Philipp Hauer's Blog (https://blog.philipphauer.de/)

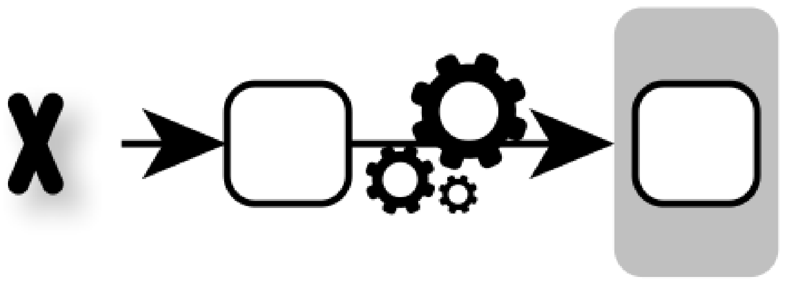
Web Architecture, Java Ecosystem, Software Craftsmanship

Tutorial: Continuous Delivery with Docker and Jenkins

POSTED ON NOV 7, 2015

Introducing Continuous Delivery means to automate the delivery process and to release our application frequently. This way, we improve the reliability of the release process, reduce the risk and get feedback faster. However, setting up a Continuous Delivery pipeline can be difficult in the beginning. In this step by step tutorial I will show you how to configure a simple Continuous Delivery pipeline using Git, Docker, Maven and Jenkins.

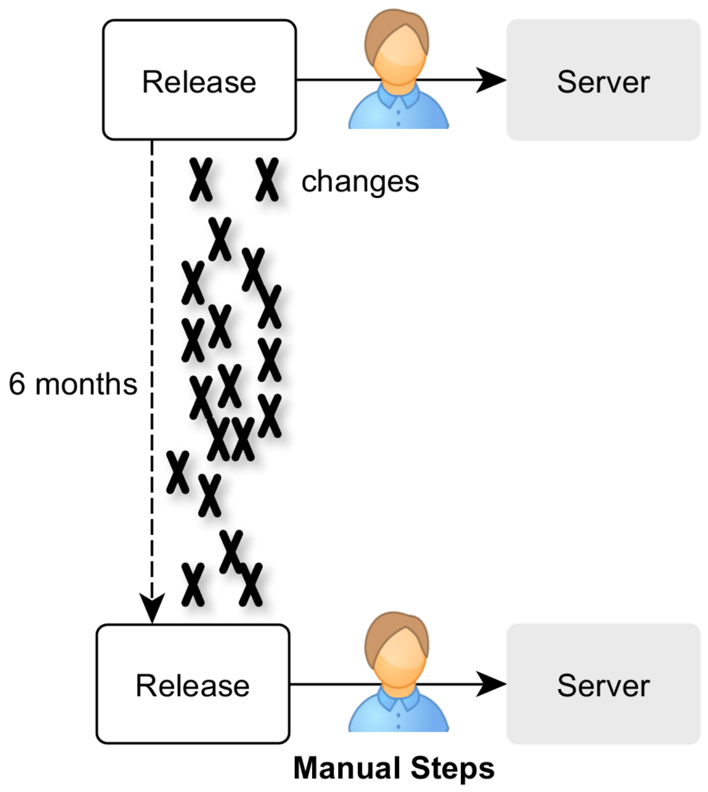
- Introduction to Continuous Delivery
  - Traditional Release Cycle
  - Continuous Delivery
  - Continuous Delivery using Docker
- Setting up a Simple Continuous Delivery Pipeline with Docker
  - Update October 2016
  - Preconditions
  - Sources
  - The Example Application and the Advantage of Using Runnable Fat Jars
  - Big Picture
  - Warning: Technical Spike
  - Step by Step
    - The Commit Stage
    - Acceptance Test Stage
    - Deployment Stage
- Further Readings



Introduction to Continuous Delivery

Traditional Release Cycle

Following the "old-school" release approach means to ship a release after a certain amount of time (let's say 6 months). We have to package the release, test it, set up or update the necessary infrastructure and finally deploy it on the server.



(/blog/2015/1107-tutorial-continuous-delivery-with-docker-jenkins/Overview1-Traditional-Release-

Cycle.png)

Traditional release cycle: rare releases and manual release process

What are the problems about this approach?

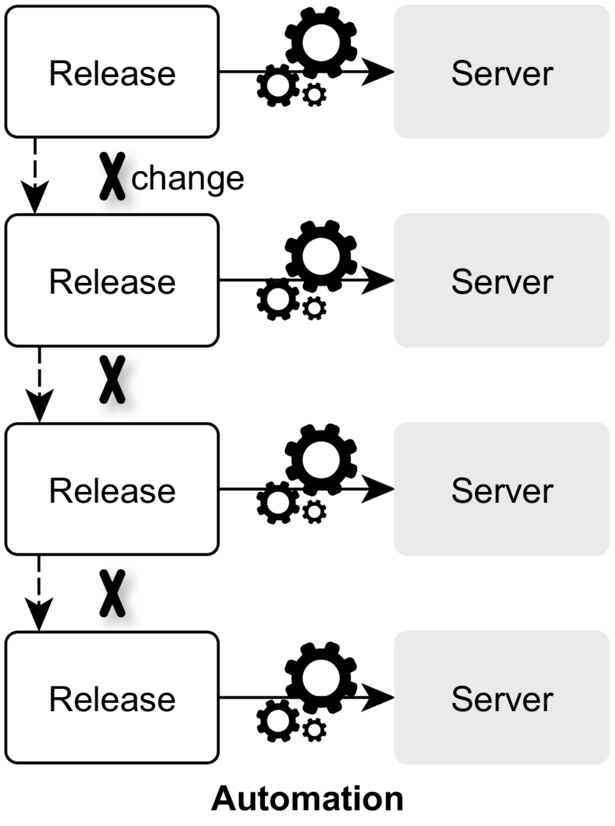
- The release process is done rarely. Consequently, we are barely practiced in releasing. Mistakes can happen more easily.
- Manual steps. The release process consists of a lot of steps which have to be performed manually (shutdown, set up/update infrastructure, deployment, restart and manual tests). The consequences:
  - Mistakes are more likely to happen when executing these steps manually.
  - The whole release process is more laborious, cumbersome and takes more time.
- There have been a lot of changes performed since the last release 6 months ago. It's likely that
  - something will go wrong when trying to put the different components together (e.g. version conflicts, side-effects, incompatible components) or
  - that there are bugs in the application itself.
  - But it is hard to see what change is causing the problem, because there have been a lot of changes. The point is that those problems are discovered too late. We are getting feedback too late, because we are trying to release the application too late in the development process and not regularly. We are only trying to release, when we really want to create a release.

All in all, we have a high risk that something will go wrong during our release process or that the application will contain bugs. Releases are dangerous and scary, aren't they? That's probably why releases are done so rarely. But doing them rarely makes them even more dangerous and scary. What can we do instead?

Continuous Delivery

> "If it hurts do it more often and bring the pain forward."
> A Practical Guide to Continuous Delivery (https://www.amazon.com/gp/product/B06XBCJHDX/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=B06XBCJHDX&linkCode=as2&tag=blogphilippha-20&linkId=a79fb331219fe4b6d8eabe1516fa0cf3) by Eberhard Wolff

We reduce the pain of releasing by releasing more often. Therefore, we have to automate the whole release process (including package release, set up/update infrastructure, deploy, final tests) and eliminate all manual steps. This way we can increase the release frequency.



(/blog/2015/1107-tutorial-continuous-delivery-with-docker-jenkins/Overview2-ContinuousDelivery.png)

Continuous Delivery: Increased release frequency and automation of the release process.

What are the benefits of this approach?

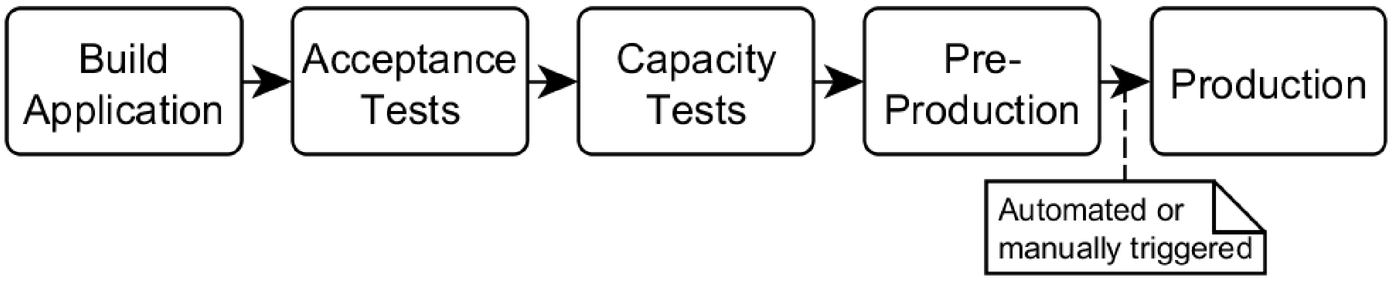
- Fewer mistakes can happen during an automated process in comparison to a manual one.
- There are fewer changes done between two releases. The danger of mistakes is quite small and we can easily track them back to the causing change.
- We don't package and ship our application at the end of a development phase. We are doing it early and frequently. This way we'll discover problems in the release process very soon.
- Due to the automated release process we can bring business value faster into production and therefore reduce the time-to-market.
- Deploying our application into production is low-risk, because we just execute the same automated process for the production as we did for the tests or the pre-production system.

All in all, Continuous Delivery is about

- **reduced risks,**
- **increased reliability,**
- faster feedback,
- accelerated release speed and time-to-market.

Continuous Delivery using Docker

From the technical point of view Continuous Delivery revolves around the automation and optimization of the delivery pipeline. A simple delivery pipeline could look like this:



(/blog/2015/1107-tutorial-continuous-delivery-with-docker-jenkins/Delivery-Pipeline.png)

A Delivery Pipeline

The big challenge is the automated setup of the infrastructure and environment, our application needs to run. And we need this infrastructure for _every stage_ of our delivery pipeline. Fortunately, Docker is great at creating reproducible infrastructures (/discussing-docker-pros-and-cons/). Using Docker we create an image that contains our application and the necessary infrastructure (/discussing-docker-pros-and-cons/#Advantages_for_the_Development_Team) (for instance the application server, JRE, VM arguments, files, permissions). The only thing we have to do is to execute the image in every stage of the delivery pipeline and our application will be up and running. Moreover, Docker is a (lightweight) virtualization, so we can easily clean up old versions of the application *and its infrastructure* just by stopping the Docker container.

Setting up a Simple Continuous Delivery Pipeline with Docker

Update October 2016

Please mind that some information in this post are a little bit dated. Some points:

- Please take a look at my post about Improving your Continuous Integration Setup with Docker and GitLab-CI (/improving-continuous-integration-setup-docker-gitlab-ci/)
- Jenkins 1.0 doesn’t support delivery pipelines as a first class citizen. Consider using Jenkins 2.0 (https://jenkins.io/2.0/) or GitLab-CI (https://about.gitlab.com/gitlab-ci/) instead.
- Use Docker-Compose (https://docs.docker.com/compose/) instead of the clumsy shell scripts calling `docker run` .

Preconditions

I used Ubuntu 14.04 LTS and Docker 1.8.2 for this tutorial. I highly recommend to use a Linux distribution in order to run Docker natively. Sure, you can try docker-toolbox/boot2docker for Windows and Mac OS X. But I wasn´t happy with this tool and the additional VM layer in daily use.

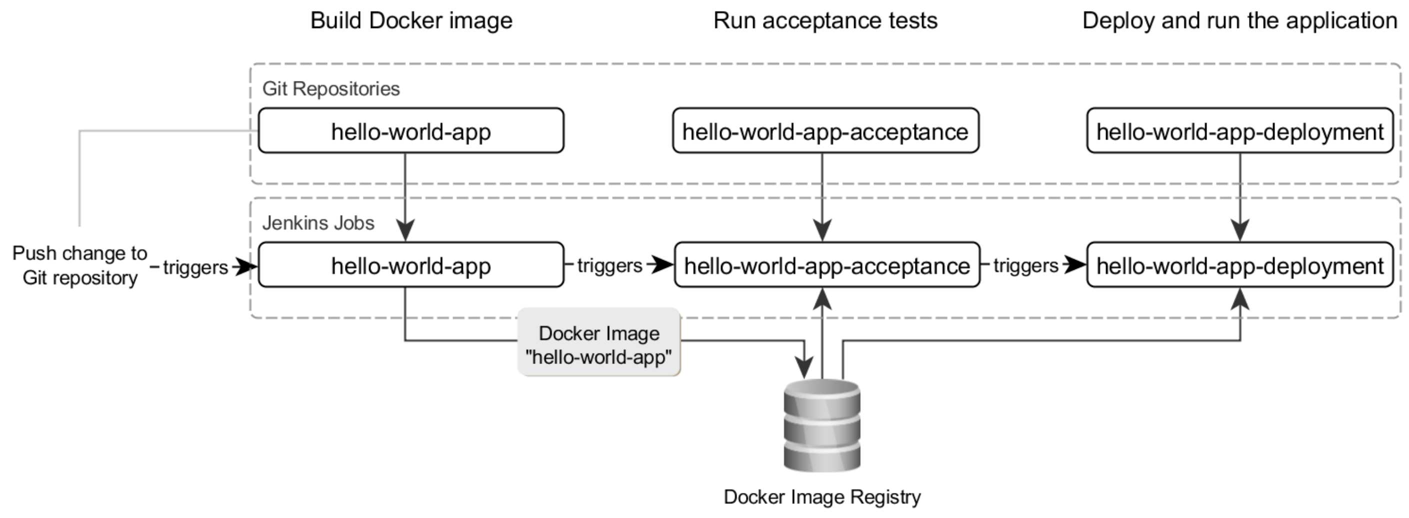
Sources

You can find all sources I used for this tutorial on my Github repository (https://github.com/phauer/continuous-delivery-playground).

The Example Application and the Advantage of Using Runnable Fat Jars

Our example application “hello-world-app” is a simple RESTful microservice (/microservices-nutshell-pros-cons/) created with Dropwizard. Dropwizard let us create a runnable fat jar which already includes an embedded jetty. So we only need to execute the jar in order to start our microservice. This simplifies the necessary infrastructure (no servlet container has to be installed in advance; no deployment of the war into the servlet container) and the deployment process (just copy the jar and execute it). Consequently the architectural decision to use runnable fat jars significantly eases the setup of a Continuous Delivery pipeline (/microservices-nutshell-pros-cons/#Benefits_when_Going_Without_anApplication_Server).

Big Picture


Build Docker image
Run acceptance tests
Deploy and run the application
Git Repositories
hello-world-app
hello-world-app-acceptance
hello-world-app-deployment
Jenkins Jobs
Push change to Git repository
triggers
hello-world-app
triggers
hello-world-app-acceptance
triggers
hello-world-app-deployment
Docker Image "hello-world-app"
Docker Image Registry

(/blog/2015/1107-tutorial-continuous-delivery-with-docker-jenkins/Big-Picture-Example-Continuous-Delivery-Pipeline.png)

Big Picture for our simple Continuous Delivery pipeline

We will need the following projects:

- hello-world-app: creates the Docker image with the microservice jar using Maven (/building-dropwizard-microservice-docker-maven/) and push the image to our Docker Registry.
- hello-world-app-acceptance: runs some tests against the Docker image, which is retrieved from the Docker Registry
- hello-world-app-deployment: runs the Docker image from the Docker Registry

During this tutorial we will set up

- a Git repository (using GitLab) for our sources,
- a Docker Registry to store our built images and
- a Jenkins to create our Continuous Delivery pipeline.

We won't clutter up our local machine by installing these components directly. Instead, we start them as Docker containers. This way, we can easily clean up once we are done. We see that besides using Docker to ship our application itself, it is also a great tool to create infrastructure for build and development.

Warning: Technical Spike

Please note, that this tutorial is a **simplified demonstration** of a Continuous Delivery pipeline. It's just a technical spike. Some important issues (databases, blue/green deployment, canary releasing, security, encryption, IP handling, distribution, communication between the containers, multiple Jenkins nodes) are poorly realized or not covered at all. But consider this spike as a starting point for your own pipeline.

Step by Step

First of all, you need the IP of your host machine (like 192.168.35.217). Therefore, run `ip route` to get the IP of the host machine. Look for the IP after the "src" keyword. You can test the IP by trying to ping the IP within a container by running `docker exec -it bash` (getting bash inside the container) and then `ping <IpOfHostMachine>`

The Commit Stage

Check out my Github repository (https://github.com/phauer/continuous-delivery-playground) using

```
git clone https://github.com/phauer/continuous-delivery-playground.git
cd continuous-delivery-playground
```

Let's start GitLab:

```
./1startGitLab.sh
```

Wait for startup. Create and configure our Git projects:

- Open GitLab on `http://localhost:10080/` , login as root (password: 5iveL!fe), change password to 12345678
- Create the projects 'hello-world-app', 'hello-world-app-acceptance' and 'hello-world-app-deployment'
- We like to notify Jenkins about changes. Therefore go to `http://localhost:10080/root/hello-world-app/hooks` and create a webhook for push events with the URL `http://<IpOfHostMachine>:8090/git/notifyCommit?url=http://<IpOfHostMachine>:10080/root/hello-world-app.git`

Next we commit the projects to our Git repository.

```
./2createProjectsAndCommitToGitLab.sh
```

Next we launch the Docker Registry.

```
./3startDockerRegistry.sh
```

You can see the saved images in the registry by calling `http://localhost:5000/v2/_catalog` .

Configure the Docker daemon to use an unsecure connection for accessing our local Docker Registry. Therefore, add `–insecure-registry <IpOfHostMachine>:5000` to the DOCKER_OPS variable in `/etc/default/docker` .

Now we are ready to start our Continuous Delivery (!) server Jenkins: :-)

```
./4startJenkins.sh
```

Configure Jenkins as follows:

- Installations:
  - Open Jenkins on `http://localhost:8090/configure` and install Maven and JDK if not already done.
  - Install the "GIT Plugin" on `http://localhost:8090/pluginManager/available` if not already done.
- Create the Maven job 'hello-world-app' and configure it as follows:
  - Configure the Git repository in the job: `http://<IpOfHostMachine>:10080/root/hello-world-app.git` . Also configure your GitLab credentials here (root, 12345678).
  - Build > Goals and options: `deploy -Ddocker.registry.name=<IpOfHostMachine>:5000/`
  - Build Triggers > Poll SCM check
- Press 'Build Now'
  - After the build has finished, you should find the created image in your local Docker Registry. Verify this by calling `http://localhost:5000/v2/_catalog` in your browser or take a look into the ~/docker-registry-data folder.

Acceptance Test Stage

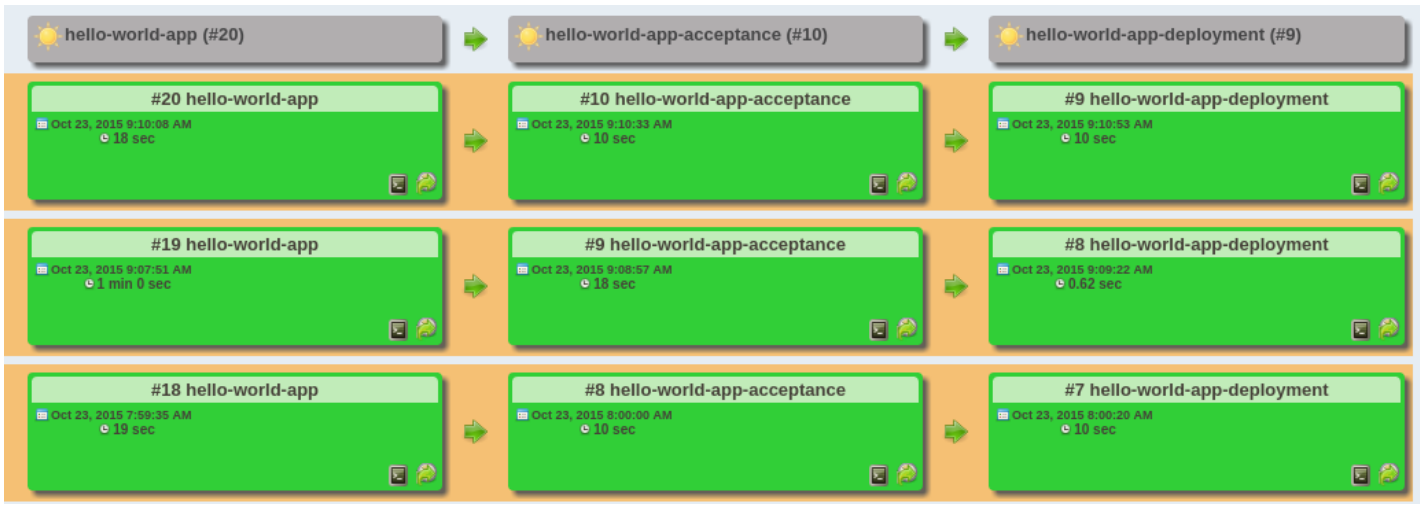
After the Docker image has been created in the Commit Stage, we'll run acceptance tests against the Docker image.

- First of all, install "Build Pipeline Plugin" in Jenkins. The plugin provides an overview over our dependent jobs and our Continuous Delivery pipeline.
- Create Job 'hello-world-app-acceptance' in Jenkins
  - Git URL: `http://<IpOfHostMachine>:10080/root/hello-world-app-acceptance.git`
  - Build > Goals and options: `verify -Ddocker.host.address=<IpOfHostMachine>`
  - Build Triggers > Poll SCM check
  - *Build after other project are built*: hello-world-app
- Create a build pipeline view
  - Create a View and select "Build Pipeline View"
  - Select Initial Job: "hello-world-app"
- Open the created build pipeline view
  - Hit "Run" and you can see your project is going through the build pipeline consisting of 2 stages. Nice! :-)

Deployment Stage

Finally, we want to run our built and tested docker image.

- Create a Jenkins job "hello-world-app-deployment" as a freestyle project
- Git-URL: `http://<IpOfHostMachine>:10080/root/hello-world-app-deployment.git`
- Build after other projects are built: hello-world-app-acceptance
- Add build step: "Execute shell" and insert `./runDockerContainer.sh`



(/blog/2015/1107-tutorial-continuous-delivery-with-docker-jenkins/Jenkins-Build-Pipeline.png)

The Jenkins Plugin 'Build Pipeline' showing our created Continuous Delivery pipeline.

Voila, that's it! We successfully set up a simple Continuous Delivery pipeline. Every time we push a change to our hello-world-app Git project, the application goes through the whole pipeline and is finally deployed in "production". This way we get feedback quickly, increase the reliability of our delivery process and reduce the risk of releasing (due to automation).

Further Readings

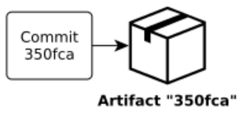
- I highly recommend the book A Practical Guide to Continuous Delivery (https://www.amazon.com/gp/product/B06XBCJHDX/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=B06XBCJHDX&linkCode=as2&tag=blogphilippha-20&linkId=a79fb331219fe4b6d8eabe1516fa0cf3) by Eberhard Wolff.
- Version Numbers for Continuous Delivery with Maven and Docker (/version-numbers-continuous-delivery-maven-docker/)

You can follow me on Twitter (https://twitter.com/philipp_hauer)

Related Posts

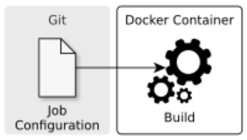
(https://blog.philipphauer.de/building-dropwizard-microservice-docker-maven/)

Building a Dropwizard Microservice with Docker and Maven



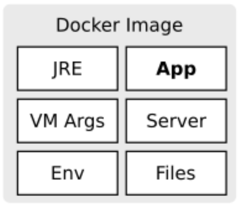
(https://blog.philipphauer.de/version-numbers-continuous-delivery-maven-docker/)

Version Numbers for Continuous Delivery with Maven and Docker



(https://blog.philipphauer.de/improving-continuous-integration-setup-docker-gitlab-ci/)

Improving your Continuous Integration Setup with Docker and GitLab-CI



(https://blog.philipphauer.de/discussing-docker-pros-and-cons/)

Discussing Docker. Pros and Cons.

This entry was posted in Build and Development Infrastructure (/categories/build-and-development-infrastructure/), and tagged with Build (/tags/build/), Continuous Integration and Continuous Delivery (/tags/continuous-integration-and-continuous-delivery/), Docker (/tags/docker/), Dropwizard (/tags/dropwizard/), Git (/tags/git/), Jenkins (/tags/jenkins/), Maven (/tags/maven/), Microservices (/tags/microservices/),

Comments

14 Comments **blog-philipphauer** 1 **Login**

Recommend **Share** **Sort by Best**

Join the discussion…

LOG IN WITH **OR SIGN UP WITH DISQUS** ?

Name

timrsfo • 13 days ago

Great stuff, I'm in the research phase of Jenkins/Docker/Maven CI/CD solutions. This is the second article I've seen referencing the use of Maven/Docker Plugins. At the top of this article you talk about updates specifically Jenkins 2.0 and CI GitLab. I also saw your article on CI-Gitlab Update. However, my Setup is Jenkins.

I've been looking into Jenkins Pipeline tutorials, so now I'm trying to figure out where the Docker portions of the solution are best placed, pom.xml or Jenkinsfile.

My question is this: With jenkins pipeline which parts of this solution would you move to Jenkins Pipeline Jenkinsfile and which would you keep in the pom.xml. I imagine you could write an entire article on this, but in the short term a brief outline would be great - Thanks

• Reply • Share ›

Siddhesh Rele • 2 months ago

https://github.com/siddhesh...

• Reply • Share ›

Marcelo Garcia • a year ago

Excellent tutorial!!

In what container do I need to add –insecure-registry <ipofhostmachine>:5000 to the DOCKER_OPS

variable in /etc/default/docker,
Or do I have to do it in the host?
Thanks in advanced

Reply • Share ›

Cedric Virgil Djahan → Marcelo Garcia • 8 months ago

i think you may do it on the host, because the registry is running on your host machine.

Reply • Share ›

Karthick • 2 years ago

Excellent Tutorial

Reply • Share ›

Manish • 2 years ago

Hi Phil,

I'm getting an error in deployment phase.

+ ./runDockerContainer.sh
docker: error while loading shared libraries: /lib/x86_64-linux-gnu/libapparmor.so.1: cannot read file data: Error 21
docker: error while loading shared libraries: /lib/x86_64-linux-gnu/libapparmor.so.1: cannot read file data: Error 21
Build step 'Execute shell' marked build as failure

Reply • Share ›

Batman → Manish • 2 years ago

I'm getting the same error too :(

Reply • Share ›

René → Batman • 2 years ago

I'm getting the same error too.
What is the purpose of ./runDockerContainer.sh?

Reply • Share ›

Mandar → René • 2 years ago

I found the fix for above error
<>
the fix is to install the missing packages inside the container
apt-get install libsystemd-journal0
apt-get install libapparmor.so.1

Reply • Share ›

Manish → Batman • 2 years ago

This command executes inside jenkins container not on the outside docker host which actually contains the jenkins container. If you run it manually on the docker host it will execute fine. If you "exec" inside the jenkins container and execute any docker command you'll get this error. For more info on the error refer....
https://docs.docker.com/eng...

Reply • Share ›

Da Jo • 2 years ago

hey phil,
at this step: Install the “GIT Plugin” on http://localhost:8090/pluginManager/available if not already done.
There is no option to install git in jenkins. And uploading downloaded git.hpi file does not work.
Any solutions.
servus Jo

Reply • Share ›

Med Da Jo • 2 years ago

Hi Jo,
You need this plugin for the next step.
without this plugin, you can not configure Jenkins to look-up sources on Git.
it's the sources repository.
If you did not like to use Git,
You have to adapte the whole project for other SCM like svn for example.

Reply • Share ›

keith • 2 years ago

Very nice tutorial.

There was an issue with name conflicts here:

```
vagrant@vagrant-ubuntu-trusty-64:/vagrant/continuous-delivery-playground$ docker run --name gitlab -d \
> --link gitlab-postgresql:postgresql --link gitlab-redis:redisio \
> --publish 10022:22 --publish 10080:80 \
> --env 'GITLAB_PORT=10080' --env 'GITLAB_SSH_PORT=10022' \
> --volume /srv/docker/gitlab/gitlab:/home/git/data \
> sameersbn/gitlab:7.14.3
Error response from daemon: Conflict. The name "gitlab" is already in use by container 016e384883e6. You have to delete (or rename) that container to be able to reuse that name.
vagrant@vagrant-ubuntu-trusty-64:/vagrant/continuous-delivery-playground$ docker run --name gitlab2 -d --link gitlab-postgresql:postgresql --link gitlab-redis:redisio --publish 10022:22 --publish 10080:80 --env 'GITLAB_PORT=10080' --env 'GITLAB_SSH_PORT=10022' --volume /srv/docker/gitlab/gitlab:/home/git/data sameersbn/gitlab:7.14.3
64c81b6dfd631ad29255fe2b57dcfb1132c0b55a4053d2d5ef8ccbe67cb35404
```

Reply • Share ›

Med keith • 2 years ago

Hi Philipp,

It's a very ineteresting article,
I folow all the steps and it's work fine :)
I will try to go deep in it

Thanks.

Reply • Share ›

Subscribe Add Disqus to your siteAdd DisqusAdd Disqus' Privacy PolicyPrivacy PolicyPrivacy

Philipp Hauer

I am Philipp Hauer (M.Sc.) and I work as a software engineer and team manager for Spreadshirt (https://www.spreadshirt.com/) in Leipzig, Germany. I focus on developing JVM-based web applications and I'm enthusiastic about Kotlin, clean code, web architectures and microservices. I'm tweeting under @philipp_hauer (https://twitter.com/philipp_hauer).

(mailto:xxblxxogxx@pxxhilippxxhauer.dxxe) (https://github.com/phauer) (https://www.linkedin.com/in/philipp-hauer-677738135) (https://twitter.com/philipp_hauer) (https://www.xing.com/profile/Philipp_Hauer3)

< Discussing Docker. Pros and Cons. (https://blog.philipphauer.de/discussing-docker-pros-and-cons/)
Databases as a Challenge for Continuous Delivery > (https://blog.philipphauer.de/databases-challenge-continuous-delivery/)

Recent Posts

Convincing Your Management to Introduce Kotlin (/convincing-management-introduce-kotlin/)
Self-Contained Systems in Practice with Vaadin and Feeds (/self-contained-systems-vaadin-feeds/)
Smooth Local Development with Docker-Compose, Seeding, Stubs, and Faker (/local-development-docker-compose-seeding-stubs/)
Talk 'Kotlin in Practice' at the JAX 2018 (/talk-jax-kotlin-in-practice-2018/)
Why I Moved Back from Gradle to Maven (/moving-back-from-gradle-to-maven/)

Categories

Web Development (/Categories/Web-Development/) (14)
Build And Development Infrastructure (/Categories/Build-And-Development-Infrastructure/) (13)
Software Craftsmanship (/Categories/Software-Craftsmanship/) (10)
Software Architecture (/Categories/Software-Architecture/) (8)
Database (/Categories/Database/) (7)
Publications And Talks (/Categories/Publications-And-Talks/) (7)
Tools And Environment (/Categories/Tools-And-Environment/) (2)
Management (/Categories/Management/) (1)

Top Tags

Docker (/Tags/Docker/) (9)
Build (/Tags/Build/) (7)
Kotlin (/Tags/Kotlin/) (7)
Rest (/Tags/Rest/) (7)
Best Practices (/Tags/Best-Practices/) (5)
Continuous Integration And Continuous Delivery (/Tags/Continuous-Integration-And-Continuous-Delivery/) (5)
Maven (/Tags/Maven/) (5)
Microservices (/Tags/Microservices/) (5)
Vaadin (/Tags/Vaadin/) (5)
Mongodb (/Tags/Mongodb/) (4)
Test (/Tags/Test/) (4)
Api Design (/Tags/Api-Design/) (3)