CICD - Working Title

DeDominic, Anthony
Eastern Connecticut State University
Willimantic, USA
dedominica@my.easternct.edu

Abstract

To be written. to see latest revision, please see:

https://github.com/adedomin/cicd-in-dist-env.git

1. Introduction

Content goes here. Example use of cite-proc [1]

2. Background

CICD is the process of accelerating the building, testing and installation of applications to end servers . In a world where web technology moves fast and new features are ideal, it's critical to go to market fast. To do this there are two major architectures that are used. CI, continuous integration is concerned with the development process. In the realm of continuous integration there is usually a source code repository. The source code repository is usually technologies like git, svn, cvs or other version and source controlling software. The idea of the source code repository is to allow for efficient ways to handle code changes and to manage multiple developers working on the same code.

Lastly, there is a server, running software similar to Jenkins CI or Travis CI which can potentially handle many tasks. At a high level, it is usually used to run a build or test step in, say, a build.gradle, project.json, Makefile or various other build tools. To enhance tools like Jenkins, there are binary and other code quality scans like SonarQube and others which attempt to find issues outside of the space of unit and mock tests. Other such triggers, like diff checking, can be used to trigger code reviews for massive revisions of source code.

Generally, the final stage of Continuous Integration is to notify the developers, or other interested parties, the results of these various tests and builds. In a full CICD build pipeline, generally there is the process of uploading so called artifacts to a artifact or binary repository. These binary repositories are more in the realm of continuous delivery.

Continuous delivery also has it's share of tools and structure. One of the first requirements of continuous delivery is having some form of binary repository. For java based projects, there are things like nexus. Other languages, like python, have their own packaging sites; python for instance has PyPI.

In order to deploy these binaries, many tools make use of remote shell protocols. Remote shell protocols give tools access to execute the needed steps to get software deployed. Generally these tools run through scripts which make the machine require the needed binaries and create the needed configurations. The remote shells are no different than conventional user-controlled shells.

There are various tools and servers that do this; a few examples are Ansible, uDeploy and Puppet. Ansible uses remote shell protocols, such as secure shell (ssh), to execute a set of instructions written in yaml, generally with the intent of installing some type of software. uDeploy is similar in how it causes change on target machines, using secure shell; however uDeploy generally requires the target machines to be running some kind of daemon.

Many POSIX-like systems make use of package management to deliver software. Packages, like .deb's and .rpm's, are a collection of installation shell scripts, including pre-installation and post-installation, and a binary or source code. Package managers, as their name implies, also manages these installed binaries, scripts and configurations. Part of their management requires tracking file locations. This allows for rolling back, upgrading or removal of installed files.

Deployment strategy can be different depending on the available infrastructure. In an environment with fixed inventory, it might not be ideal to do things like 'make install' which are hard to reverse. However in an environment where machines can be spun up at whim, or even automatically, it wouldn't matter as much; in such cases one would simply destroy and create a new machine for every deployment. In a fixed architecture, it might be ideal to stick to package management solutions

for delivering binaries. It might not matter if it is dynamic as the cleanliness of the system.

3. Ansible and other Domain Specific Languages

Ansible is one of the newer configuration management tools. One of it's advantages over it's predecessors is that it is agent-less; agent-less as in, it creates self-contained python payloads that is executes on the remote machine. This means less setup as other tools require a running service to manage and trigger jobs. However, the one thing that makes them the same is how they all kind of work.

Ansible, like many of it's competitors: puppet, chef, uDeploy, etc; consume domain specific languages. The domain specific language is then parsed and compiled (or interpreted) into some actionable set of instructions. To make them more extensible, many of these tools allow for creating what are called modules; These modules are written in a much more general programming language and are thus, more expressive, but theoretically harder to write for non-programmers.

Domain specific languages differentiate themselves from general purpose languages in the sense that their scope is significantly limited to a very specific problem. The goal of this is to make it simpler to solve problems in a particular domain by doing less. Generally, domain specific languages make use of simple grammars, like yaml, json, etc; to write in. These languages have mature parsers in most general purpose languages like python, ruby and perl. Thus, instead of writing a whole new language, generally one can fallback on these simpler languages and then generate the code using the structured output of the parsers.

Ansible for instance uses yaml syntax to describe "playbooks;" basically a set of instructions to execute on machines. The yaml is parsed into a simple object that is a collection of key value pairs, where the values can be numbers, strings, boolean, objects or lastly, arrays of all of the other types. From there, the ansible tool creates a python script by using the data contained in the parsed result. Calls to modules and other such things in the parsed structure are replaced with literal python source. Effectively, it's just a translation to library calls.

4. Downsides to Ansible, et al.

One of the biggest issues with these deployment tools, like ansible, is their irreversibility. Changes made to remote machines are not easily tracked and are thus harder to undo.

5. Domain Specific Language for Integration testing, Packaging and Deployment

Below will be the explanation of a new DSL that I will develop that will attempt to solve shortfalls in other tools. Ideally, to also merge them into a universal testing, packaging and deployment framework. The goal of the language is to solve the following.

- Irreversibility of many configuration management tools.
- Resolving build tool, and other environment specific dependencies
- Lack of pull and push centric deployment architectures.
- Clean artifacting of configurations out of binaries/scripts
- Confusing structure and execution flow of deployments

DSL should feature:

- Multi-paradigm for describing deployments
  - graphs (DOT language)
  - sequential steps (yaml | toml | json | etc)
- Dependency management for playbooks & components
  - think npm, pip, etc.
- Integration of all the key steps in a integration and deployment pipeline
  - Integration (testing)
  - Packaging (artifacting)
  - Deployment
- "Agentless" service
  - Utilize builtins only
  - Scripting languages

Citations

[1] M. Fenner, "One-click science marketing," *Nature Materials*, vol. 11, no. 4, pp. 261–263, 2012.