## S2I (Source to image)

* A toolkit and workflow for ***building reproducible Docker images*** from source code.
* S2I produces ready-to-run images by injecting source code into a Docker container and letting the container prepare that source code for execution.
* By creating self-assembling builder images, you can version and control your build environments exactly like you use Docker images to version your runtime environments.

## Why do we need it??

* Source to Image (S2I) is a tool for creating Docker images. It is also one of the main ways to build an image on openshift.

***Advantages:***

* *Speed*: S2I can implement very complex operational logic without creating a new mirror layer, so it runs very fast.
* *Patching*: if the image you rely on requires a security patch, S2I allows you to rebuild all the images at once.
* *Efficient*: S2I does not allow arbitrary yum install commands to be run during the build process to prevent slow development iterations
* *Ecology*: S2I encourages a shared mirroring ecosystem. So your application can implement best practices.

## How S2I works

* The implementation philosophy of S2I is: the so-called mirror, which is a running environment + source code.
* If you are a type of source code, if Python, Ruby, HTML, etc., then the runtime environment is basically fixed.
* Therefore, for a certain type of code, as long as a basic image of a standard operating environment is provided.
* When using it, first put the user's code into this image and generate a new image.
* S2I will rely on a special base image, which mainly includes the underlying runtime environment, such as Python, PHP, Nginx, etc. And a few special S2I scripts, like

## Dependencies

1. [docker](https://docker.com/) >= 1.6
2. [Go](https://golang.org/dl/) >= 1.7.1

## Installation

(For Mac users’ easier installation, we recommend you use brew, installation link:

<https://brew.sh/> )

Mac: $ brew install source-to-image

Windows: Download the latest [64-bit Windows release](https://github.com/openshift/source-to-image/releases/latest). Extract the zip file through a file browser. Add the extracted directory to your PATH. You can now use s2i from the command line.

Link: https://github.com/openshift/source-to-image

## Getting Started

You can start using s2i right away (see [releases](https://github.com/openshift/source-to-image/releases)) with the following test sources and publicly available images:

$ s2i build https://github.com/openshift/ruby-hello-world centos/ruby-23-centos7 test-ruby-app

$ docker run --rm -i -p :8080 -t test-ruby-app

## Build an image with S2I

Let's take a quick look at the capabilities of S2I to build a Python application.

First, build a generic Python 2.7 base image first.

**$** git clone https://github.com/sclorg/s2i-python-container.git

**$** cd s2i-python-container/2.7

**$** docker build . -t local/s2i-python27

* This step is the same as the normal Docker image build, and it's no different. You can see by looking at the Dockerfile:
* The parent image of the Dockerfile is “centos/s2i-base-centos74”, which is an S2I base image provided by CentOS
* The same common Python package is installed in this image.
* The only special thing is to copy the “ ./s2i/bin/ directory to the STI\_SCRIPTS\_PATH “ (that is, /usr/libexec/s2i ) location.
* Looking at the files in this directory, you will see the three files run, assemble and usage mentioned above.
* So we get a basic image called “local/s2i-python27:latest”. Next, package the Python source code into the basic image of python.
* **$** cd test/django-test-app
* **$** s2i build test/django-test-app **\**
* local/s2i-python27:latest **\**
* django-test-app:latest
* ---> Installing application source ...
* ---> Installing dependencies ...
* Collecting Django==1.8.1 (from -r requirements.txt (line 1))
* Downloading Django-1.8.1-py2.py3-none-any.whl (6.2MB)
* ...
* Build completed successfully
* In this process, S2I will do the following things
* Start a temporary container with the
* ‘local/s2i-python27:latest’ image
* Put the ‘test/django-test-app’ code in the ‘/tmp/src location’ inside the container
* Here we first convert the source code into a tarball, then pass the tar file into the container through the container's stdout and extract it to the ‘/tmp/ location’.
* Call the assemble script to install the source code
* After the operation is finished, the container will exit.
* After the container exits, submit the container as a mirror
* ‘Docker commit’
* At this time, a name is ‘ django-test-app:latest’ and you can see his history:

**$** docker history django-test-app

IMAGE CREATED BY SIZE

c1775906f248 /bin/sh -c tar -C /tmp -xf - && /usr/libexec… 21.6MB

df3a71bbe741 /bin/sh -c #(nop) CMD ["/bin/sh" "-c" "$STI… 0B

c699d75eb185 /bin/sh -c #(nop) USER 1001 0B

d3fa140aca9e /bin/sh -c source scl\_source enable python27… 18.4MB

...

Finally, start the image verification directly.

**$** docker run -it --rm django-test-app

---> Migrating database ...

Operations to perform:

Synchronize unmigrated apps: staticfiles, messages

Apply all migrations: admin, contenttypes, auth, sessions

Synchronizing apps without migrations:

Creating tables...

Running deferred SQL...

Installing custom SQL...

Running migrations:

Rendering model states... DONE

Applying contenttypes.0001\_initial... OK

Applying auth.0001\_initial... OK

...

Django version 1.8.1, using settings 'project.settings'

Starting development server at http://0.0.0.0:8080/

Quit the server with CONTROL-C.

## Repo2Docker

What is repo2docker??

Repo2docker is to fetch a git repository (eg. GitHub), build a container image based on the specifications found in the repository & optionally launch the container that you can use to explore the repository.

Work flow: fetch github repository 🡪 builf a container image 🡪 launch the container

Usage: Fetches a git repository and builds a container image based on the configuration files found in the repository.

* ***Prerequisites***

1. Docker to build & run the repositories. The [community edition](https://store.docker.com/search?type=edition&offering=community) is recommended.
2. Python 3.4+.
3. Supported on Linux and macOS
4. Very detailed installation doc for Windows users: <https://repo2docker.readthedocs.io/en/latest/install.html#note-about-windows-support>

* ***Installation for Mac Users***

pip install jupyter-repo2docker

To install from source:

git clone https://github.com/jupyter/repo2docker.git

cd repo2docker

pip install -e .

Test and example (Docker needs to run on your pc)

jupyter-repo2docker https://github.com/norvig/pytudes

Github image example we will use for video: <https://github.com/norvig/pytudes>

Pip installation video(Mac): <https://www.youtube.com/watch?v=yBdZZGPpYxg>

**Binderhub:**

**BinderHub** is the server technology that turns computational material into interactive computational environments in the cloud. It utilizes Kubernetes and JupyterHub in order to simplify the deployment process and make it easy to scale.

Steps

**1.** Create Cloud Resources (Google Cloud)

I.Setting up Kubernetes on Google Cloud

II.Install Helm

1.Installation command line: (run it in terminal)

**curl https://raw.githubusercontent.com/kubernetes/helm/master/scripts/get | bash**

2.Initialization (input the command on kubernete cluster)

Set up a ServiceAccount for use by tiller:

**kubectl --namespace kube-system create serviceaccount tiller**

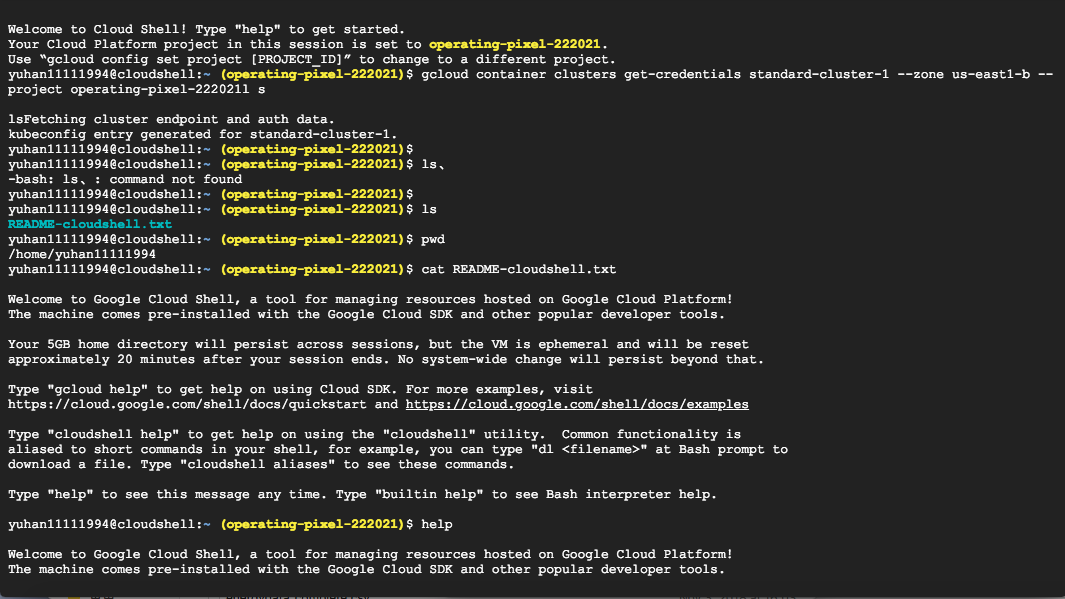
Give the ServiceAccount full permissions to manage the cluster:

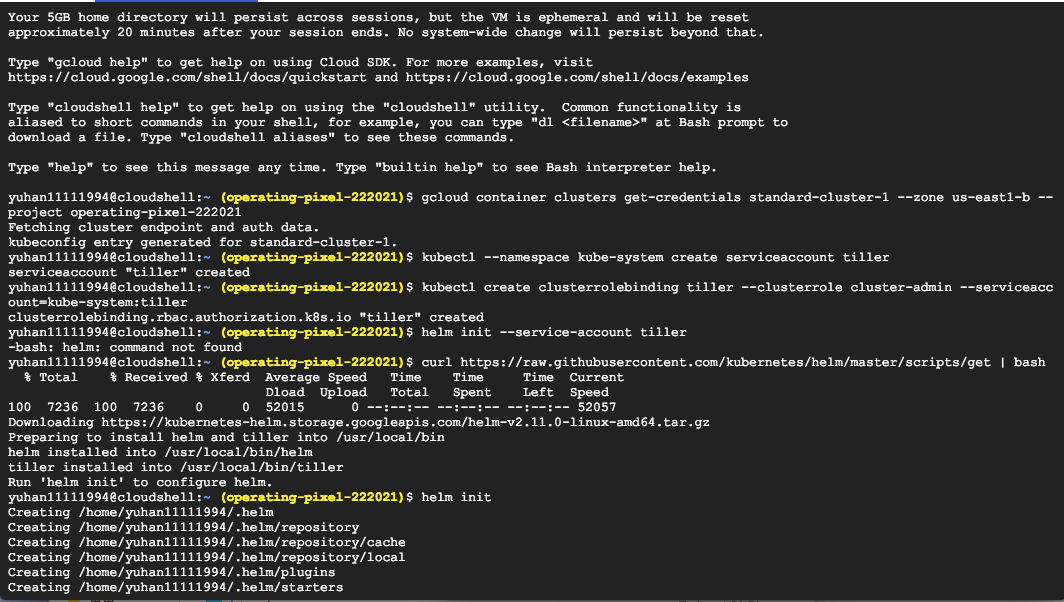
**kubectl create clusterrolebinding tiller --clusterrole cluster-admin --serviceaccount=kube-system:tiller**

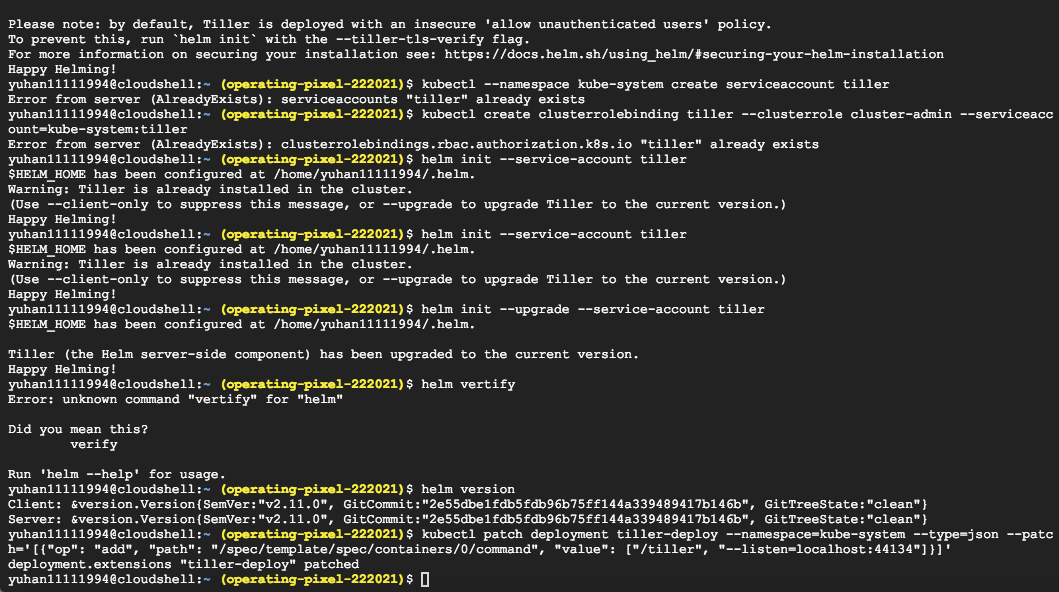
Initialize helm and tiller:

**helm init --service-account tiller**

This is the screen shot of the process:







**2.** set up Google Container Registry:

1.Go to console.cloud.google.com

2.Make sure your project is selected

3.Click **<top-left menu w/ three horizontal bars> -> IAM & Admin -> Service Accounts** menu option

4.Click Create service account

5.Give your account a descriptive name such as “binderhub-builder”

6.Click Role -> Storage -> Storage Admin menu option

7.Check Furnish new private key

8.Leave key type as default of JSON

9.Click Create

**3.** set up BinderHub:

Prepare to install:

**mkdir binderhub**

**cd binderhub**

Create two random tokens by running the following commands then copying the outputs.:

**openssl rand -hex 32**

**openssl rand -hex 32**

Create **secret.yaml** file:

jupyterhub:

hub:

services:

binder:

apiToken: "<output of FIRST `openssl rand -hex 32` command>"

proxy:

secretToken: "<output of SECOND `openssl rand -hex 32` command>"

**Create config.yaml**

Create a file called config.yaml and choose the following directions based on the registry you are using. (**Docker Hub**)

Update config.yaml by entering the following:

config:

BinderHub:

use\_registry: true

image\_prefix: <docker-id|organization-name>/<prefix>-

## Install BinderHub

## First, get the latest helm chart for BinderHub.:

helm repo add jupyterhub https://jupyterhub.github.io/helm-chart

helm repo update

## Next, install the Helm Chart using the configuration files that you’ve just created. Do this by running the following command:

helm install jupyterhub/binderhub --version=0.1.0-... --name=<choose-name> --namespace=<choose-namespace> -f secret.yaml -f config.yaml

Connect BinderHub and JupyterHub：

kubectl --namespace=<namespace-from-above> get svc proxy-public

Copy the IP address under EXTERNAL-IP. This is the IP of your JupyterHub. Now, add the following lines to config.yaml file:

config:

BinderHub:

hub\_url: http://<IP **in** EXTERNAL-IP>

Next, upgrade the helm chart to deploy this change:

helm upgrade <name-from-above> jupyterhub/binderhub --version=v0.1.0-... -f secret.yaml -f config.yaml

## try out your BinderHub Deployment

kubectl --namespace=<namespace-from-above> get svc binder

**4.Tear down the binder**

## First, contracting the size of your cluster then deleting the cluster on google cloud console.