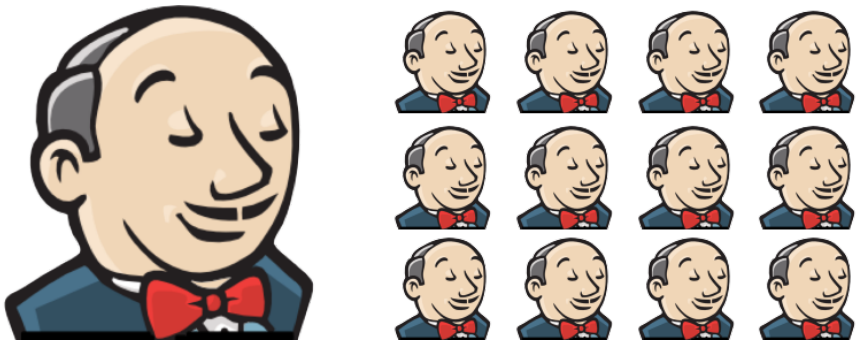
Dermot Burke
May 14 · 9 min read

Breaking the Jenkins Monolith



Much has been written about the advantages of breaking monolithic systems into smaller parts. The freedom to evolve in isolation, the reduced complexity of systems with fewer responsibilities and simplified ownership model are just a few of the compelling reasons why many software teams are moving away from monolithic platforms and codebases towards systems that are composed of smaller isolated parts.

This trend, however, is not unique to software. With the emergence of "Devops", development teams have started to take on more and more operational responsibilities. This, in theory, means that systems with historically centralised control (like CI/CD platforms) can be owned and managed by the teams who use them, rather than some third party with other responsibilities, deadlines and priorities.

The Thoughtworks Tech Radar gives the following advice in this area

> *We're compelled to caution, again, against creating a single CI instance for all teams. While it's a nice idea in theory to consolidate and centralise Continuous Integration (CI) infrastructure, in reality we do not see enough maturity in the tools and products in this space to achieve the desired outcome. Software delivery teams which must use the centralised CI offering regularly have long delays depending on a central team to perform minor configuration tasks, or to troubleshoot problems in the shared infrastructure and tooling. At this stage, we continue to recommend that organisations limit their centralised investment to establishing patterns, guidelines and support for delivery teams to operate their own CI infrastructure.*

> *https://www.thoughtworks.com/radar/techniques/a-single-ci-instance-for-all-teams*

When presented with a monolithic CI platform on a recent client engagement, we were faced with the issues outlined above. The solution we helped implement sought to break down this platform into many small systems. The following is an attempt to explain where we were and where we eventually got to as we built tooling to help solve these problems and attempted to promote a model that gave teams the freedom to evolve in isolation.

Where we were

When we first engaged with the client there was a single Jenkins master for 30 or so different teams. While the setup had originally been configured using Ansible scripts, numerous manual changes had been made over time which meant those scripts were no longer representative of what was actually running. The central team running the instance were thus relying on the daily backups to ensure the system could be restored in the event of a catastrophic failure.

Each team's jobs were configured to run on their own dedicated VMs. Again, while the initial setup of a new VM was done via Ansible scripts, many of the VMs had some manual customisation performed at one time or another in order to overcome some particular issue.

With so many teams relying on a single Jenkins master, the installation of new (or even the upgrade of existing) plugins could be a nerve-wracking process. A central team of ordained individuals would receive requests via Jira tickets for plugin or configuration changes on a regular basis. Once these requests were reviewed, changes would be scheduled for the least disruptive time. This entire process end to end could take days or even weeks.

Once a change was made, there was no way of telling whether or not they would have adverse effects on unsuspecting teams who may not have been aware that a change had just taken place. A job that had been working for months could simply stop working without a reasonable explanation—leaving teams scrambling around trying to figure out what the last change had been and whether or not it was responsible for their latest issue.

When the system did have downtime (often unscheduled) teams would be unable to deploy to testing environments to verify their changes or create new versions of their code for release. This could equate to a lot

of lost time for lots of different teams working on many different projects.

For many of the reasons outlined above, the Jenkins instance itself was still based on an old version without any clear path to upgrade. Key features like Jenkins pipeline could not be introduced as the version of Jenkins was too old to support the necessary plugins. New teams were forced to use an outdated toolset because of the burden of supporting existing teams who were already heavily invested in the platform.

What we needed

Allowing teams to run their own Jenkins masters would not have been difficult in itself. Starting a Jenkins instance is simply a case of running a war file after all. Without the proper tooling, however, there would have been the very real possibility of creating lots of small messes akin to the big mess we already had. Configuring a standard Jenkins instance via the UI yields no record of what changed when, why or by who. Disaster recovery means relying on backups or desperately trying to remember what plugins you had installed and how they were configured.

With the development community already heavily invested in Jenkins, it didn't make much sense to introduce another tool. Though opinions vary, it is hard to deny the power and flexibility that Jenkins provides as a CI platform. With its more recent ability to allow users to define pipelines as code while providing the option to consume shared libraries, it seemed like the best option to fit our needs.

We needed to change the way that people thought about Jenkins. Configuration changes should be traceable over time and teams should have the ability to redeploy or rollback their instance at the click of a button. The instance should have the ability to be completely transient —spun up in a fully configured state and blown away at will.

We had come across a number of Groovy initialisation scripts that were scattered around Github. These range from scripts to set the number of executors through to scripts that can configure a specific plugin. While these scripts were useful, they most often came with no tests or contained hard-coded values. Those of which did consume configuration did not seem to consider versioning in any way— meaning that there was no way to tell which script was compatible with which version of a particular plugin.

What we needed was a simple and consistent framework with which to declare the Jenkins and plugin versions together along with the plugin configuration and jobs. We needed to provide teams with a platform on which to deploy their own instance and the autonomy to manage it over time.

First Steps

The first thing we did was try to get buy-in from the various different parties involved. We realised that creating a solution in isolation and springing it on an unsuspecting development community would not necessarily win their hearts and minds. We understood that if we could encourage people to contribute to the solution then they would be more likely to use it. We held fortnightly get-togethers to discuss the issues people were having and to showcase ideas or our latest developments. While it is true that not everyone wanted to work together, we found that majority of people were willing to try something that ultimately gave them more control and removed bottlenecks. Though these meetings eventually died out, these face to face meetings were pivotal to the early stages of the project, they helped form a community around the project and ensured that what we were building met the demands of a group of people with genuine requirements.

The Nuts and Bolts

After many iterations, the solution that eventually evolved was a framework that would allow a user to define their entire Jenkins configuration in one or more simple configuration files. These files could contain everything from the Jenkins version and list of plugins down to the configuration and credentials required to interact with third-party systems. Each Jenkins Master runs in a container with the config location set as an environment variable. When the container starts, it triggers a script that fetches the configuration and looks for the section which defines where the Jenkins war file should be pulled from.

```
jenkins {
  version {
    artifactPattern = ‘http://ftp-
nyc.osuosl.org/pub/jenkins/war/[revision]/[module].[ext]'
    artifact = “:jenkins:2.112@war”
  }
}
```

Once downloaded, the war file is initialised using the “java -jar” command. During initialisation, the platform makes use of Jenkins

Startup Hooks to execute another groovy script that will download and run whichever version of the scripts are defined in the startupScripts section of the config

```
startupScripts=[
  bintray: [
    artifactPattern: 'https://dl.bintray.com/buildit...',
    artifacts: [“jenkins:jenkins-startup-
scripts:1.10.0@zip”]
  ]
]
```

These scripts are then run one by one, using the configuration values as input. A simple example is the mail configuration script. A sample config for the mail script is as follows

```
mail {
  host = “smtp.somwhere.com”
  defaultSuffix = “@somewhere.com”
}
```

This config is then passed to the script in order that it can set the smtp host and default suffix values.

```
import jenkins.model.*

def inst = Jenkins.getInstance()
def desc = inst.getDescriptor(“hudson.tasks.Mailer”)

desc.setSmtpHost(“${config.host}”)
desc.setDefaultSuffix(“${config.defaultSuffix}”)

desc.save()
```

Each script has its own integration test which starts a Jenkins instance with the necessary plugins, configuration and startup script in place. When Jenkins starts, the script is executed and the correct configuration values can then be asserted in the body of the test.

```
package scripts
```

```
import jenkins.model.Jenkins
import org.junit.BeforeClass
import org.junit.Test
import org.jvnet.hudson.test.recipes.LocalData
import utilities.ZipTestFiles
import static org.hamcrest.CoreMatchers.equalTo
import static org.junit.Assert.assertThat

class MailTest extends StartupTest {

  @BeforeClass
  public static void setUp() {
    setUp(MailTest.class, [“scripts/mail.groovy”])
  }

  @Test
  @LocalData
  @ZipTestFiles(files = [“jenkins.config”])
  void shouldConfigureMailFromConfig() {
    def s =
Jenkins.instance.getDescriptor(“hudson.tasks.Mailer”)
    assertThat(s.smtpHost as String,
equalTo(“smtp.somwhere.com”))
    assertThat(s.defaultSuffix as String,
equalTo(“@somewhere.com”))
  }
}
```

Using this pattern, a script can be written to configure values for any arbitrary Jenkins plugin. While some scripts are simple, writing more complicated scripts can involve reverse engineering plugin source code to figure out how certain values are set. Though this can be a tricky process, the testing framework ensures that there is a reasonably quick feedback loop to keep you on the right track.

One of the scripts provides functionality to generate the complete list of jobs via jobdsl during startup. The following config will create and run a single job called “jobdsl” that will execute any groovy files in the jobdsl directory of the given repository.

```
jobdsl {
  jobdsl=[
    url:"https://github.com/buildit/jenkins-config",
    targets:"jobdsl/*.groovy",
    branch:"*/master"]
}
```

The framework has additional functionality to allow the user to encrypt sensitive information and passwords. There is a web app that can be

used to encrypt text and text files using a secret key. The key must then be passed to the container as an environment variable in order that it can decrypt any encrypted values in the config at runtime. There is also the option of using something like Hashicorp's Vault to store secrets in a central location and access them at runtime.

A fully configured Jenkins instance can thus be initialised using the following command

```
docker run -it -e
JENKINS_CONFIG_FILE=https://github.com/buildit/jenkins-
config.git -e
JENKINS_STARTUP_SECRET=43bc78d572aaa2df61e5cdb5b3725203 -e
JAVA_OPTS=-Djenkins.install.runSetupWizard=false -p
8080:8080 builditdigital/jenkins-image:2.1.0-alpine
```

Deployment Platform

We chose to use Rancher as our deployment platform as it was already being used successfully in other areas for the client. Running a new Jenkins instance was thus as easy as creating a config and using a Rancher Catalogue to fill in the details. A Kubernetes cluster was set up as a build farm on which to run the jobs themselves. We found that the Kubernetes plugin made it simple to spin up docker containers as dynamic slaves. Since many of the jobs relied on the same version of Maven or NPM, users could reuse the same slave image across many jobs. Any team requiring a custom slave type could simply write a new Dockerfile and deploy the image to the repository. As more and more teams came on board it was easy to expand the capacity of the cluster by adding more virtual machines.

Given that the Jenkins masters are redeployed from scratch on a regular basis, any team wishing to store build history had the option of using the Jenkins Splunk plugin with a centrally managed Splunk instance. The Splunk App for Jenkins makes use of the build data to create dashboards and chart patterns over time. Ultimately, storing build logs and history in Splunk makes more sense long term than maintaining a long build history in Jenkins itself.

Where we are

Months after the launch of the platform, it is safe to say that the CI/CD landscape for the client is very different from what we first encountered. We have moved from a monolithic platform which was unstable and resistant to change toward a system that forces teams to

take ownership of their own CI/CD concerns and enables experimentation. When people hit issues, there is a community of teams who are willing to share information and experience. With no central team controlling it and no one to complain about, people are forced to be more proactive when something doesn't work. Whereas before, developers were locked out of picking up administrative tasks, they are now actively encouraged to take on these responsibilities.

The tooling we built to allow people to manage Jenkins has been fundamental to this change. Ensuring people have to store their configuration as code has enforced good hygiene and has in turn given people the freedom to experiment with new plugins and the latest Jenkins versions without the fear of being unable to easily rollback.

The project source code is available on Github here https://github.com/buildit/jenkins-image and a sample configuration here https://github.com/buildit/jenkins-config. I encourage anyone currently using Jenkins with a more traditional approach to try out the framework and perhaps get involved. Whether you have one or one hundred projects, being able to generate and regenerate your CI/CD platform from a single file will prove beneficial in the long run. If you have many teams all relying on a single Jenkins Master, then perhaps let them try to take ownership of their own instance and remove yourself as a bottleneck.