Setting Up Build Pipeline (Continuous Integration)

Overview

This article describes how to set up build pipeline (continuous integration) for mission operations application, open source data visualization widgets or enterprise applications (e.g. Julia, GMAT, etc). For Jenkins configuration or build pipeline setup, please refer to separate article on Jenkins.

Objectives

- This article provides sample instructions how to set up a build pipeline for open source mission operations software.
- This article illustrates how to write unit test for open source mission operations software, and how to automate unit tests.

Use Case Scenarios

- Developers complete code changes for a Web application (e.g. HTML Web pages, JavaScript, NodeJS application), and they want to test if the application runs without errors in a separate test/production-like environment without errors, or impacting production.
- Space engineers write orbit simulation codes and want to test the simulation and to assess the simulation duration in a separate system environment (aka light-weight performance testing).

Assumptions

- Engineers check in source codes to either GitHub (public repositories) or GitLab (private projects).
- Build pipeline is required for quality control and automation since it can streamline and automate the process to test software codes by pulling the latest source codes, compiling/building the binaries, executing automated tests and deploying to the target system environment without manual intervention.

Benefits

- Automated build pipeline will reduce the troubleshooting effort of manual deployment, which may be prone to human errors of missing components, and small system environment changes.
- Reduction of deployment time from hours (complex projects due to dependency deployment and manual testing) to minutes.

- Build pipeline reinforces an operations runbook process, and raises visibility of code quality with auto-deployment after code changes checked in to GitHub.

Example: Auto-deploying Mission Operations Widgets

Variants: Web Application, NodeJS Web server, HTML Web pages

Steps	Purpose	Instructions
	Define and prepare NodeJS Dockerfile container definition for your target machine.	
1	Details In this case, we need to create a NodeJS Web server that includes the following Node.js modules: • Express.js (HTTP Web server) • Mocha.js and Supertest.js (executing JavaScript unit tests) • fs.js (file I/O module to read/write files)	SSH to your docker host devops.audacy.space %cd ~/docker (for first-time, mkdir -p ~/docker) %mkdir -p nodejs %vim Dockerfile (create the instructions how to create NodeJS Web server) %sudo docker build -t rayymlai/nodejs .

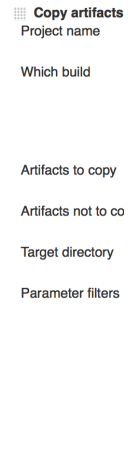
```
FROM cento
MAINTAINER

#Helpful u
RUN yum -y
RUN yum -y
RUN yum -y
RUN yum -y
## for Cen
RUN rpm -i

# Upgradin
RUN yum -y
RUN yum -y
RUN yum -y

#  Downloa
RUN yum -y

RUN yum -y

ADD . /opt

RUN cd /op

EXPOSE 808
```

2	**Create a docker container** for the NodeJS Web server Details • Beware of the access rights of the shared data volume or folder. Don't make it world writable for security reasons.	%sudo docker run --name node02 -v /mnt/data01/jenkins/build/nodejs:/opt/nodejs/missionOps -p 8904:8080 -d rayymlai/nodejs where /mnt/data01/jenkins/build/nodejs is a shared storage volume attached to the docker host (parent Linux machine), and it is mapped to /opt/nodejs/missionOps within the docker container called node02. This NodeJS Web server can be accessible from a Web browser via http://devops.audacy.space:8904/missionOps.
3	Configure Jenkins to **define a build pipeline.** This build process will consist of 3 tasks: • Build • Deploy • Test	Login to http://devops.audacy.space:8906. Define task #1 MIssionOpsApp Build • Install GitHub personal access token • Specify the GIT repository https://github.com/audacyDevOps/sampleMissionOpsApp • Add a post-build action to archive the artifacts (all HTML, JavaScript and test scripts) to a temporary folder • Kick off task #2

Define task #2 MissionOpsApp Deploy

- Copy the artifacts from task #1 to a shared folder accessible by the NodeJS docker container (i.e. deploy the codes)
- Kick off task #3

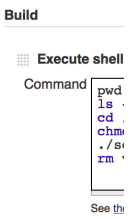
Assumption - NodeJS container should have all required dependencies (e.g. express, mocha, supertest) within the infrastructure stack to run as HTTP Web server.



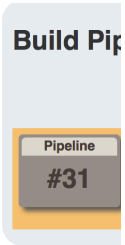
Define task #3 MissionOpsApp Test

- Run automated unit test

This step just invokes test.sh. The unit test /script/test.sh is a driver that wraps a set of JavaScript test scripts under /test.



Create a build pipeline view that kicks off task #1 MissionOpsApp Build, then task #2 and task #3.



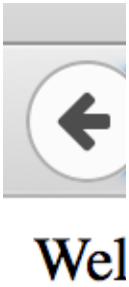
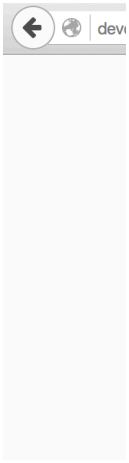
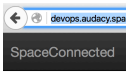
4 **Developers to check in code changes** in GitHub (or GitLab)

Developers write HTML Web pages, JavaScript, CSS or test scripts. They can check in to GitHub (or GitLab), e.g.

%git add *

%git commit -m "bug fix xxx"

%git push -u origin master

5	Optional: verify if Web server is running	Open a Web browser with the URL http://devops.audacy.space:8904/ Since the software codes are not deployed yet, you should only see welcome page.	
		Verify if the software codes are deployed yet by entering a new URL http://devops.audacy.space:8904/missionOps/ Remark: if you have prior deployment, then you should expect the software codes were deployed to the shared folder /mnt/data01/jenkins/build/missionOpsApp. And you should expect a landing Web page from previous deployment.	
6	Observe Build Pipeline view in Jenkins to see if any errors	Go back to Jenkins home page and select the build pipeline, e.g. http://devops.audacy.space:8906/view/Mission%20Operations%20App/	
7	Verify if Web page app / mission operations widgets are deployed successfully.	Open a Web browser with the URL http://devops.audacy.space:8904/missionOps/ You should be able to see the new deployed codes without manual deployment effort.	

Best Practices

- Install all target and minimum NodeJS modules (e.g. express.js, mocha) while you create your NodeJS Dockerfile. Avoid installing Mocha and Supertest (for unit testing) after you deploy a docker container.
- Always create unit tests to verify if your Web server or your docker container is valid.

- Avoid using hard-coded path or folder. Use relative if possible (e.g. Jenkins has PATH variables for build steps).

Example: First Unit Test

To deploy a Web application or Mission Operations Widget (which heavily leverages JavaScript), it is important to verify:

- If the Web server is running, e.g. check for port 8080 is listening, check the default landing page index.html returns any text such as hello world.
- Validate the positive and negative test cases for each function, e.g. if widget should return a canvas of line graph with 10 data items
- Verify the positive and negative test cases for each API, e.g. request(app).get('/api/v1/orbitSimulation')

The following snippet shows a JavaScript unit test example using NodeJS Mocha and Supertest framework, which looks for the file verifyMe.html, and checks if the HTTP Web server is running normally. If the Web server runs normally, it should return "hello world". Please note that the unit test creates a temporary file verifyMe.html on the fly. This file is not bundled with the code base in GitHub or GitLab.

Source Code

```
var request = require('supertest');
var app = require('../app.js');

// dependency: create verifyMe.html
fs = require('fs');
fs.writeFile('verifyMe.html', '<!doctype html>\n<html>\n<h1>hello
world</h1>\n</html>\n',
  function (err) {
   if (err) return console.log(err);
   console.log('Creating verifyMe.html to verify if NodeJS Web server is
working.');
  }
);

// test case 1: test if NodeJS is running
// dependency: caller will create verifyMe.html
//
describe('GET /', function() {
  it('Verify NodeJS Web server with verifyMe.html', function(done) {
    request(app).get('/verifyMe.html').expect('<!doctype
html>\n<html>\n<h1>hello world</h1>\n</html>\n', done);
  });

});
```

Pitfalls

- Make world writable folder shared by all docker containers because shared storage volume with world writable access may be vulnerable to hackers' attacks. Hackers may enumerate all world writable storage volume by bots.
- Store private keys or sensitive credentials in Dockerfile or in the VM.
- Working with relative paths in Jenkins and shared with different docker containers are tricky, and often prone to errors. Always test each scenario. If doubtful, start with explicit path and folder first.