



May 14, 2020

3 Examples of Automation in DevOps

DEVOPS | INFRASTRUCTURE AUTOMATION

By Adam DuVander

DevOps automation can significantly improve productivity and efficiency when delivering software and infrastructure. By automating various tasks and processes, you can free up time and resources that can be better spent on other areas of development. However, implementing automation can be a challenge.

Automation is a key part of DevOps, and there are many ways to automate your workflows. In this post, we'll look at three examples of automation using Jira, Jenkins, and Docker. These examples should give you a good idea of how DevOps automation can be applied to your workflows, no matter what tools you use.

Table of Contents:

- [What Is DevOps Automation?](#)
- [Benefits of DevOps Automation](#)
- [3 Examples of Automation in DevOps](#)
- [Quickly Deploy Event-Driven Workflows](#)
- [Get Started With Puppet Enterprise](#)

What Is DevOps Automation?

DevOps automation relies on infrastructure-as-code tools to automate repetitive and manual tasks. Automation is essential to carrying out the fundamental principles of DevOps, which are centered around developing and deploying software and infrastructure more quickly and reliably.

Related >> [What is DevOps?](#)

Benefits of DevOps Automation

DevOps automation has many benefits, including consistency, efficiency, and speed. Automation streamlines all stages of the development lifecycle for development and operations teams, from deployments to configuration management.

Some of the Benefits of DevOps Automation Include:


- Faster delivery of software and infrastructure
- Greater operational efficiency
- Easier scalability
- Quickly identify and address issues
- Greater transparency in project communication
- Wiser adoption of shared resources
- More room for quality assurance

3 Examples of Automation in DevOps

Example #1: Build Resources (and Transparency) From Work Tickets

Transparency is key to open communication within [DevOps teams](#).

Imagine a world where any project manager can see and build information in a business-friendly context. Taking some time to synchronize your Jenkins pipelines with JIRA, for example, would give a PM real-time insight to EC2 deployment jobs, and prevent hail storms of status pulse checks.

 **Skip to the best part: see automation in action with a free trial of Puppet Enterprise >>**

The following steps will promote Jira from a 2D digital cork board into a mission control interface, giving orders and taking receipts. You could use a similar approach with your project management and build tools.

How To Set Up an Automation Using Jira and Jenkins:

1. Prepare your Jira project to trade information to Jenkins through a webhook. Let's say your project is an app migration and you need to repeatedly provision a suite of AWS services for testing. Instead of manually uploading CloudFormation scripts, add custom fields in the Issue configuration console that will tell Jenkins where to find those scripts and what parameters to provision them with.
2. Create a Build Status field that will be controlled by updates from Jenkins. Verify that those new fields now appear in your tickets.
3. Under the Advanced configuration menu, create a connection with Jenkins by plugging its server into Webhooks configuration form.
4. Select "Created" and "Updated" as Events triggers.
5. Set up Jenkins to receive build parameters and pass build statuses. We assume the pipelines are already built, and we simply need to get them working with the Jira webhook. To do so, install the following plugins and configure them to connect to your Jira account, and more specifically, the project prefix for the app migration board:

- Jira
- Jira Pipeline Steps
- Jira Trigger Plugin
- jira-ext Plugin

These will allow Jenkins to listen for the appropriate updates from Jira, as well as expose functionality to communicate with the triggering ticket from the build.

6. With both applications now locked into each other, update your pipeline to handle the Jira payload and hand it off to the build.
7. Write a JQL filter in the pipeline configuration so it can only be triggered by "Create" and "Update" webhook events.
8. Map the CloudFormation template location and the other parameters as Custom Fields and make sure the Jira ticket key is also captured as an Issue Attribute Path field. The CloudFormation template and parameter are now available as environmental variables to guide that build. Also, the ticket key value is available for the new functions exposed by the Jira plugins to build status updates and log info directly into the ticket for all stakeholders to see.

An automation like this can help you do less repetitive work. It does take some effort to set up and maintain the connections between your tools.

Example #2: Streamline Your Docker Builds With Dynamic Variables

Docker, like CloudFormation, is a powerful tool for rebuilding apps and infrastructure on demand. Here we're going to extend the power of parameterization into Docker-based pipeline builds. Ideally, you could transport a Docker container as-is into any environment. In practice, however, there will be differences. Maybe you have different database endpoints for development, testing, and production. If your application's integration tests rely on hitting the right endpoint per environment, we can use build-specific parameters to keep your Docker containers environment-agnostic.

How To Route Parameters For a Test Environment Build:

```
pipeline {
  // ...
  agent {
    dockerfile {
      filename "my-dockerfile"
      label "my-docker-label"
      args "-v DB_URL= ${env.DB_URL}" // set to
      'test_db.example.com'
    }
  }
  // ...
}
```

Let's assume we're passing a controlled set of parameters from the triggering action into Jenkins, like in the previous example. We're running a test pipeline, so Jenkins received the database endpoints as `DB_URL=test_db.example.com` from the triggering action. We can access this value in our Jenkinsfile from the `env` object.

Now we must funnel the DB_URL value into the Docker build. We begin by declaring a build agent. To maximize the use of your team's Docker program, pull in a custom-built Dockerfile, by declaring your custom Dockerfile as your agent. Next we must pass the environment variable as an interpolated string to the args: `args "-v DB_URL= ${env.DB_URL}"`. In your custom Dockerfile, map the DB_URL ARG instruction to an ENV instruction. Now that URL will be accessible as a variable in your containerized test scripts without having to modify the file at all.

Anything that happens frequently based on events is a good match, so testing is another area ripe for automation.

Example #3: Make Testing Routine Inside of Your Containerized Build

Testing is a non-negotiable pillar of any software development program, but it can become a bottleneck to deployment. DevOps teams often mitigate the logjam with a less-than-comprehensive testing regime. For example, a combination of functional and unit testing on updates to an API may suffice to verify the CRUD functionality with a minimum passable integration to a locally spun-up data source. However, you can replicate all of the APIs' dependencies and run an extensible suite of tests within a one-off docker-compose job with little overhead.

For this example, let's imagine you are building a small Node app that reads and writes to MongoDB. It will be easy to represent the stack in a docker-compose file. Docker-compose files are YAML scripts that tell Docker what to inject into a Dockerfile build. In this case, we're going to build a Node.js image in the Dockerfile and declare two services in our docker-compose.yml: the API service and a MondoDB instance. Save this docker-compose file and a Node.js Dockerfile to build it into an `integration-test` subdirectory in the project repo.

You can now perform some sanity checks by running docker-compose against the `integration-test` directory by running Postman actions against the local API and checking MongoDB on whichever port you exposed it to. More importantly, we're just a few short files away from automatically testing the full integrated operation. Create an `index.js` file with your test scripts. Point your API calls and database operations at the names you gave your services in the docker-compose file and return results within the container. Lastly, create a shell script to manage the docker containers and output the test results. Now you can automatically trigger the tests in other CI/CD pipelines, version control them, and add more services or tests as the business logic evolves.

Quickly Deploy Event-Driven Workflows

As you can see, your workflows are more efficient when you build enough overhead to scale out event-triggered behavior. We've seen how dynamically orchestrating Docker builds from Jira can kick off extensible testing programs within Jenkins. A good optimization principle to consider now is iteration. Some of these integrations may work or falter for your needs. Being able to swap out and remap your services without turning off the pipelines keeps your DevOps workflows flowing.

Get Started With Puppet Enterprise

[Puppet Enterprise](#) is a comprehensive solution for managing and automating event-driven workflows across your organization. With Puppet, you can easily deploy and manage applications, infrastructure, and compliance policies in a way that supports your DevOps goals.

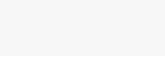
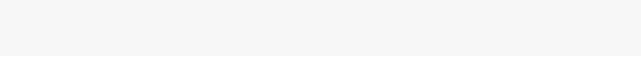
Ready to get started? Sign up for a free trial of Puppet Enterprise today!

START MY TRIAL

Learn More

- Check out this podcast on [why automation is like parenting](#)
- Find out [how to run Puppet in Docker](#)
- Get insights from DevOps engineers on [how to build an awesome open source community](#)

Adam DuVander



ALSO OF INTEREST

DevOps Tools That Can Support Your DevOps Initiatives

What Is DevOps? The History, Definition, Best...

What Is IT Automation? Software, Use Cases &...