**How To Deploy a Go Web Application with Docker**

**Creating a Simple Go Web Application**

We’ll create a simple web application in Go for demonstration in this article. This application, which we’ll call *MathApp*, will:

* Expose routes for different mathematical operations,
* Use HTML templates for views,
* Use a configuration file to customize the application, and
* Include tests for selected functions.

Visiting /sum/3/6 will show a page with the result of adding 3 and 6. Likewise, visiting /product/3/6 will show a page with the product of 3 and 6.

**Final Directory Structure**

Upon completion, the directory structure of MathApp will look like:

MathApp

├── Dockerfile

├── Dockerfile.production

└── src

├── conf

│ └── app.conf

├── go.mod

├── go.src

├── main.go

├── main\_test.go

├── vendor

└── views

├── invalid-route.html

└── result.html

The main application file is main.go, located at the src directory. This file contains all the functionality of the app. Some of the functionality from main.go is tested using main\_test.go.

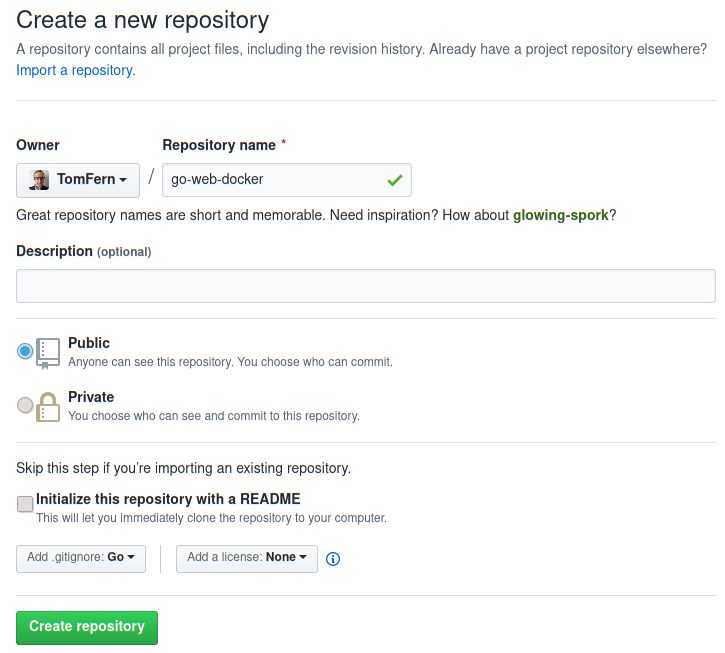
The views folder contains the view files invalid-route.html and result.html. The configuration file app.conf is placed in the conf folder. Beego uses this file to customize the application.

**Create the GitHub Repository**

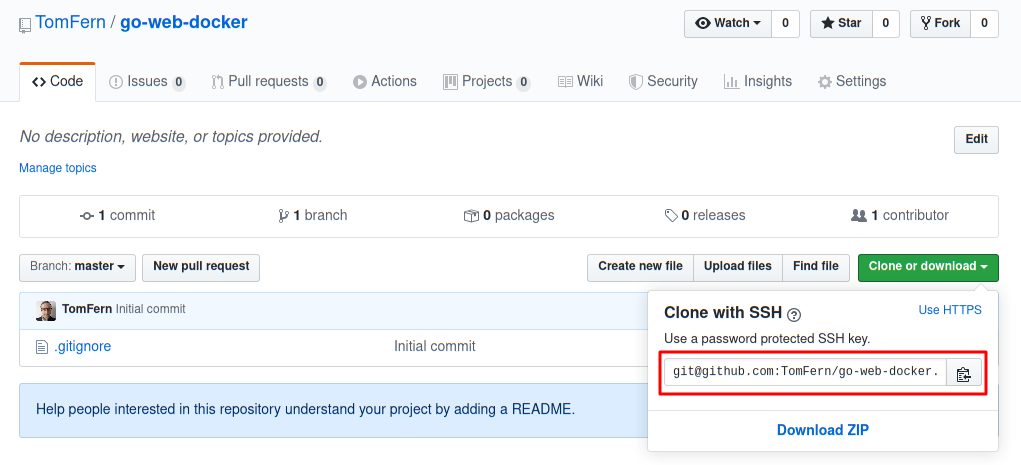
We’ll use [Go mod](https://blog.golang.org/using-go-modules), the official module manager, to handle Go modules in a portable way without having to worry about GOPATH.

We’ll start by creating a GitHub repository:

* Go to [GitHub](https://github.com/) and login or sign up.
* Create a [new repository](https://github.com/new).
* Select **Go**as the language:



* Use the **Clone or download** button to get your repo URL:



* Clone the repository to your machine:

$ git clone YOUR\_REPOSITORY\_URL

We can use the repository name to initialize the project:

$ mkdir src

$ cd src

$ export GOFLAGS=-mod=vendor

$ export GO111MODULE=on

$ go mod init github.com/YOUR\_GITHUB\_USER/YOUR\_REPOSITORY\_NAME

# (example: go mod init github.com/tomfern/go-web-docker)

From now on, we can use these commands:

$ go mod download

$ go mod vendor

$ go mod verify

To download the required dependencies in the vendor/ directory, this is much easier than downloading the modules one by one manually. It also will make our lives easier later when we set up Continuous Integration.

**Application File Contents**

Before continuing, let’s create the file structure:

$ mkdir conf views

The main application file (main.go) contains all the application logic. The contents of this file are as follows:

// main.go

package main

import (

"strconv"

"github.com/astaxie/beego"

)

func main() {

/\* This would match routes like the following:

/sum/3/5

/product/6/23

...

\*/

beego.Router("/:operation/:num1:int/:num2:int", &mainController{})

beego.Run()

}

type mainController struct {

beego.Controller

}

func (c \*mainController) Get() {

//Obtain the values of the route parameters defined in the route above

operation := c.Ctx.Input.Param(":operation")

num1, \_ := strconv.Atoi(c.Ctx.Input.Param(":num1"))

num2, \_ := strconv.Atoi(c.Ctx.Input.Param(":num2"))

//Set the values for use in the template

c.Data["operation"] = operation

c.Data["num1"] = num1

c.Data["num2"] = num2

c.TplName = "result.html"

// Perform the calculation depending on the 'operation' route parameter

switch operation {

case "sum":

c.Data["result"] = add(num1, num2)

case "product":

c.Data["result"] = multiply(num1, num2)

default:

c.TplName = "invalid-route.html"

}

}

func add(n1, n2 int) int {

return n1 + n2

}

func multiply(n1, n2 int) int {

return n1 \* n2

}

In your application, this might be split across several files. However, for the purpose of this tutorial, I like to have everything in one place.

**Test File Contents**

The main.go file has some functions which need to be tested. The tests for these functions can be found in main\_test.go. The contents of this file are as follows:

// main\_test.go

package main

import "testing"

func TestSum(t \*testing.T) {

if add(2, 5) != 7 {

t.Fail()

}

if add(2, 100) != 102 {

t.Fail()

}

if add(222, 100) != 322 {

t.Fail()

}

}

func TestProduct(t \*testing.T) {

if multiply(2, 5) != 10 {

t.Fail()

}

if multiply(2, 100) != 200 {

t.Fail()

}

if multiply(222, 3) != 666 {

t.Fail()

}

}

Testing your application is particularly useful if you want to do [Continuous Deployment](https://semaphoreci.com/cicd). If you have adequate testing in place, then you can make stress-free deployments anytime, any day of the week.

**View Files Contents**

The view files are HTML templates; these are used by the application to display the response to a request. The content of views/result.html is as follows:

<!-- views/result.html -->

<!doctype html>

<html>

<head>

<title>MathApp - {{.operation}}</title>

</head>

<body>

The {{.operation}} of {{.num1}} and {{.num2}} is {{.result}}

</body>

</html>

The content of views/invalid-route.html is as follows:

<!-- invalid-route.html -->

<!doctype html>

<html>

<head>

<title>MathApp</title>

<meta name="viewport" content="width=device-width, initial-scale=1">

<meta charset="UTF-8">

</head>

<body>

Invalid operation

</body>

</html>

**Configuration File Contents**

The conf/app.conf file is read by Beego to configure the application. Its content is as follows:

appname = mathapp

runmode = "dev"

httpport = 8010

In this file:

* **appname**: is the name of the process that the application will run as,
* **httpport**: is the port on which the application will be served, and
* **runmode**: specifies which mode the application should run in. Valid values include dev for development and prod for production.

Finally, install the Go modules with:

$ go mod download

$ go mod vendor

$ go mod verify

**Using Docker During Development**

This section will explain the benefits of using Docker during development, and walk you through the steps required to use Docker in development.

**Configuring Docker for Development**

We’ll use a Dockerfile to configure Docker for development. The setup should satisfy the following requirements for the development environment:

* We will use the application mentioned in the previous section,
* The files should be accessible both from inside and outside of the container,
* We will use the bee tool, this will be used to live-reload the app (inside the Docker container) during development,
* Docker will expose the application on port 8010,
* In the Docker container, the application is located at /home/app,
* The name of the Docker image we’ll create for development will be mathapp.

**Step 1 – Creating the Dockerfile**

Go back to the top level of your project:

$ cd ..

The following Dockerfile should satisfy the above requirements.

FROM golang:1.15.7-buster

RUN go get -u github.com/beego/bee

ENV GO111MODULE=on

ENV GOFLAGS=-mod=vendor

ENV APP\_USER app

ENV APP\_HOME /go/src/mathapp

ARG GROUP\_ID

ARG USER\_ID

RUN groupadd --gid $GROUP\_ID app && useradd -m -l --uid $USER\_ID --gid $GROUP\_ID $APP\_USER

RUN mkdir -p $APP\_HOME && chown -R $APP\_USER:$APP\_USER $APP\_HOME

USER $APP\_USER

WORKDIR $APP\_HOME

EXPOSE 8010

CMD ["bee", "run"]

The first line:

FROM golang:1.15.7-buster

References the official image for Go as the base image. This image comes with Go 1.15 pre-installed.

The second line:

RUN go get -u github.com/beego/bee

Installs the bee tool globally (Docker commands run as root by default), which will be used to live-reload our code during development.

Next, we configure the environment variables for Go:

ENV GO111MODULE=on

ENV GOFLAGS=-mod=vendor

There are two types of variables:

* **ARG**: these take effect at build time, we must set these values when we create the image.

ARG GROUP\_ID

ARG USER\_ID

* **ENV**: define run-time variables, these are activated when the container is started.

ENV APP\_USER app

ENV APP\_HOME /go/src/mathapp

The next lines:

RUN groupadd --gid $GROUP\_ID app && useradd -m -l --uid $USER\_ID --gid $GROUP\_ID $APP\_USER

RUN mkdir -p $APP\_HOME && chown -R $APP\_USER:$APP\_USER $APP\_HOME

USER $APP\_USER

WORKDIR $APP\_HOME

Creates a user called app, a home directory and an app directory inside the container.

The next to last line:

EXPOSE 8010

Tells Docker that port 8010 is interesting.

The final line:

CMD ["bee", "run"]

Uses the bee command to start our application.

**Step 2 – Building the Image**

Once the Docker file is created, run the following command to create the image:

$ docker build \

--build-arg USER\_ID=$(id -u) \

--build-arg GROUP\_ID=$(id -g) \

-t mathapp-development .

Executing the above command will create an image named mathapp:

* **–build-arg**: sets a build time variable. We’ll use it to make the user and group IDs in your machine and the container match.
* **-t mathapp**: sets the tag name for the new image, we can reference the image later as mathapp:latest
* Don’t forget to type the last dot (.) in the command.

This command can be used by everyone working on this application. This will ensure that an identical development environment is used across the team.

To see the list of images on your system, run the following command:

$ docker images

Note that the exact names and number of images might vary. However, you should see at least the golang and mathapp images in the list:

REPOSITORY TAG IMAGE ID CREATED SIZE

golang 1.15 25c4671a1478 2 weeks ago 809MB

mathapp-development latest 8ae092824585 60 seconds ago 838MB

**Step 3 – Running the Container**

Once you have mathapp, you can start a container with:

$ docker run -it --rm -p 8010:8010 -v $PWD/src:/go/src/mathapp mathapp-development

Let’s break down the above command to see what it does.

* The docker run command is used to run a container from an image,
* The -it flag starts the container in an interactive mode (tie it to the current shell),
* The --rm flag cleans out the container after it shuts down,
* The --name mathapp-instance names the container mathapp-instance,
* The -p 8010:8010 flag allows the container to be accessed at port 8010,
* The -v $PWD/src:/go/src/mathapp is more involved. It maps the src/ directory from the machine to /go/src/mathapp in the container. This makes the development files available inside and outside the container, and
* The mathapp part specifies the image name to use in the container.

Executing the above command starts the Docker container. This container exposes your application on port 8010. It also rebuilds your application automatically whenever you make a change. You should see the following output in your console:

\_\_\_\_\_\_

| \_\_\_ \

| |\_/ / \_\_\_ \_\_\_

| \_\_\_ \ / \_ \ / \_ \

| |\_/ /| \_\_/| \_\_/

\\_\_\_\_/ \\_\_\_| \\_\_\_| v1.10.0

2020/03/17 14:43:16 INFO ▶ 0001 Using 'mathapp' as 'appname'

2020/03/17 14:43:16 INFO ▶ 0002 Initializing watcher...

go: downloading github.com/astaxie/beego v1.12.1

go: downloading golang.org/x/crypto v0.0.0-20191011191535-87dc89f01550

go: downloading gopkg.in/yaml.v2 v2.2.1

go: downloading github.com/shiena/ansicolor v0.0.0-20151119151921-a422bbe96644

go: downloading golang.org/x/net v0.0.0-20190620200207-3b0461eec859

go: downloading golang.org/x/text v0.3.0

github.com/shiena/ansicolor

golang.org/x/text/transform

github.com/astaxie/beego/config

github.com/astaxie/beego/utils

gopkg.in/yaml.v2

github.com/astaxie/beego/logs

github.com/astaxie/beego/grace

github.com/astaxie/beego/session

github.com/astaxie/beego/toolbox

golang.org/x/crypto/acme

golang.org/x/text/unicode/bidi

golang.org/x/text/unicode/norm

github.com/astaxie/beego/context

golang.org/x/text/secure/bidirule

github.com/astaxie/beego/context/param

golang.org/x/net/idna

golang.org/x/crypto/acme/autocert

github.com/astaxie/beego

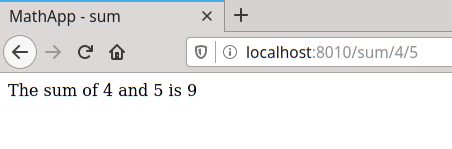
2020/03/17 14:43:24 SUCCESS ▶ 0003 Built Successfully!

2020/03/17 14:43:24 INFO ▶ 0004 Restarting 'mathapp'...

2020/03/17 14:43:24 SUCCESS ▶ 0005 './mathapp' is running...

2020/03/17 14:43:24.912 [I] [asm\_amd64.s:1373] http server Running on http://:8010

To check the setup, visit http://localhost:8010/sum/4/5 in your browser. You should see something similar to the following:



**Note:** This assumes that you’re working on your local machine.

To try the live-reload feature, make a modification in any of the source files. For instance, edit src/main.go, replace this line:

c.Data["operation"] = operation

To something like this:

c.Data["operation"] = "real " + operation

Bee should pick up the change, even inside the container, and reload the application seamlessly:

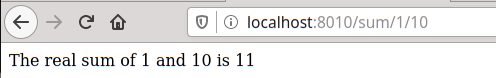
2020/03/17 18:50:38 SUCCESS ▶ 0011 Built Successfully!

2020/03/17 18:50:38 INFO ▶ 0012 Restarting 'mathapp'...

2020/03/17 18:50:38 SUCCESS ▶ 0013 './mathapp' is running...

2020/03/17 18:50:38.427 [I] [asm\_amd64.s:1373] http server Running on http://:8010

Now reload the page on the browser to see the modified message:



**Using Docker in Production**

This section will explain how to deploy a Go application in a Docker container. We will use Semaphore to do the following:

* Automatically build after changes are pushed to the git repository,
* Automatically run tests,
* Create a Docker image if the build is successful and if the tests pass, and
* Push the Docker image to Docker Hub.

**Creating a Dockerfile for Production**

We’ll write a new Dockerfile to create a complete, self-contained image; without external dependencies.

Enter the following contents in a new file called Dockerfile.deploy:

# Dockerfile.production

FROM registry.semaphoreci.com/golang:1.15 as builder

ENV APP\_USER app

ENV APP\_HOME /go/src/mathapp

RUN groupadd $APP\_USER && useradd -m -g $APP\_USER -l $APP\_USER

RUN mkdir -p $APP\_HOME && chown -R $APP\_USER:$APP\_USER $APP\_HOME

WORKDIR $APP\_HOME

USER $APP\_USER

COPY src/ .

RUN go mod download

RUN go mod verify

RUN go build -o mathapp

FROM debian:buster

FROM registry.semaphoreci.com/golang:1.15

ENV APP\_USER app

ENV APP\_HOME /go/src/mathapp

RUN groupadd $APP\_USER && useradd -m -g $APP\_USER -l $APP\_USER

RUN mkdir -p $APP\_HOME

WORKDIR $APP\_HOME

COPY src/conf/ conf/

COPY src/views/ views/

COPY --chown=0:0 --from=builder $APP\_HOME/mathapp $APP\_HOME

EXPOSE 8010

USER $APP\_USER

CMD ["./mathapp"]

Let’s take a detailed look at what each of these commands does. The first command:

FROM registry.semaphoreci.com/golang:1.15 as builder

Tells us this is a [multi-stage build](https://docs.docker.com/develop/develop-images/multistage-build/); it defines an intermediate image that will only have one job: compile the Go binary.

You might have noticed that we’re not pulling the image from Docker Hub, the default image registry. Instead, we’re using the Semaphore Docker Registry, which is more convenient, faster, and pulls don’t count against your Docker Hub [rate limits](https://docs.docker.com/docker-hub/download-rate-limit/).

The following commands:

RUN groupadd $APP\_USER && useradd -m -g $APP\_USER -l $APP\_USER

RUN mkdir -p $APP\_HOME && chown -R $APP\_USER:$APP\_USER $APP\_HOME

WORKDIR $APP\_HOME

USER $APP\_USER

COPY src/ .

Creates the home and application directories for the app user. Application users are optional, but they are considered good practice to avoid running all processes as root.

The last commands in the intermediate image download the modules and build the executable:

RUN go mod download

RUN go mod verify

RUN go build -o mathapp

Next comes the final and definitive container, where we will run the services. We don’t need a full Go installation to run the executable so we can start from a smaller Debian image:

FROM registry.semaphoreci.com/golang:1.15

We use the COPY command to copy files into the image.

* **chown**: changes the owner and group of the files and directories.
* **from**: copies the executable from the intermediate builder image.

COPY src/conf/ conf/

COPY src/views/ views/

COPY --chown=0:0 --from=builder $APP\_HOME/mathapp $APP\_HOME

Then we change the active user:

USER app

We finalize by exposing the port and starting the binary:

EXPOSE 8010

CMD ["./mathapp"]

To build the deployment image:

$ docker build -t mathapp-production -f Dockerfile.production .

You can run it with:

$ docker run -it -p 8010:8010 mathapp-production

Notice that we don’t need to map any directories, as all the source files are included in the container.

**Continuous Integration with Semaphore**

Docker is a great solution to package and deploy Go applications. The only downside is the additional steps required to build and test the image. This hurdle is easily is best dealt with [Continuous Integration and Continuous Delivery](https://semaphoreci.com/cicd) (CI/CD).

A [Continuous Integration](https://semaphoreci.com/continuous-integration) (CI) platform can test our code on every iteration, on every push and every merge. Developers adopting CI no longer have to fear of merging branches, nor be anxious about release day. In fact, CI lets developers merge all the time and make safe releases any day of the week. A good CI setup will run a series of comprehensive tests, like the ones we prepared so far, to weed out any bugs.

Once the code is ready, we can extend our CI setup with [Continuous Delivery](https://semaphoreci.com/cicd) (CD). CD can prepare and build the Docker images, leaving them ready to deploy at any time.

**Push the Code to GitHub**

Let’s push our modifications to GitHub:

* Open .gitignore and uncomment the vendor/ line, so vendored modules are not committed:

# Dependency directories (remove the comment below to include it)

vendor/

# Build artifact

src/mathapp

* Push all the code with git:

$ git add Dockerfile\*

$ git add src

$ git add .gitignore

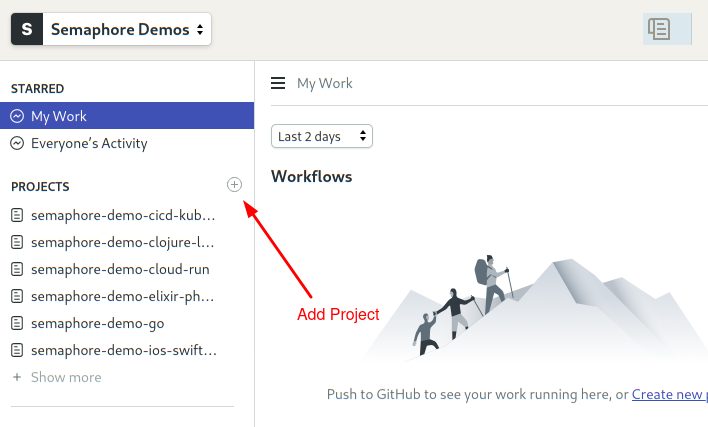
$ git commit -m "initial commit"

$ git push origin master

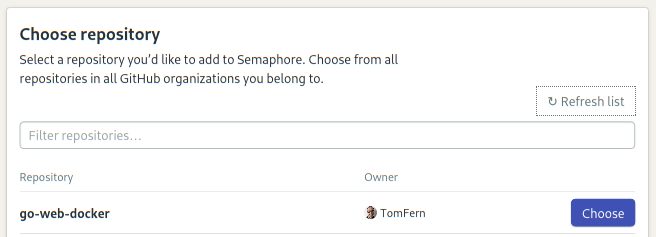
**Adding the Repository to Semaphore**

We can add CI to our project for free in just a few minutes:

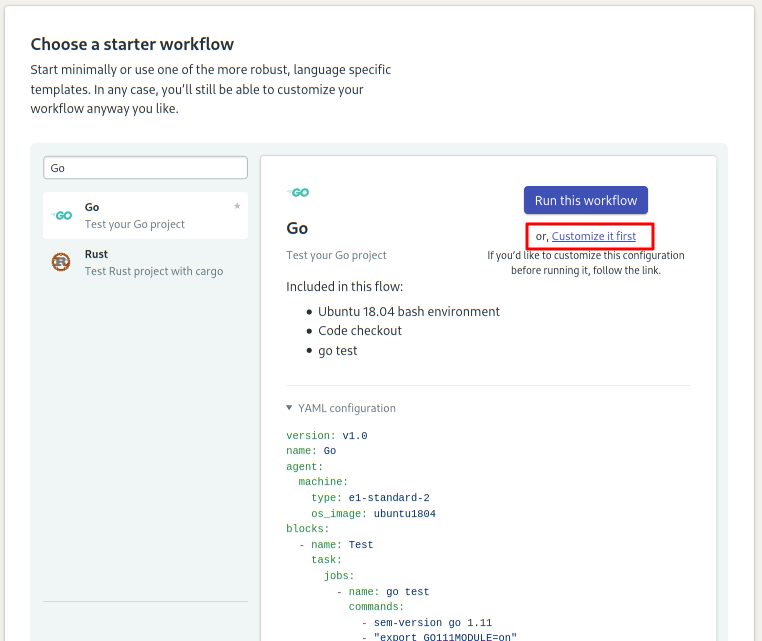
* Go to Semaphore and sign up using the **Sign up with GitHub** button. This will link up both accounts.
* Click on the **+ (plus sign)** next to **Projects** to create a new project:



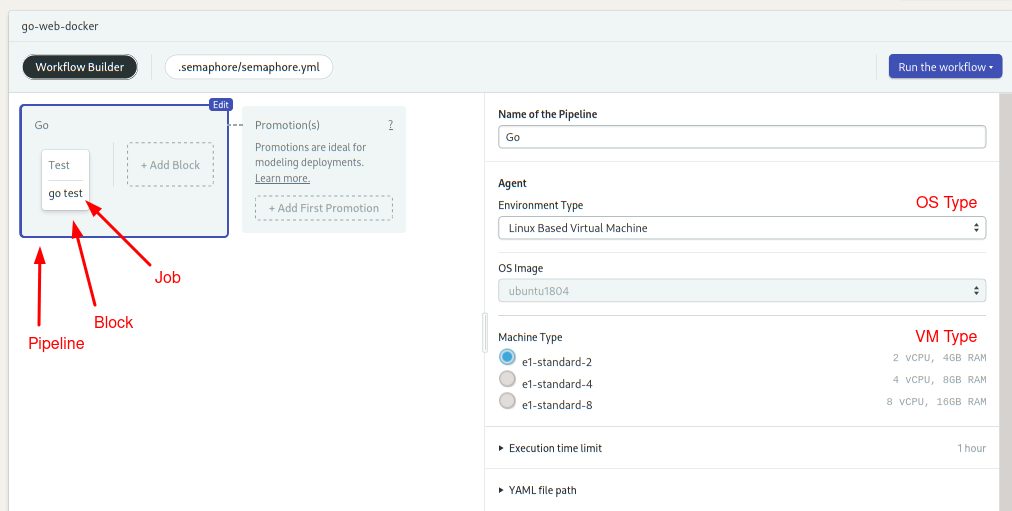
* Find your GitHub repository and click on **Choose**:



* Select the **Go** starter workflow. Click on **Customize it first**:



You’ll get the **Workflow Editor**. Here’s an overview of how it works:

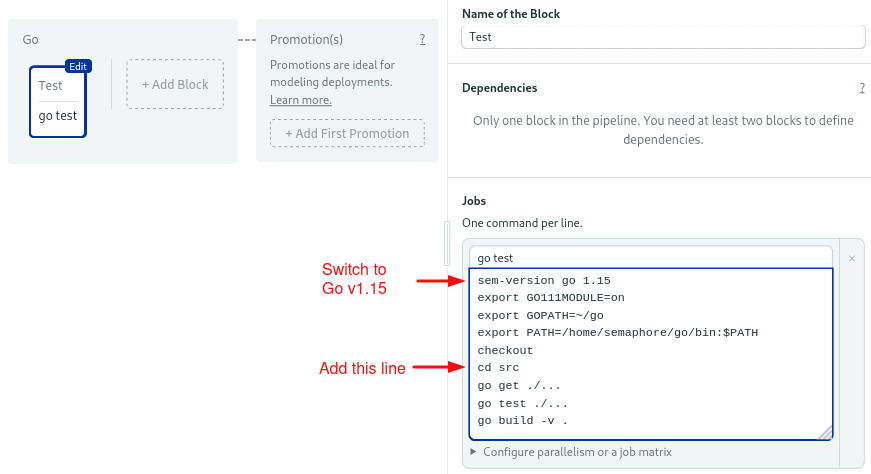


* **Pipeline**: A [pipeline](https://semaphoreci.com/blog/cicd-pipeline) has a specific objective, e.g. building or testing. Pipelines are made of blocks that are executed from left to right in an agent.
* **Agent**: The agent is the virtual machine that powers the pipeline. We have three [machine types](https://docs.semaphoreci.com/ci-cd-environment/machine-types/) to choose from. The machine runs an optimized [Ubuntu 18.04](https://docs.semaphoreci.com/ci-cd-environment/ubuntu-18.04-image/) image with build tools for many languages.
* **Block**: blocks group jobs that can be executed in parallel. Jobs in a block usually have similar commands and configurations. Once all jobs in a block complete, the next block begins.
* **Job**: jobs define the commands that do the work. They inherit their configuration from their parent block.

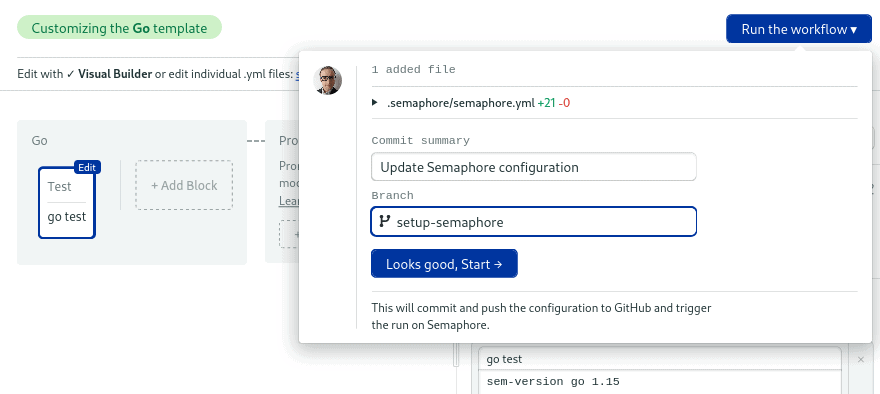
Coming back to our setup. The started workflow expects the code at the project’s root, but our code is inside the src directory so we need to make a small modification:

* Click on the **Test** block.
* On the right side, you’ll find the job’s commands. Add the following line right after the checkout line and before the go get ./... line:

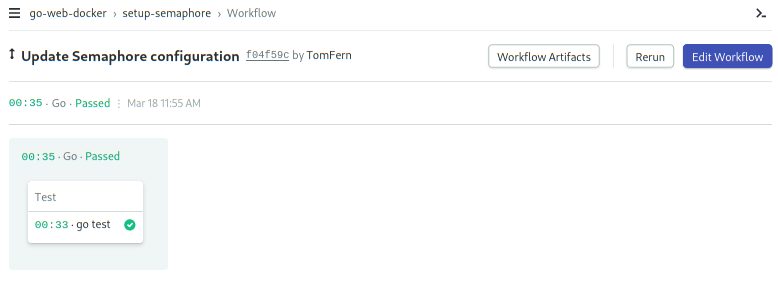
cd src



* Click on the **Run the Workflow** and then on **Start** to get the pipeline running:



If all goes well, after a few seconds the job should be completed without errors:

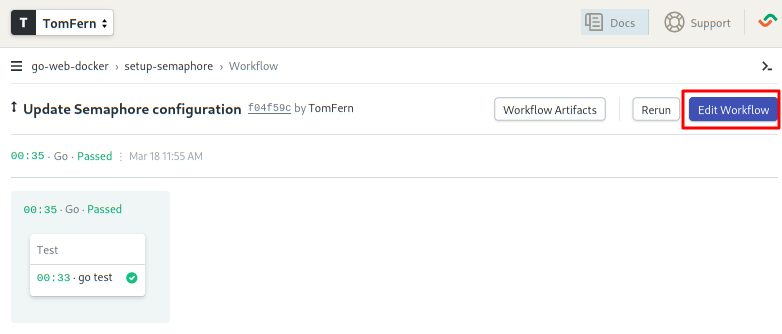


**Enhancing the CI Pipeline**

In this section, we’ll modify the pipeline so that:

* Go dependencies are cached to having to avoid re-download on each run.
* Tests get their own block so we can scale out testing more easily.

To get started, click on the **Edit Workflow button:**



* Click on the block. We’ll completely replace its contents.
* Change the name of the block and the job to “Install”.
* Open the **Environment Variables** section and create these variables:
  + GO111MODULE = on
  + GOFLAGS = -mod=vendor
* Type the following content in the Job command box:

sem-version go 1.15

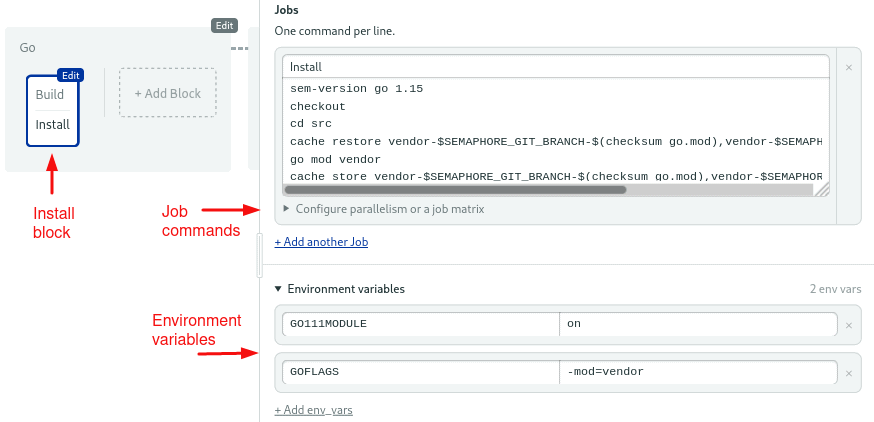
checkout

cd src

cache restore vendor-$SEMAPHORE\_GIT\_BRANCH-$(checksum go.mod),vendor-$SEMAPHORE\_GIT\_BRANCH,vendor-master

go mod vendor

cache store vendor-$SEMAPHORE\_GIT\_BRANCH-$(checksum go.mod),vendor-$SEMAPHORE\_GIT\_BRANCH,vendor-master vendor



I think this is a good opportunity to learn about the Semaphore [toolbox](https://docs.semaphoreci.com/reference/toolbox-reference/) of built-in commands:

* **checkout**: the checkout commands clones the correct revision of the GitHub repository and changes the directory. It’s usually the first command in a job.
* **sem-version**: with sem-version, we can switch the active version of a language. Semaphore fully supports many languages, including [Go](https://docs.semaphoreci.com/programming-languages/go/).
* **cache**: the cache is a project file storage. We’ll use the cache to persist the vendor/ directory.

Let’s go back to our pipeline:

* Use the **+ Add Block** dotted line button to create a new block.
* Call the block and the job “Test”.
* Open the **Environment Variables** section and create the GO111MODULE and GOFLAGS variables like we did on the previous block.
* Open the **Prologue** section, which executed before each job in the block, and type the following commands:

sem-version go 1.15

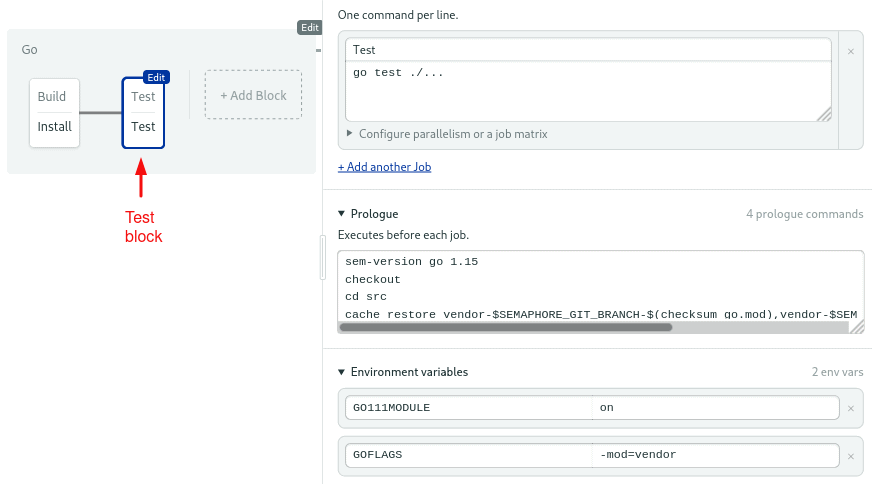
checkout

cd src

cache restore vendor-$SEMAPHORE\_GIT\_BRANCH-$(checksum go.mod),vendor-$SEMAPHORE\_GIT\_BRANCH,vendor-master

* Type the following command in the job:

go test ./...



* Click on **Run the Workflow** and **Start** to try the updated pipeline.

**Building the Docker Image**

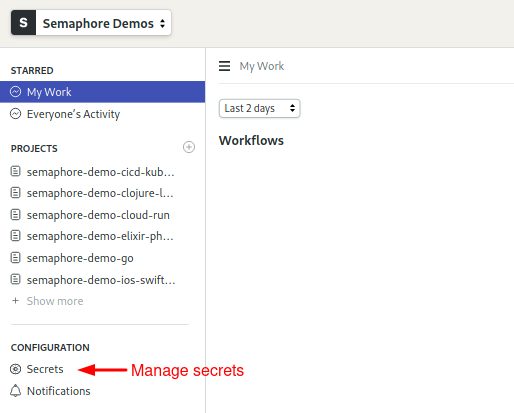
So far all we did enters in the Continuous Integration category, the natural next stage is to pack the application in a Docker container.

We’ll create a new delivery pipeline to:

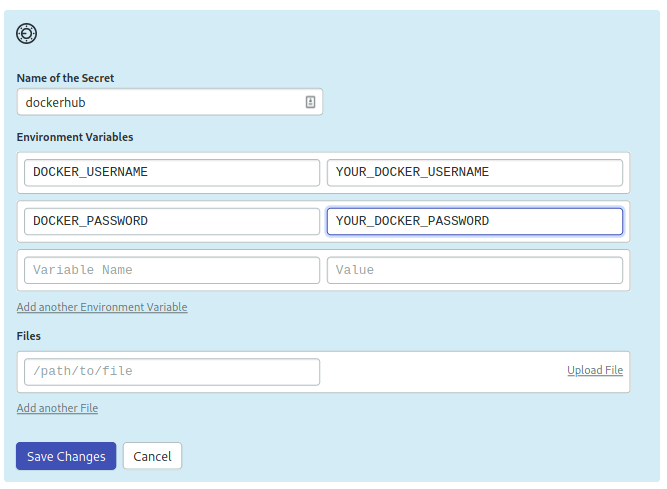
* Build a Docker Image with our Go binary and HTML templates.
* Upload the image to Docker Hub so it’s ready for deployment.

First, we have to tell Semaphore how to connect to Docker Hub:

* On the left navigation menu, click on **Secrets** under **Configuration**:



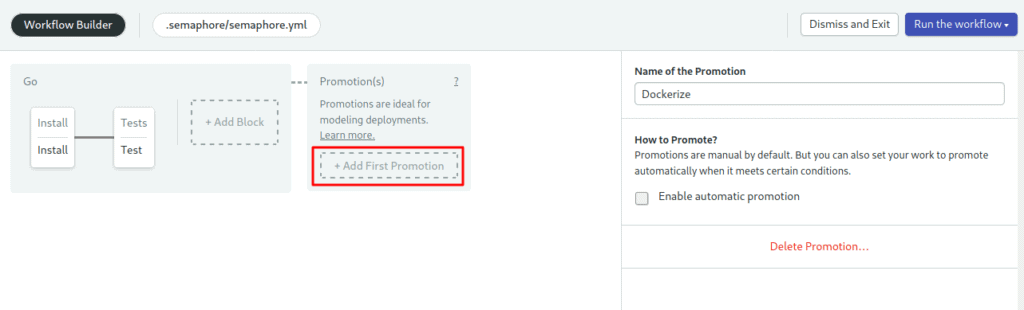
* Click **Create New Secret**.
* Create two variables for your Docker Hub username and password:
  + DOCKER\_USENAME = YOUR DOCKER USERNAME
  + DOCKER\_PASSWORD = YOU DOCKER PASSWORD



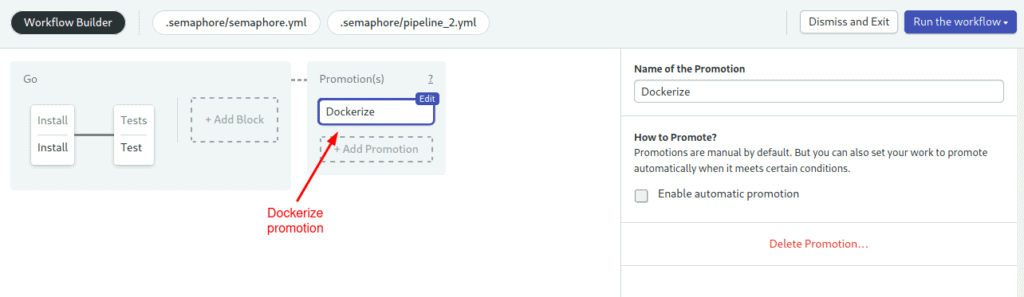
* Click on **Save**.

Going back to the pipeline:

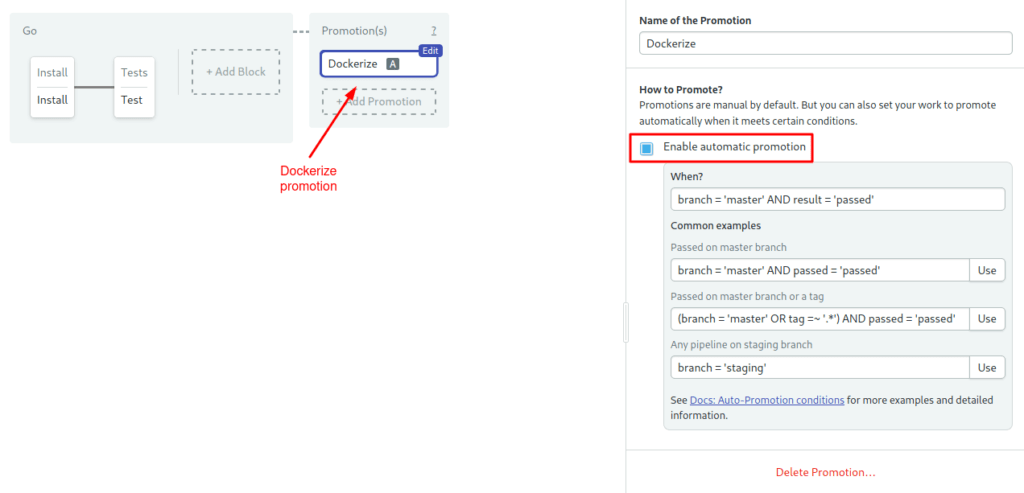
* Click on **Edit Workflow**.
* Use the **+Add First Promotion** button to create a new linked pipeline:



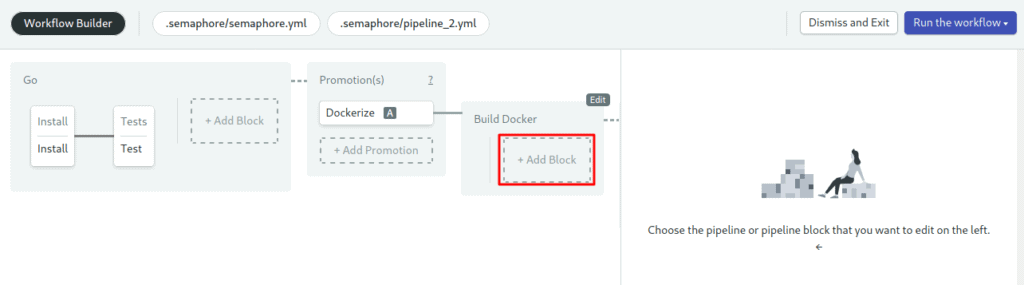
* Change the name of the pipeline to “Dockerize”:



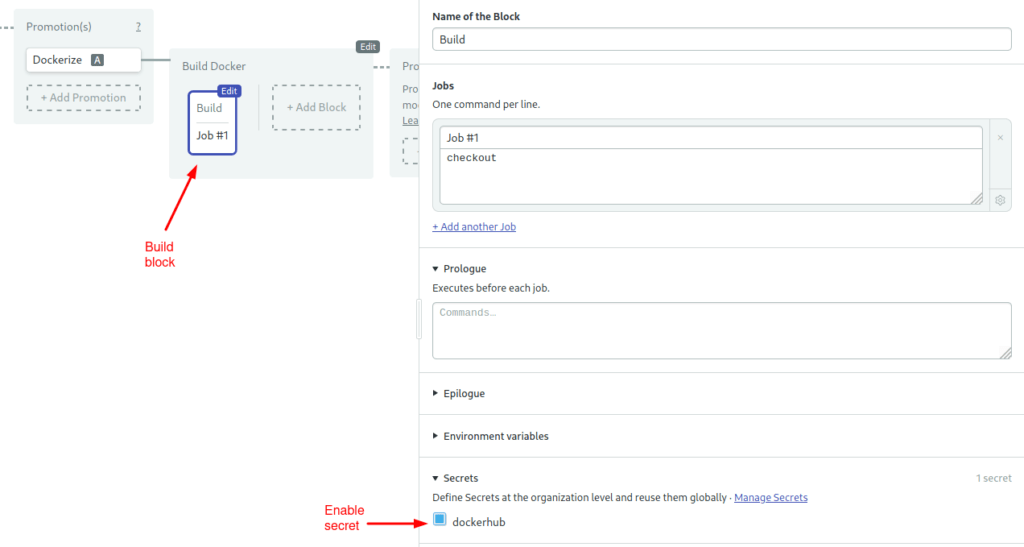
* Check **Enable automatic promotion**. You can set conditions to trigger the pipeline here:



* Click **+Add Block**. We’ll call the new block “Build”



* Open the **Secrets** section and check the **dockerhub** box. This will import the variables we created earlier into the jobs in the block:



* Type the following commands in the job:

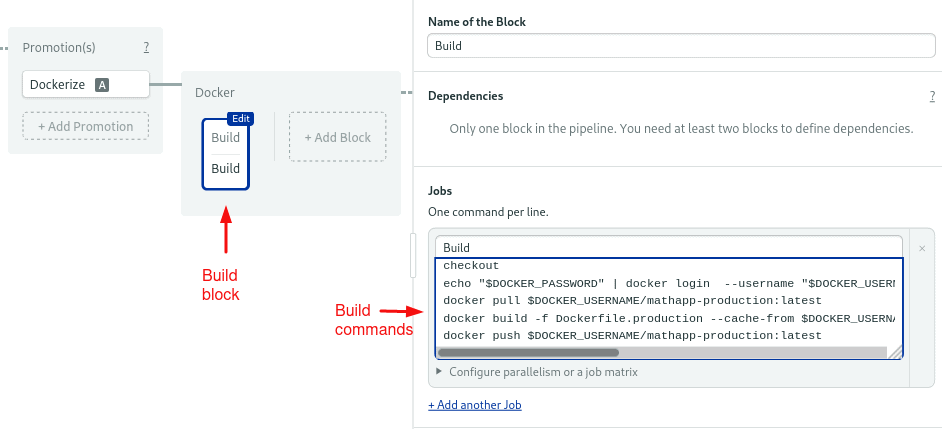
checkout

echo "$DOCKER\_PASSWORD" | docker login --username "$DOCKER\_USERNAME" --password-stdin

docker pull $DOCKER\_USERNAME/mathapp-production:latest

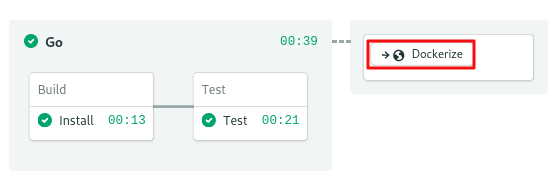
docker build -f Dockerfile.production --cache-from $DOCKER\_USERNAME/mathapp-production:latest -t $DOCKER\_USERNAME/mathapp-production:latest .

docker push $DOCKER\_USERNAME/mathapp-production:latest

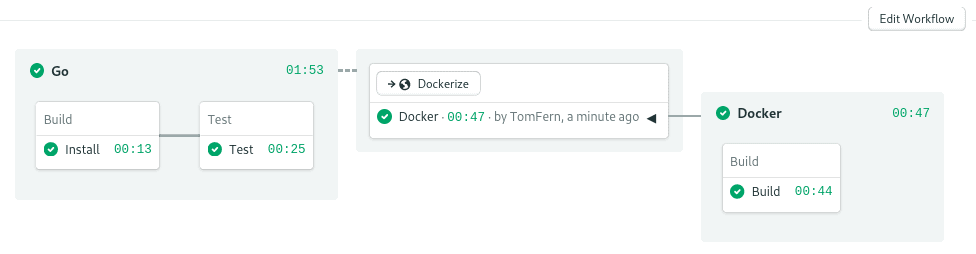


* Click on **Run the Workflow** and **Start**.

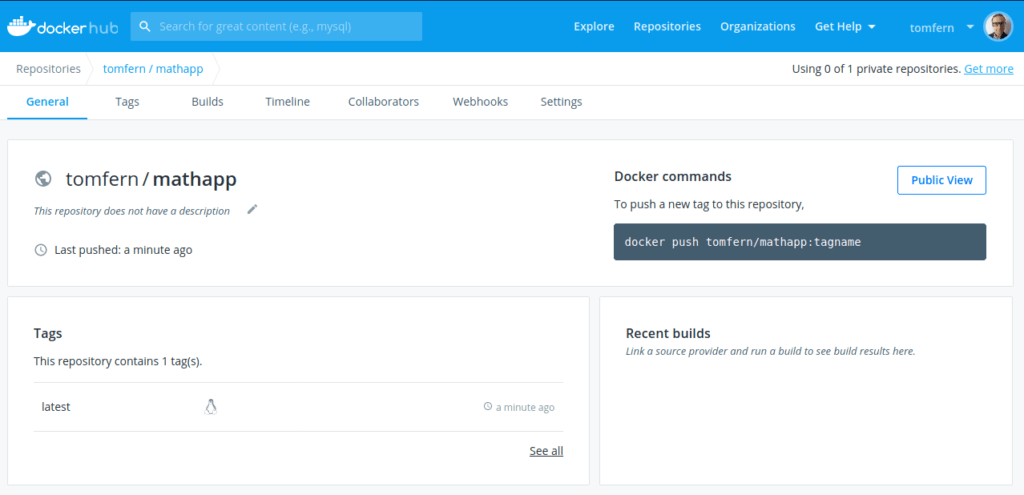
Once the first two blocks are done, click on the **Promote** button:



Wait a few seconds until the Dockerize pipeline is done:



Check your Docker Hub repositories, you should find the new image, ready to use:



Pull and test the new image in your machine:

$ docker pull YOUR\_DOCKERHUB\_USERNAME/mathapp-production

$ docker run -it -p 8010:8010 YOUR\_DOCKERHUB\_USERNAME/mathapp-production