

## Build your Docker images automatically when pushing new code to GitHub

In this tutorial, we will explore two techniques to automatically build and publish Docker images to Docker Hub every time you push new code into your GitHub source code repository.

### Overview

This article will outline two techniques for automating the build of Docker images after every push or merge to the master branch in **GitHub**. The built images will be pushed to a **Docker Hub** container repository.

The two approaches being demonstrated are;

1. using Docker Hub – use GitHub webhooks to notify Docker Hub about code changes and trigger the build of a new Docker image within Docker Hub itself.
2. using GitHub Actions – using GitHub's service for running Continuous Integration pipelines, we will build the new Docker image within GitHub machines and push the image to Docker Hub.

## Prerequisites

To begin this tutorial, you will need to set yourself up with accounts on GitHub and Docker Hub.

### 1. GitHub account ([register here](#))

Within your GitHub account, create a repository with a *Dockerfile* and any additional source code. I'm using a simple Hello World example for the purposes of this article.

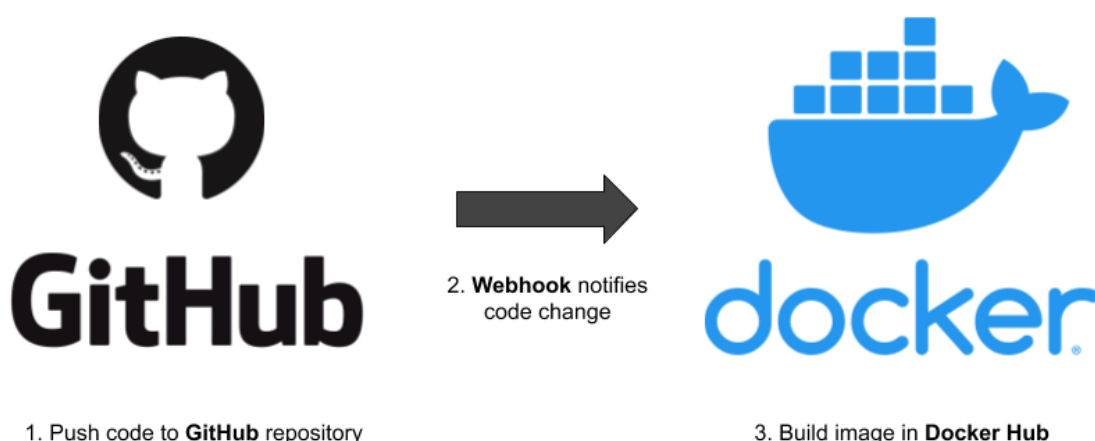
```
Dockerfile > ...  
1 FROM busybox  
2 CMD echo "Hello world!"
```

"Hello World" Dockerfile

### 2. Docker Hub account ([register here](#))





## Approach #1 — using Docker Hub to build image

In the first approach, we will configure Docker Hub to receive notifications from GitHub whenever there are any changes to our source repository. This is achieved using GitHub webhooks. On receipt of the notification, Docker Hub will build the new Docker image and publish it for consumption.



## Step 1. Associate your GitHub and Docker Hub accounts.

Within Docker Hub visit [Account Settings > Linked Accounts](#) and click "Connect" to allow access to your source repositories.

Provider	Account			
GitHub	No account linked			<a href="#">Connect</a>
Bitbucket	No account linked			<a href="#">Connect</a>

## Step 2. Create container repository in Docker Hub

Within Docker Hub [create a new repository](#) and under “Build Settings” click on the GitHub icon to associate your source code repository.

### Build Settings *(optional)*

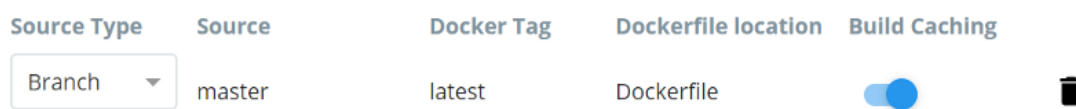
Autobuild triggers a new build with every **git push** to your source code repository. [Learn More.](#)



From the drop-down select your GitHub *organisation* (this will default to your username) and the *source code repository*.

## Step 3. Configure Build Rules

There are many options available, and several examples are displayed in the help. For the purpose of this demo, we’ll keep the default settings and set up the trigger to be on pushes to the *master* branch and use the *latest* docker tag.



Other options exist to create Docker images from tags or release branches, and the pattern matching allows you to create dynamic image tags.

Scenario	Source Type	Source	Docker Tag	Matches	Docker Tag Built
Exact match	Branch	master	latest	master	latest
Match versions	Tag	/^[0-9.]+\$	release-{sourceref}	1.2.0	release-1.2.0
Trailing modifiers	Tag	/^[0-9.]+/	release-{sourceref}	1.2.0-rc	release-1.2.0-rc
Extract version number	Tag	/^v([0-9.]+)\$/	version-{v1}	v1.2.3	version-1.2.3

Example Build Rule options

Click “Create & Build” to set up your new container repository and build your first Docker image — the first build will trigger automatically.

## Step 5. Viewing builds.

Within Docker Hub you will find the build status in the “General” tab of your repository. Viewing the “Builds” tab will show more information about each build. From here you will see the status of jobs and view build logs.

Docker Tag	Source	Latest Build Status	Autobuild	Build caching	
latest	master	IN PROGRESS	✓	✓	Trigger ▶

Once built we can pull and run the newly built image.





```
$ docker run davelms/hello-world:latest
```

```
Hello world!
```

## Step 6. Push new source code changes








Final step is to simulate a code change — I’ll do this by editing the “Hello world!” message — and push the new commit up to GitHub.

GitHub will send a notification to Docker Hub about the code change; Docker Hub will initiate a new build; and we can run that new image once completed.

Change message to Hello Dave davelms committed now	Verified		888874f	
Initial Dockerfile davelms committed 2 hours ago ✓	Verified		42b8c79	

GitHub commit log

Immediately after pushing the code, we see a new build initiated in Docker Hub. You can see that the commit sha is referenced — tip: you can use this in tags should you wish instead of “latest”.

Recent Builds				
 Build in 'master' (888874fa)	 latest	<a href="#">888874f</a>	 a minute ago	
 Build in 'master' (42b8c799)	 latest	<a href="#">42b8c79</a>	 15 minutes ago	

Docker Hub build history

When the build has finished, update locally with `docker pull` and re-run your container. This time we see the output from our new source code.

**\$ `docker run davelms/hello-world`**

Hello Dave!

## Approach #2 — using GitHub Actions to build Docker images

This time we will use GitHub’s custom CI/CD platform — GitHub Actions — to build the Docker image after every push to the source code repository. The workflow will define a single job that builds the image and pushes the new image to Docker Hub.



3. Image is pushed to Docker Hub



1. Push code to **GitHub** repository
2. **GitHub Actions** will build image

## Step 1. Create a Docker Hub security access token

First of all, within Docker Hub create yourself an access token by visiting [Settings > Security](#). Give it a name you recognise it later.

Access Tokens					<a href="#">New Access Token</a>
<input type="checkbox"/>	DESCRIPTION	LAST USED	CREATED	ACTIVE	
<input type="checkbox"/>	GitHub Actions	Never	May 16, 2020 22:23:20	Yes	<a href="#">⋮</a>

Once created, head over to GitHub and create secrets within your source code repository for **DOCKER\_HUB\_USERNAME** and **DOCKER\_HUB\_TOKEN**, along with **DOCKER\_HUB\_REPOSITORY**.

DOCKER_HUB_REPOSITORY	Updated now	<a href="#">Update</a>	<a href="#">Remove</a>
DOCKER_HUB_TOKEN	Updated 10 minutes ago	<a href="#">Update</a>	<a href="#">Remove</a>
DOCKER_HUB_USERNAME	Updated 10 minutes ago	<a href="#">Update</a>	<a href="#">Remove</a>

## Step 2. Create a GitHub Action to build and push images

Keeping within GitHub, head into the “Actions” tab of your source code repository. It’s likely that it will have detected the Dockerfile

and will recommend you Docker-related workflow examples to get you started.

Since I will share an example, skip these helpers for now and select “set up a workflow yourself”. Use the workflow definition below and commit the file.

```
name: Docker Image CI
```

```
on:
```

```
  push:
```

```
    branches: [ master ]
```

```
jobs:
```

```
  build:
```

```
    runs-on: ubuntu-latest
```

```
    steps:
```

```
      - uses: actions/checkout@v2
```

```
      - name: Build the Docker image
```

```
        run: |
```

```
          echo "${{ secrets.DOCKER_HUB_TOKEN }}" | docker login -u "${{ secrets.DOCKER_HUB_USERNAME }}" --password-stdin docker.io
```

```
          docker build . --file Dockerfile --tag docker.io/${{ secrets.DOCKER_HUB_USERNAME }}/${{ secrets.DOCKER_HUB_REPOSITORY }}:${GITHUB_SHA}
```

```
          docker push docker.io/${{ secrets.DOCKER_HUB_USERNAME }}/${{ secrets.DOCKER_HUB_REPOSITORY }}:${GITHUB_SHA}
```

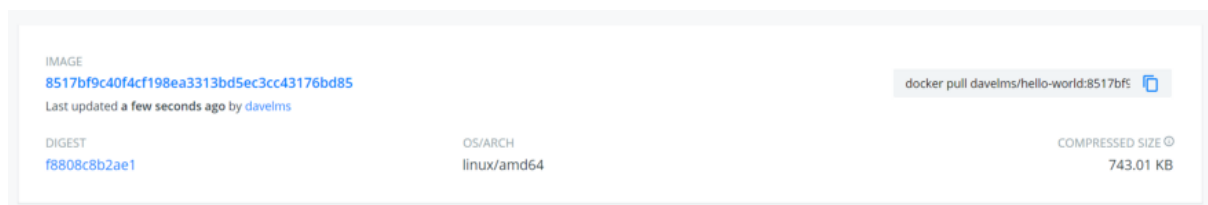
## What does this CI pipeline do?

1. Define that we will trigger a job after all pushes to master branch.

2. Configures a single job called “build” to run on an Ubuntu machine.
3. The source code repository will be checked out and will run three inline commands: 1. `docker login` 2. `docker build` 3. `docker push`. These will use our secrets we saved off earlier.
4. Note that the example uses the commit sha as the image tag — GitHub Actions makes the sha available in an environment variable called `GITHUB_SHA`. Of course, you could stick with “latest” like we did in the first example should you wish to do so.

### Step 3. View the build

When the build has finished, the image will appear over on Docker Hub.



Now verify you can pull the new image down successfully.

```
$ docker run
```

```
davelms/hello-world:8517bf9c40f4cf198ea3313bd5ec3cc43176bd85
```

```
Unable to find image
```

```
'davelms/hello-world:8517bf9c40f4cf198ea3313bd5ec3cc43176bd85' locally
```

```
8517bf9c40f4cf198ea3313bd5ec3cc43176bd85: Pulling from davelms/hello-world
```

```
d9cbbca60e5f: Already exists
```

```
Digest:
```

```
sha256:f8808c8b2ae19f6f3700e51a127e04d8366a1285bdfc6e4006092807f0eced1b
```

```
Status: Downloaded newer image for
```

```
davelms/hello-world:8517bf9c40f4cf198ea3313bd5ec3cc43176bd85
```





```
Hello Dave!
```

### Step 4. Push new source code changes



Final step is to simulate a code change — I'll do this by editing the message back to "Hello world!" — and push the new commit up to GitHub.

Commits on May 16, 2020

Changing message back to Hello World davelms committed now	Verified		a26c531	
Update dockerimage.yml davelms committed 12 minutes ago	Verified		8517bf9	

Commit log

After the push, a new CI/CD workflow containing our "build" job is run.

Event	Status	Branch	Actor
✓ Changing message back to Hello World Docker Image CI #4: Commit a26c531 pushed by davelms		master	1 minute ago 1m 21s
✓ Update dockerimage.yml Docker Image CI #3: Commit 8517bf9 pushed by davelms		master	13 minutes ago 17s

CI Pipeline history

The new image will have been pushed to Docker Hub successfully.

IMAGE a26c53183fac84a9c7ce128ce6ae6250fae26c7d Last updated a minute ago by davelms	docker pull davelms/hello-world:a26c531
DIGEST 3cfa80d4fa8c	OS/ARCH linux/amd64
	COMPRESSED SIZE 743.01 KB

This time we see the output from our new source code — the message is back to "Hello World!". All is working as expected.

**\$ docker run**

**davelms/hello-world:a26c53183fac84a9c7ce128ce6ae6250fae26c7d**

Unable to find image

'davelms/hello-world:a26c53183fac84a9c7ce128ce6ae6250fae26c7d' locally

a26c53183fac84a9c7ce128ce6ae6250fae26c7d: Pulling from davelms/hello-world

d9cbbca60e5f: Already exists

Digest:

sha256:3cfa80d4fa8c51271928f0294d89293a7a7fc7022a416a1758fc37394bc12808

Status: Downloaded newer image for  
davelms/hello-world:a26c53183fac84a9c7ce128ce6ae6250fae26c7d

**Hello World!**

## Conclusion

In this article we walked through two techniques for automating the build of Docker images after every source code change in GitHub. In both examples, we pushed the images into Docker Hub.

Firstly, we used webhooks and built the image directly within **Docker Hub** where the “free plan” offers the ability to create images (the limitation with this plan is that it limits to building one image at a time).

Next, we used **GitHub Actions** to build a Continuous Integration pipeline and push the built image to Docker Hub — we could extend this pipeline later to include some unit and integration tests. GitHub Actions provides 2,000 minutes of machine time per month under its “free plan”.