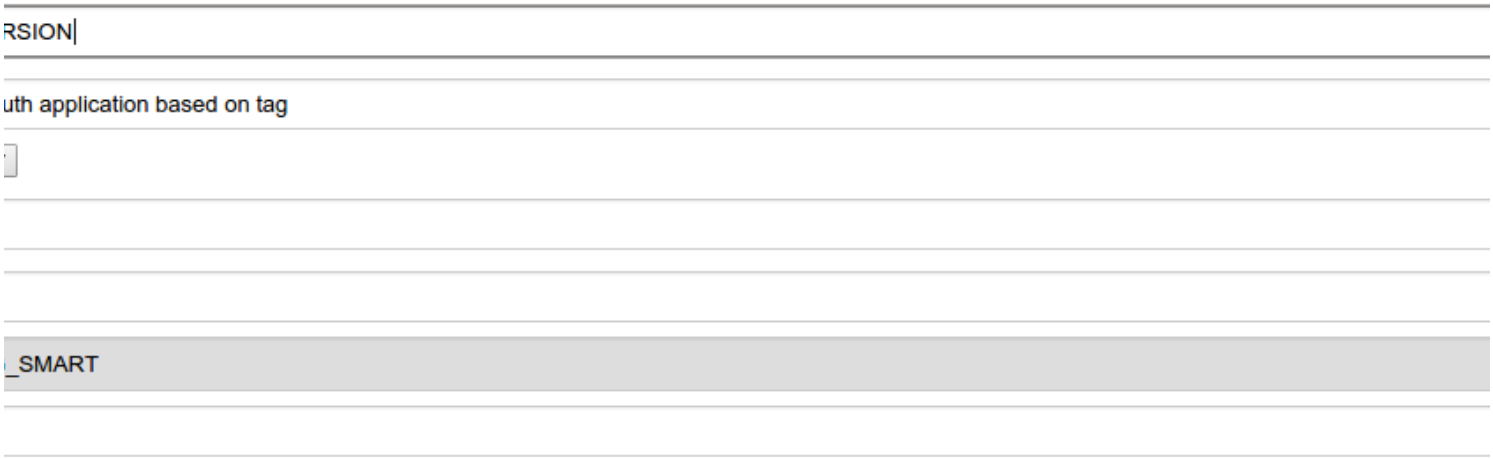
```
4).

 merged into 'develop'
n removed
```

eature 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 cor
rnative 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

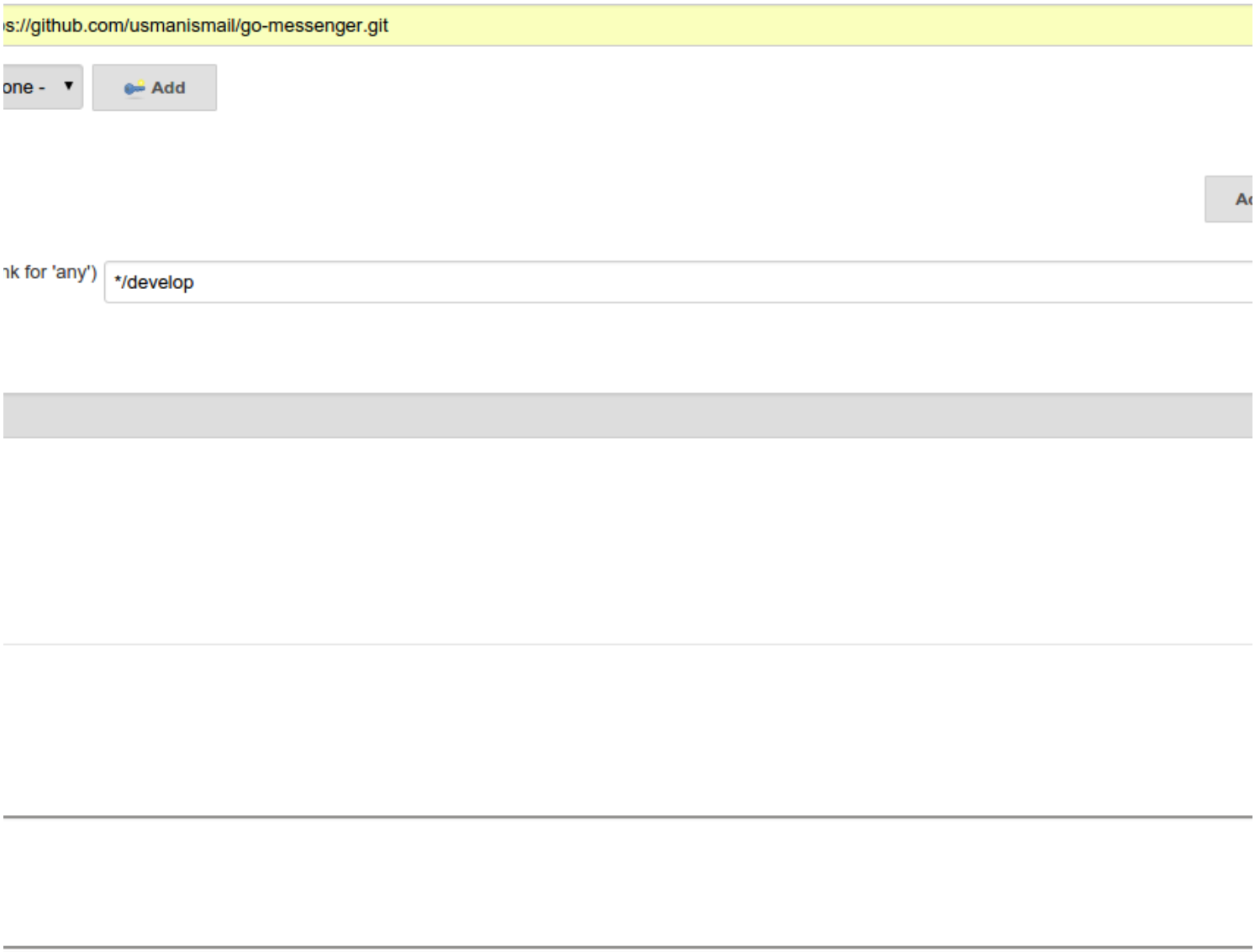
ster up and running. If not, you can read more about setting up a scalable Jenkins cluster in our earlier post. Once you have Jenkins running, we need the following plugins a

st 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.

source control after each commit and ensure that it compiles. It will also run units tests. To setup the first job for our sample project select *New Item > Freestyle Project*. S
configure the parameter to pick up any tags matching \"v*\" (e.g., v2.0) and default to develop (branch) if no value is specified for the parameter. This is quite useful for getti
lue is specified, the value of GO_AUTH_VERSION defaults to develop.



ub.com/usmanismail/go-messenger.git as the *repository url*,* *specify the branch as */develop and set a poll interval, e.g., 5 minutes. With this, Jenkins will keep tracking ou



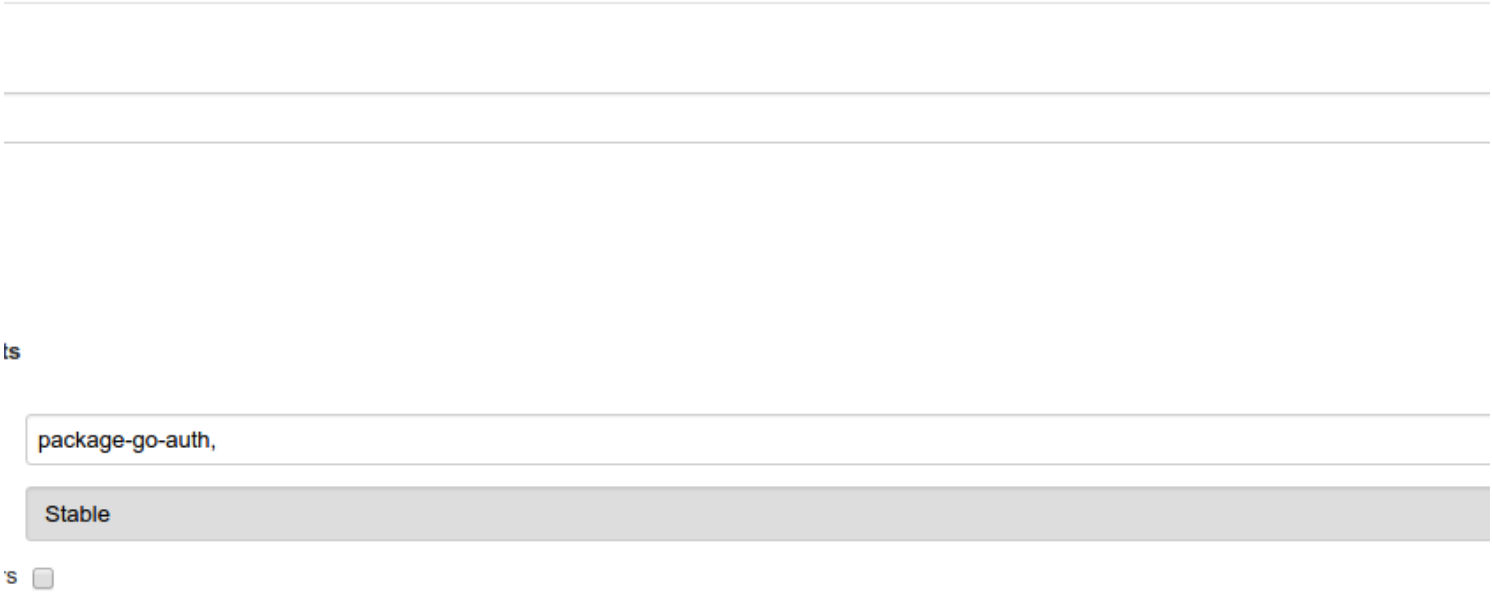
Sunday, November 15, 2015 6:20:39 AM UTC; would next run at Sunday, November 15, 2015 6:25:39 AM UTC.

// and copy the docker run command from earlier in the article. This will get the latest code from Github and build the code into the go-auth executable.

```
.thub.com/usmanismail/go-messenger/go-auth/ \
:hub.com/usmanismail/go-messenger/go-auth/ \
..4
```

ariables

rchive the Artifacts to archive the go-auth binary that we build in this job and *Trigger parameterized builds* to kick off the next job in the pipeline as shown below. When addi
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 s



llowing. You can see that we use a dockerized container to run the build. The build will use go fmt to fix an formatting inconsistencies in our code and also run our unit tests
cations 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.

```
ild-go-auth/workspace
imeout=10
itory
thub.com/usmanismail/go-messenger.git # timeout=10
nub.com/usmanismail/go-messenger.git

-progress https://github.com/usmanismail/go-messenger.git +refs/heads/*:refs/remotes/origin/*
op^{commit} # timeout=10
n/develop^{commit} # timeout=10
5b7429bca9bcda994131 (refs/remotes/origin/develop)
t=10
429bca9bcda994131
56e208773361b4 # timeout=10
00899558419690.sh


ld-go-auth/workspace/go-auth:/go/src/github.com/usmanismail/go-messenger/go-auth/ -e SOURCE_PATH=github.com/usmanismail/


r/go-auth [no test files]
r/go-auth/app [no test files]
r/go-auth/database    [no test files]
r/go-auth/logger  [no test files]
r/go-auth/user    0.328s
```

ess control for builds, so falling back to legacy behavior of permitting any downstream builds to be triggered

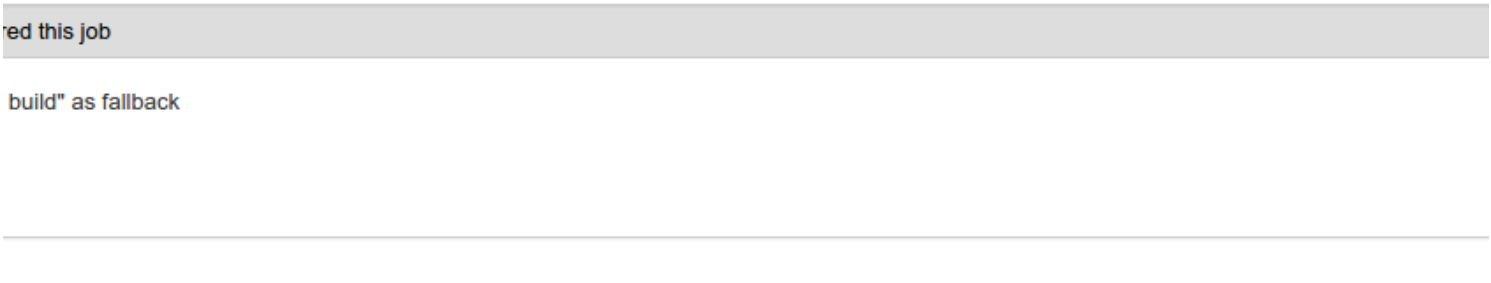
ker container. To create the package job select, *New Item* > *Freestyle Project* and give your second job a name matching what you specified in the previous job. As before, th
quent jobs, the GO_AUTH_VERSION is simply a string parameter with a default value of \"develop\". The expectation here is that this value would be coming from the upstre

eview

and add a build step to execute shell.

RSION} .

xecutable we built in the previous step. To do this we add a build step to copy artifacts from the upstream build. This will make sure that we have the executable available fo
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 ne



uild 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 temp

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
like 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

mpose to bring up our environment and 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. Con 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-a

```
DEBUG [0m Connecting to database db:3306
DEBUG [0m Unable to connec to to database: dial tcp 10.0.0.28:3306: connection refused. Retrying...
DEBUG [0m Connecting to database db:3306
DEBUG [0m Unable to connec to to database: dial tcp 10.0.0.28:3306: connection refused. Retrying...
DEBUG [0m Connecting to database db:3306
G [0m Connected to DB db:3306/messenger
 Created User Table
n Created Token Table
DEBUG [0m Connected to database
istening on port 9000
```

ine 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 j ogressing through your build and deployment pipeline.

at 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 paran the develop branch. You can also just click 'Run' in the pipeline view, however, at the time of writing, there is an open bug in Jenkins which prevents it from starting the pipel ation with the following steps:

m into develop

application in a containerized environment

r compose

nts

nerged 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 deepe
rsions to various deployment environments as they clear testing phases.

ntinuous integration pipeline for our project which is centrally managed, testable, and repeatable across machines and in time. We were able to isolate the environmental de
ipeline which we'll continue to build and document in a series of write ups. The next step in our pipeline is to setup continuous deployment. Next week we will show how to u
running testing environment and deployment pipeline for large scale projects.

ta. You can also download our free eBook \"Continuous Integration and Delivery with Docker and Rancher\" which covers all aspects of building a docker-based developmen

xperience in building large scale distributed services on top of various cloud platforms. You can read more of their work at techtraits.com, or follow them on twitter @usmar

an expert through our classes on Kubernetes and Rancher