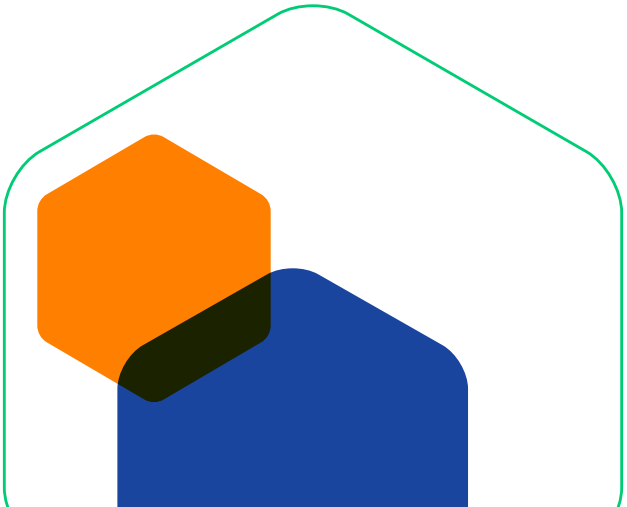
CloudBees.

Jenkins Pipeline with Plugins: Real-World Use Cases for Jenkins Pipeline



Contents

Introduction

Jenkins is one of the preeminent automation tools. Jenkins is extensible by design, using plugins. Plugins are what give Jenkins its great flexibility for automating a wide range of processes on diverse platforms. Jenkins Pipeline builds on that flexibility and rich plugin ecosystem while enabling Jenkins users to write their Jenkins automation as code. This technical guide will show a number of common use cases for plugins with Jenkins Pipeline.

Overview

These use cases include:

- Publishing HTML reports
- Sending notifications
- Continuous delivery using Docker
- Running UI tests in Sauce OnDemand
- Test result interpretation and reporting

Audience

This paper assumes familiarity with the following areas:

- Installing and configuring Jenkins plugins
- Configuring Jenkins jobs via the UI
- Jenkins Pipeline basics
- Groovy language as it applies to Jenkins Pipeline
- Usage of Git

Important Note About Style and Code Samples

All sections are presented in a tutorial style, but are not strictly tutorials.

All the code shown in this guide is real code, mostly Jenkins Pipeline (Groovy) code, but there are also a few uses of JavaScript and Ruby. The code shown was run and it produced the output shown. Effort has been made to clearly show the progressive changes and their results, but your result may look different even if you run the same code. The code shown may not work in your Jenkins instance without modification.

In particular, copying the Jenkins Pipeline code shown directly into a Jenkins Pipeline job via the Jenkins UI will not work. The pipelines shown expect to be run from a Jenkinsfile in the software configuration management (SCM) of the project shown.

Given all that, the projects and code shown are all publicly available. You should feel free to fork a copy and play with them. The Jenkins setup instructions below and at the beginning of each section should provide enough information for you to follow along through each section on your own.

Jenkins Setup

These examples have been verified to work on a specific version of Jenkins and specific versions of various plugins. They should generally continue to work on later versions of Jenkins and the same plugins.

Component	Description
Jenkins	2.19.4 (LTS)
Jenkins Pipeline plugin	2.4
Pipeline Groovy plugin	2.23

Each section will list the plugins required and their minimum version.

Publishing HTML Reports

Most projects need more than just JUnit result reporting. Rather than writing a custom plugin for each type of report, we can use the HTML Publisher plugin.

Introduction

For this first example, we will use a Ruby project called "hermann." We will perform a build of this project using Jenkins Pipeline. We will also have the code coverage results published with each build job. We could write a plugin to publish this data, but the build already creates an HTML report file using SimpleCov when the unit tests run. We'll use the HTML Publisher plugin to add the HTML-formatted code coverage report to each build.

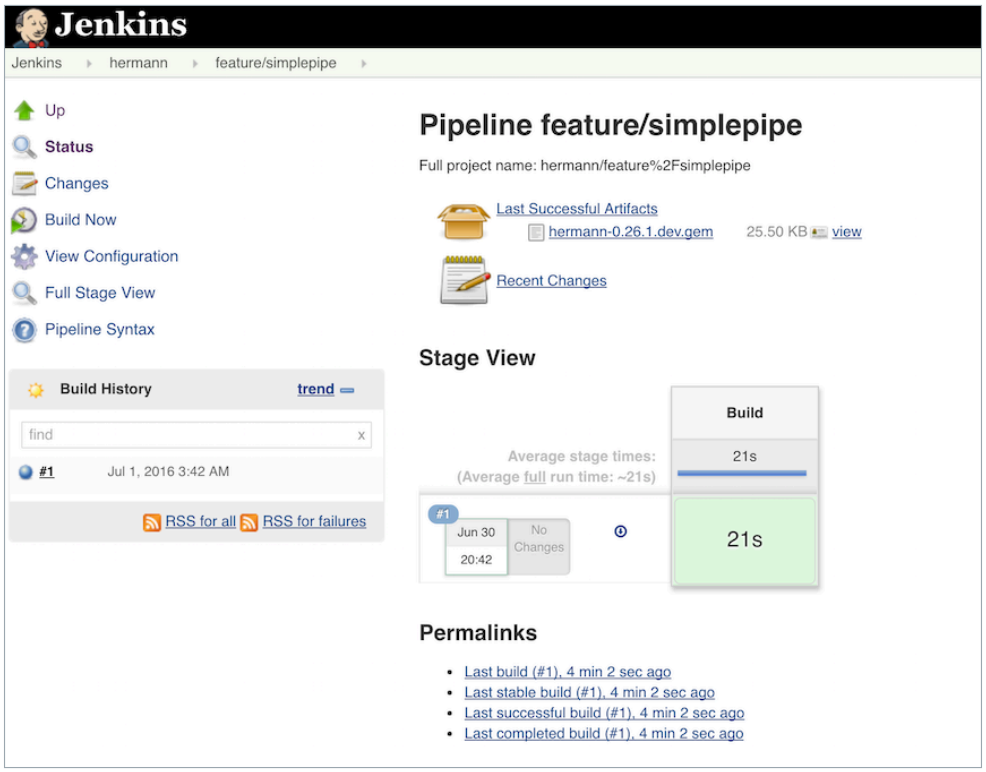
Required plugins

» HTML Publisher plugin (v1.11 or greater)

Setup

Here's a simple pipeline for building the hermann project before we add the report publishing. Simple enough — it builds, runs tests and archives the package:

```
stage ('Build') {
  node {
    // Checkout
    checkout scm

    // install required bundles
    sh 'bundle install'

    // build and run tests with coverage
    sh 'bundle exec rake build spec'

    // Archive the built artifacts
    archive (includes: 'pkg/*.gem')
  }
}
```

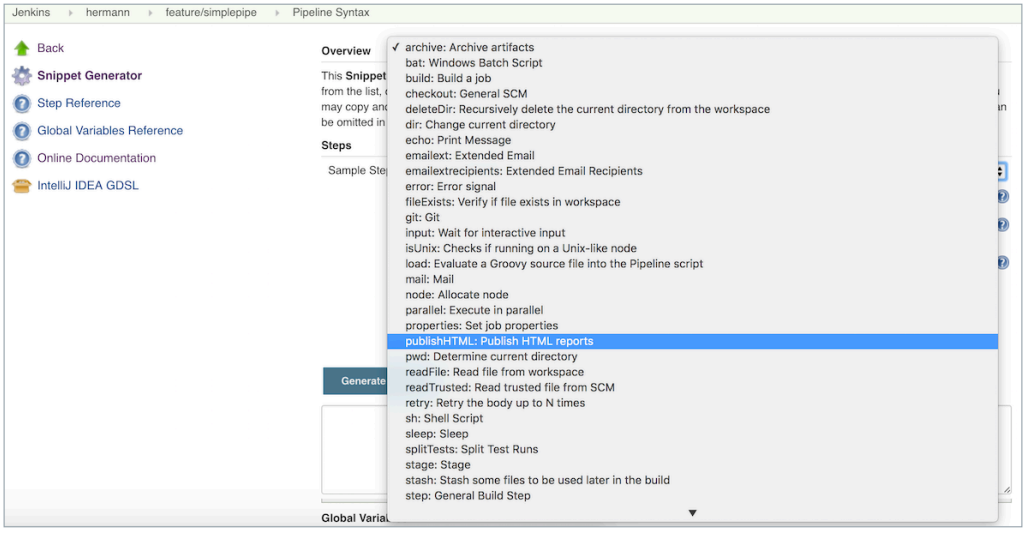


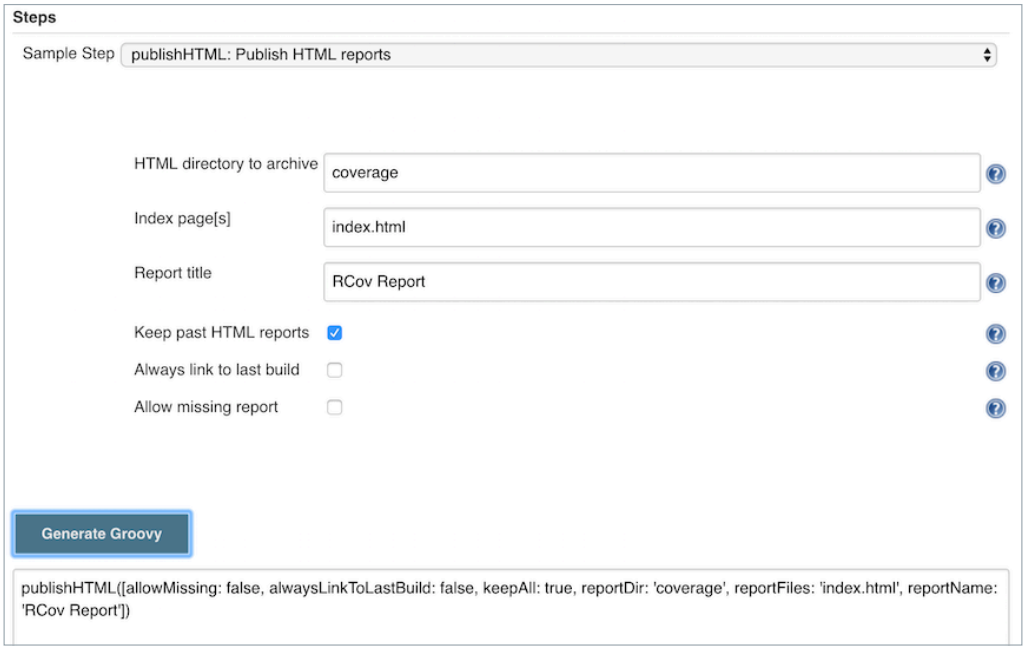
Now we will add the step to publish the code coverage report. Running `rake spec` for this project creates an `index.html` file in the `coverage` directory. The HTML Publisher Plugin is already installed, but how do we add the HTML publishing step to the pipeline? The plugin page doesn't say anything about it.

Snippet Generator

Documentation is hard to maintain and easy to miss, even more so in a system like Jenkins with hundreds of plugins that each potentially have one or more Groovy fixtures to add to the pipeline. The Jenkins Pipeline Snippet Generator helps users navigate this jungle by providing a way to generate a code snippet for any step using provided inputs.

The Snippet Generator offers a dynamically generated list of steps, based on the installed plugins. From that list we select the `publishHTML` step. Then it shows a UI similar to the one used in job configuration. We can fill in the fields, click "generate" and it will show us a snippet of Groovy generated from that input.



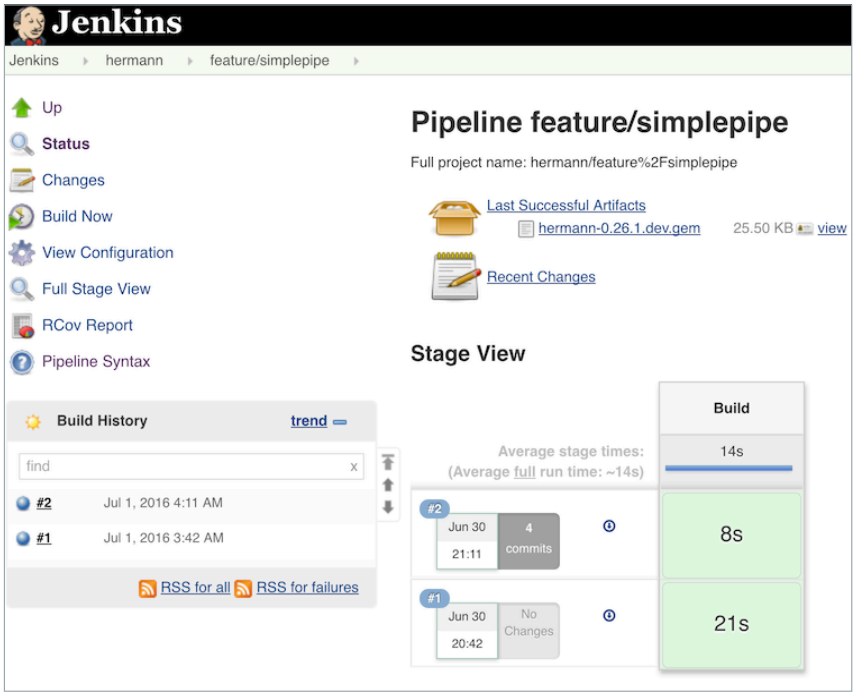


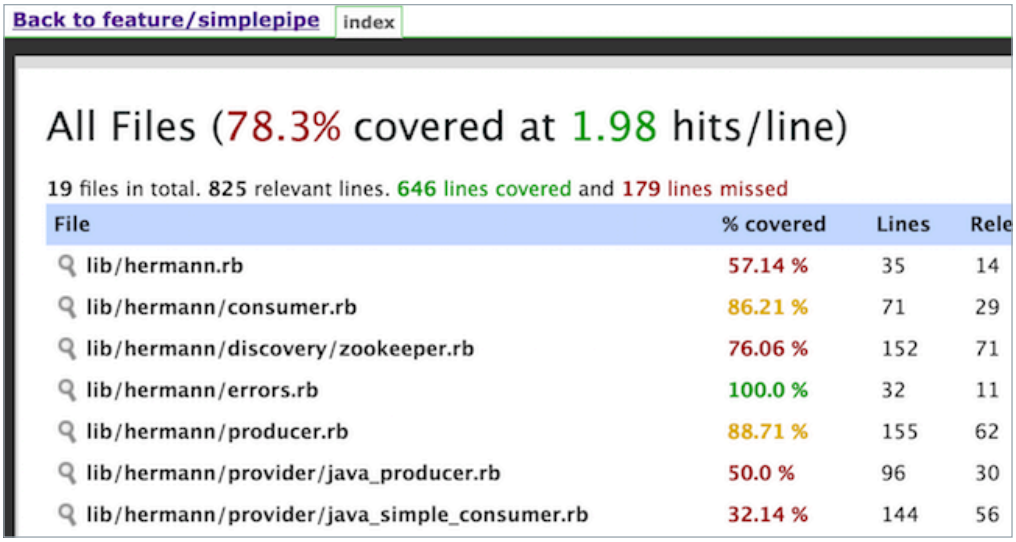
Publishing HTML

We can use that snippet directly or as a basis for further customization. In this case, we'll just reformat and copy it in at the end of the pipeline.

```
stage ('Build') {
  node {
    /* ...unchanged... */

    // Archive the built artifacts
    archive (includes: 'pkg/*.gem')

    // publish html
    publishHTML ([
         allowMissing: false,
         alwaysLinkToLastBuild: false,
         keepAll: true,
         reportDir: 'coverage',
         reportFiles: 'index.html',
         reportName: "RCov Report"
       ])
  }
}
```

Note, we have set `keepAll` to true so we can go back and look at reports on old jobs as new ones come in. When we run this new pipeline, we are rewarded with an `RCov Report` link on the left side, which we can follow to show the HTML code coverage report.





Conclusion

It took a little while to construct it, but that one command is all it takes to publish HTML reports as part of our jobs. Admittedly, manually loading HTML is not as slick as what could be done with a custom plugin, but it is also much easier and works with any static HTML.

Links

- HTML Publisher plugin
- Jenkins Pipeline Snippet Generator

Notifications

Rather than sitting and watching Jenkins for job status, most users would prefer Jenkins to send notifications when events occur. There are Jenkins plugins for sending notifications via Slack, HipChat or even email, among others.

Introduction

Getting notified when events occur is preferable to having to constantly monitor job status just in case something occurs. We will continue from where we left off in the previous section with the hermann project. We added a Jenkins Pipeline with an HTML publisher for code coverage. In this section we'll make Jenkins notify us when builds start and when they succeed or fail.

Required plugins

- Slack plugin (v2.0.1 or greater)
- HipChat plugin (v1.0.0 or greater)
- Email-ext plugin (v2.47 or greater)

Setup and Configuration

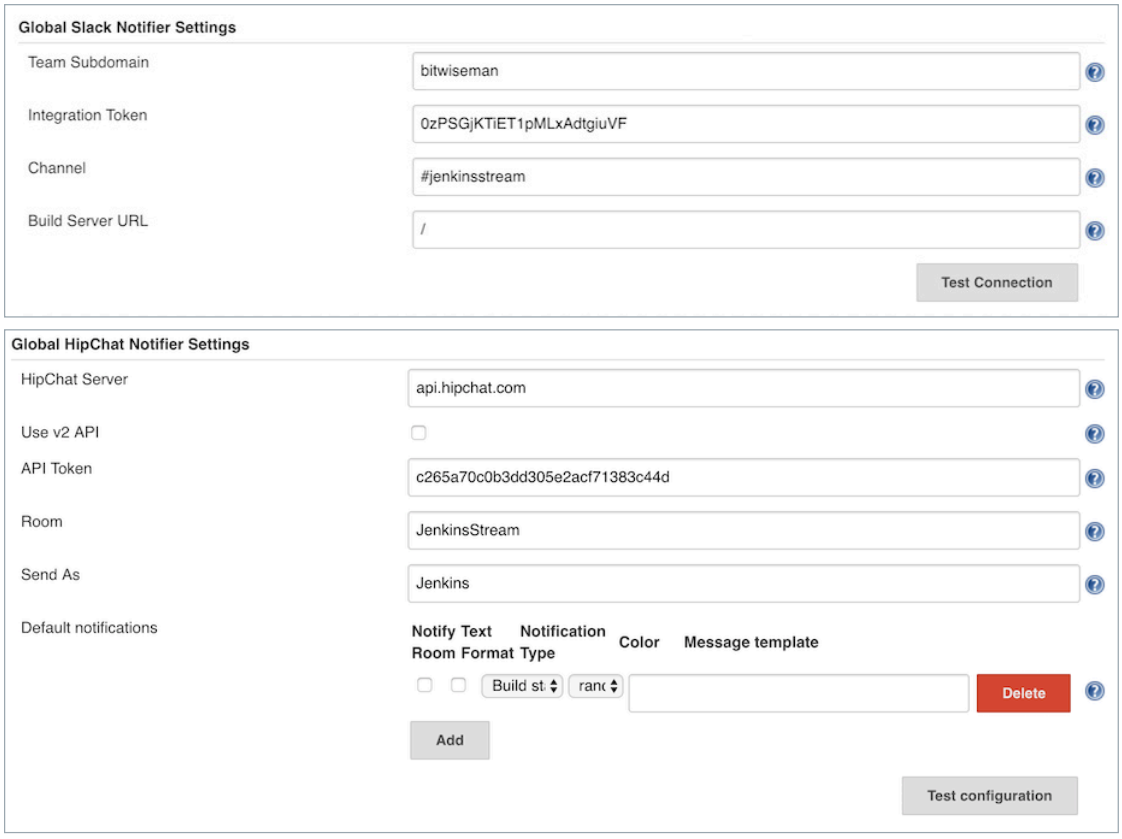
For the rest of this section, we will use sample targets that we created specifically for this purpose. To make this work on your system, you'd need to setup these notifications similar to what we did but using values that match your own instances of these notifications. For example, we created Slack and HipChat organizations called "bitwiseman," each with one member for testing. For email notifications, we ran a Ruby SMTP server called mailcatcher that is perfect for local testing such as this. You'd need to have your own instances of these three types of notification channels.

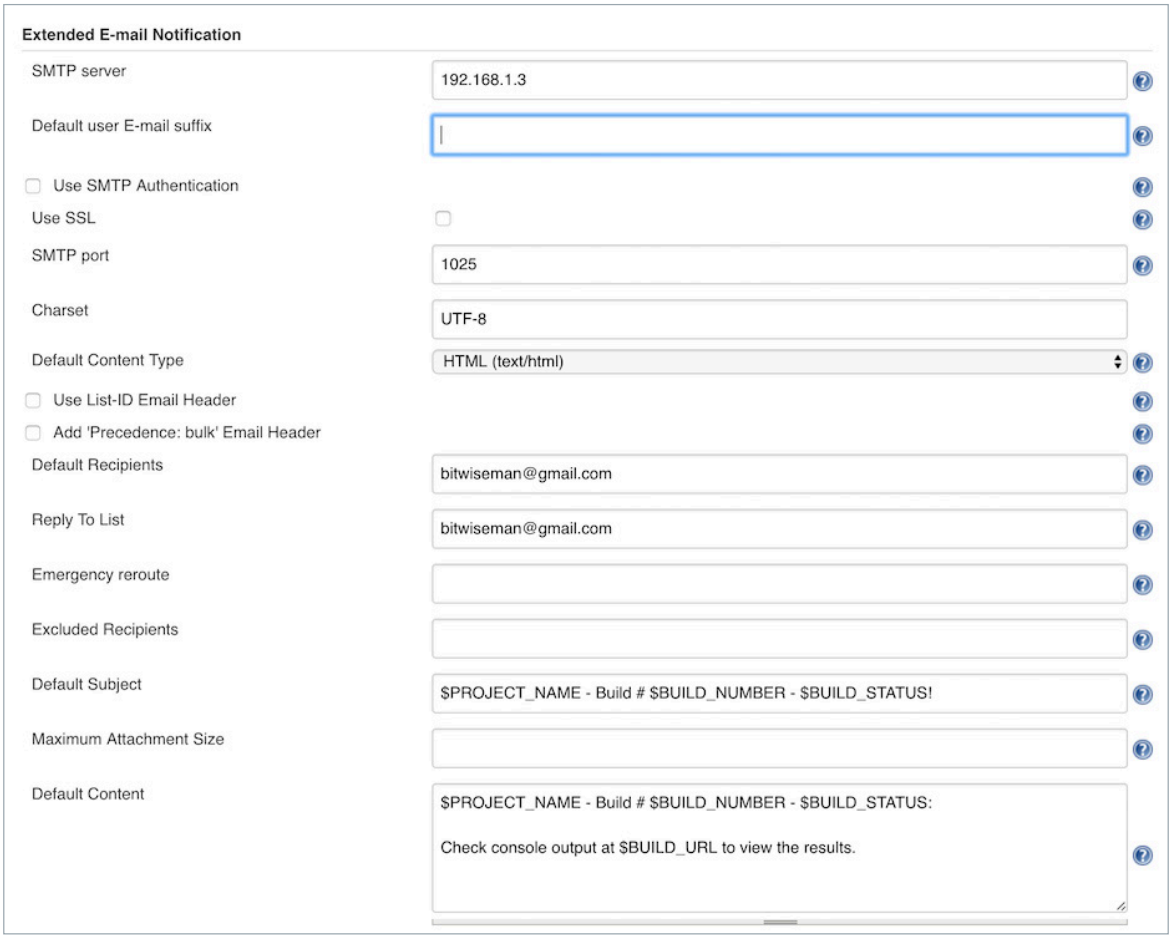
We also installed the Slack, HipChat and Email-ext plugins and added server-wide configuration for each. Slack and HipChat use API tokens - both products have integration points on their side that generate tokens, which we copied into our Jenkins configuration. Mailcatcher SMTP runs locally, so we just pointed Jenkins at it.

Your configuration values will differ from ours, but here's what our Jenkins configuration section for each of these channels looked like:

Original Pipeline

Now we can start adding notification steps to our pipeline. The same as in the previous section, we'll use the





Jenkins Pipeline Snippet Generator to explore the step syntax for the notification plugins. Here's the base pipeline code before we start making changes:

```
stage ('Build') {
  node {
    // Checkout
    checkout scm

    // install required bundles
    sh 'bundle install'

    // build and run tests with coverage
    sh 'bundle exec rake build spec'

    // Archive the built artifacts
    archive (includes: 'pkg/*.gem')
```

```
    // publish html
    publishHTML ([
        allowMissing: false,
        alwaysLinkToLastBuild: false,
        keepAll: true,
        reportDir: 'coverage',
        reportFiles: 'index.html',
        reportName: "RCov Report"
      ])
  }
}
```

Job Started Notification

For the first change, we will add a "Job Started" notification. Using the Snippet Generator and then reformatting make this straightforward:

```
node {
  notifyStarted()

  /* ... existing build steps ... */
}

def notifyStarted() {
  // send to Slack
  slackSend (
    color: '#FFFF00',
    message: "STARTED: Job '${env.JOB_NAME} [${env.BUILD_NUMBER}]'" +
      " (${env.BUILD_URL})")

  // send to HipChat
  hipchatSend (
    color: 'YELLOW',
    notify: true,
    message: "STARTED: Job '${env.JOB_NAME} [${env.BUILD_NUMBER}]'" +
      " (${env.BUILD_URL})")

  // send to email
  emailext (
    subject: "STARTED: Job '${env.JOB_NAME} [${env.BUILD_NUMBER}]'",
    body: """
      <p>STARTED: Job '${env.JOB_NAME} [${env.BUILD_NUMBER}]':</p>
      <p>
```

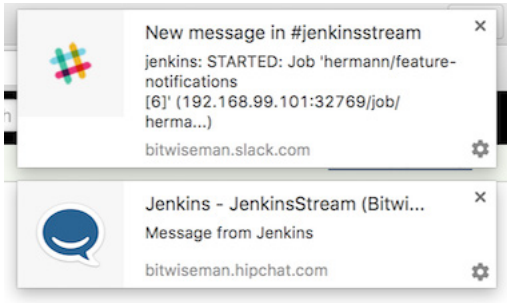
```
        Check console output at
        &QUOT;<a href='${env.BUILD_URL}'>${env.JOB_NAME}
        [${env.BUILD_NUMBER}]</a>&QUOT;
      </p>""",
    recipientProviders: [[$class: 'DevelopersRecipientProvider']])
}
```

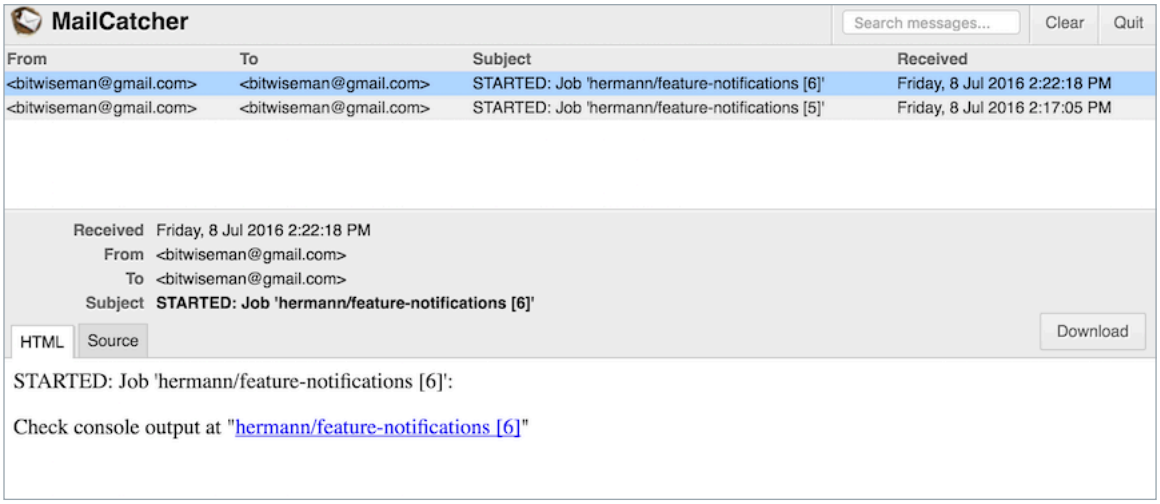
Since Jenkins Pipeline is a Groovy-based DSL, we use string interpolation and variables to add the exact details we want in our notification messages.

When we run this, we'll get the following notifications:

Job Successful Notification

The next logical choice is to receive notifications when a job succeeds. We will copy and paste based on the

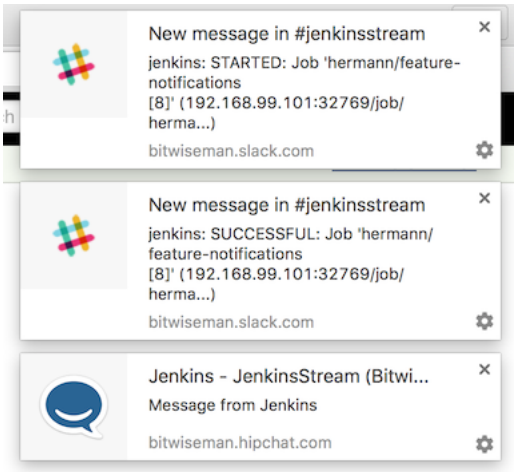




notifyStarted method for now and do some refactoring later.

```
node {

  notifyStarted()

  /* ... existing build steps ... */

  notifySuccessful()
}

def notifyStarted() { /* .. */ }

def notifySuccessful() {
  slackSend (
    color: '#00FF00',
    message: "SUCCESSFUL: Job '${env.JOB_NAME} [${env.BUILD_NUMBER}]'" +
      " (${env.BUILD_URL})")

  hipchatSend (
    color: 'GREEN',
    notify: true,
    message: "SUCCESSFUL: Job '${env.JOB_NAME} [${env.BUILD_NUMBER}]'" +
      " (${env.BUILD_URL})")

  emailext (
    subject: "SUCCESSFUL: Job '${env.JOB_NAME} [${env.BUILD_NUMBER}]'",
    body: """
      <p>SUCCESSFUL: Job '${env.JOB_NAME} [${env.BUILD_NUMBER}]':</p>
      <p>
        Check console output at
        &QUOT;<a href='${env.BUILD_URL}'>${env.JOB_NAME}
        [${env.BUILD_NUMBER}]</a>&QUOT;
      </p>""",
    recipientProviders: [[$class: 'DevelopersRecipientProvider']])
}
```

Again, we get notifications, as expected. If this build is fast enough, some of them may even be on the screen at the same time:

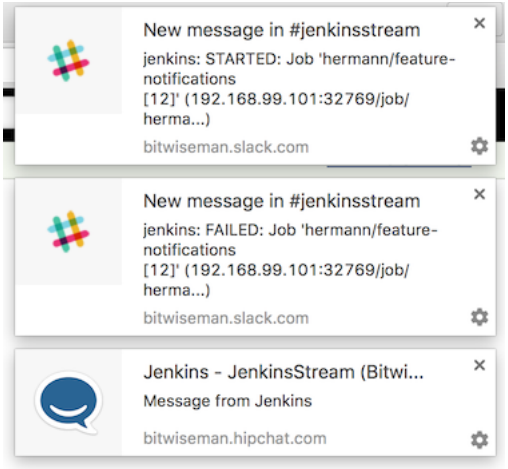


Job Failed Notification

Next we'll add failure notification. Here is where we really start to see the power and expressiveness of Jenkins Pipeline. A pipeline is a Groovy script, so as we would expect in any Groovy script, we can handle errors using `try-catch` blocks.

```
node {
  try {
    notifyStarted()

    /* ... existing build steps ... */

    notifySuccessful()
  } catch (e) {
    currentBuild.result = "FAILED"
    notifyFailed()
    throw e
  }
}

def notifyStarted() { /* .. */ }

def notifySuccessful() { /* .. */ }

def notifyFailed() {
  slackSend (
    color: '#FF0000',
    message: "FAILED: Job '${env.JOB_NAME} [${env.BUILD_NUMBER}]'" +
      " (${env.BUILD_URL})")
```

```
    hipchatSend (
      color: 'RED',
      notify: true,
      message: "FAILED: Job '${env.JOB_NAME} [${env.BUILD_NUMBER}]'" +
        " (${env.BUILD_URL})")

    emailext (
      subject: "FAILED: Job '${env.JOB_NAME} [${env.BUILD_NUMBER}]'",
      body: """
        <p>FAILED: Job '${env.JOB_NAME} [${env.BUILD_NUMBER}]':</p>
        <p>
          Check console output at
          &QUOT;<a href='${env.BUILD_URL}'>${env.JOB_NAME}
          [${env.BUILD_NUMBER}]</a>&QUOT;
        </p>""",
      recipientProviders: [[$class: 'DevelopersRecipientProvider']])
}
```



Code Cleanup

Lastly, now that we have it all working, we can do some refactoring. Let's unify all the notifications in one method and move the final success/failure notification into a `finally` block.

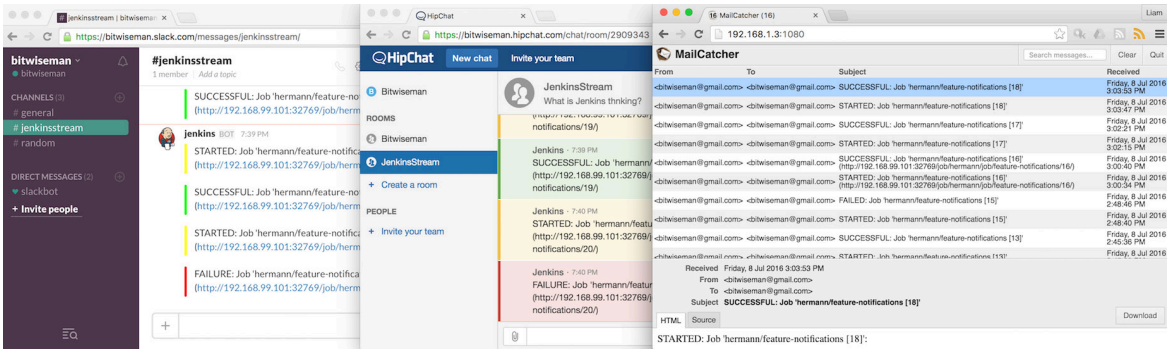
```
stage ('Build') {
  node {
    try {
      notifyBuild('STARTED')

      /* ... existing build steps ... */
```

```
  } catch (e) {
    // If there was an exception thrown, the build failed
    currentBuild.result = "FAILED"
    throw e
  } finally {
    // Success or failure, always send notifications
    notifyBuild(currentBuild.result)
  }
}

def notifyBuild(String buildStatus = 'STARTED') {
  // build status of null means successful
  buildStatus =  buildStatus ?: 'SUCCESSFUL'

  // Default values
  def colorName = 'RED'
  def colorCode = '#FF0000'
  def subject =
    "${buildStatus}: Job '${env.JOB_NAME} [${env.BUILD_NUMBER}]'"
  def summary = "${subject} (${env.BUILD_URL})"
  def details = """
    <p>${buildStatus}: Job '${env.JOB_NAME} [${env.BUILD_NUMBER}]':</p>
    <p>
      Check console output at
      &QUOT;<a href='${env.BUILD_URL}'>${env.JOB_NAME}
      [${env.BUILD_NUMBER}]</a>&QUOT;
    </p>"""

  // Override default values based on build status
  if (buildStatus == 'STARTED') {
    color = 'YELLOW'
    colorCode = '#FFFF00'
  } else if (buildStatus == 'SUCCESSFUL') {
    color = 'GREEN'
    colorCode = '#00FF00'
  } else {
    color = 'RED'
    colorCode = '#FF0000'
  }

  // Send notifications
  slackSend (color: colorCode, message: summary)
```

```
    hipchatSend (color: color, notify: true, message: summary)

    emailext (
      subject: subject,
      body: details,
      recipientProviders: [[$class: 'DevelopersRecipientProvider']])
  }
}
```

Conclusion

We now get notified twice per build, on three different channels. This is probably more than anyone needs, especially for such a short build. However, for a longer or complex CD pipeline, we might want exactly this. If needed, we could even improve the `notifyBuild` code to handle other status strings and call it as needed throughout our pipeline.



Links

- Slack plugin
- HipChat plugin
- Email-ext plugin
- Jenkins Pipeline Snippet Generator

Continuous Delivery

Introduction

When the Ruby on Rails framework debuted, it changed the industry in two noteworthy ways: it created a trend towards opinionated web application frameworks (Django, Play, Grails) and it *strongly* encouraged thousands of developers to embrace test-driven development, along with many other modern best practices (source control, dependency management, etc). Because Ruby, the language underneath Rails, is interpreted instead of compiled there isn't a build, per se, but rather tens - if not hundreds - of tests, linters and scans which are run to ensure the application's quality. With the rise in popularity of Rails, the popularity of application hosting services with easy-to-use deployment tools like Heroku or Engine Yard has risen, too. This combination of good test coverage and easily automated deployments makes Rails easy to continuously deliver with Jenkins. In this section we'll cover testing non-trivial Rails applications with Jenkins Pipeline and, as an added bonus, we will add security scanning via Brakeman and the Brakeman plugin.

For this section, we'll use Ruby Central's cfp-app:

A Ruby on Rails application that lets you manage your conference's call for proposal (CFP), program and schedule. It was written by Ruby Central to run the CFPs for RailsConf and RubyConf.

This Rails app is not only a sizable application with lots of tests, but it's actually the application the Jenkins project used to collect talk proposals for the Community Tracks for Jenkins World 2016. For the most part, cfp-app is a standard Rails application. It uses PostgreSQL for its database, RSpec for its tests and Ruby 2.3.x as its runtime.

Required plugins

- Brakeman plugin (v0.8 or greater)
- CloudBees Docker Pipeline plugin (v1.9 or greater)

Preparing the App

For *most* Rails applications there are few, if any, changes needed to enable continuous delivery with Jenkins. In the case of cfp-app, we'll add two gems to get the most optimal integration into Jenkins:

1. ci_reporter, for test report integration
2. brakeman, for security scanning

Adding these is simple, we'll just update the `Gemfile` and the `Rakefile` in the root of the repository to contain:

Gemfile

```
# .. snip ..
group :test do
  # RSpec, etc
  gem 'ci_reporter'
  gem 'ci_reporter_rspec'
  gem "brakeman", :require => false
end
```

Rakefile

```
# .. snip ..
require 'ci/reporter/rake/rspec'
# Make sure we setup ci_reporter before executing our RSpec examples
task :spec => 'ci:setup:rspec'
```

Preparing Jenkins

With the cfp-app project set up, next we'll ensure that Jenkins itself is ready with the following plugins installed:

- Brakeman plugin
- CloudBees Docker Pipeline plugin

In addition to the plugins listed above, we also need at least *one* Jenkins agent with the Docker daemon installed and running on it, with the agent labeled "docker" to let us assign Docker-based workloads to them.

Writing the Pipeline

To make sense of the various things that the `Jenkinsfile` needs to do, we'll start by simply defining the stages of our pipeline. This will help us think, in broad terms, of what order of operations our pipeline should have. For example:

```
/* Assign our work to an agent labelled 'docker' */
node('docker') {
  stage 'Prepare Container'
  stage 'Install Gems'
  stage 'Prepare Database'
  stage 'Invoke Rake'
  stage 'Security scan'
  stage 'Deploy'
}
```

As mentioned previously, this `Jenkinsfile` will rely on the CloudBees Docker Pipeline plugin. The plugin provides two very important features:

1. Ability to execute steps *inside* of a running Docker container
2. Ability to run a container in the background

Like most Rails applications, one can effectively test the application with two commands: `bundle install` followed by `bundle exec rake`. We already have Docker images prepared with RVM and Ruby 2.3.0 installed, which ensures a common and consistent starting point:

```
node('docker') {
  // .. 'stage' steps removed
  docker.image('rtyler/rvm:2.3.0').inside { // <1>
    rvm 'bundle install'  // <2>
    rvm 'bundle exec rake'
  }  // <3>
}
```

Notes:

1. Run the named container. The inside method can take optional additional flags for the Docker run command.
2. Execute our shell commands using our tiny sh step wrapper rvm. This ensures that the shell code is executed in the correct RVM environment.
3. When the closure completes, the container will be destroyed.

Unfortunately, with this application, the bundle exec rake command will fail if PostgreSQL isn't available when the process starts. This is where the *second* important feature of the CloudBees Docker Pipeline plugin comes into effect: the ability to run a container in the background.

```
node('docker') {
  // .. 'stage' steps removed
  /* Pull the latest `postgres` container and run it in the background */
  docker.image('postgres').withRun { container ->  // <1>
    echo "PostgreSQL running in container ${container.id}" // <2>
  } // <3>
}
```

Notes:

1. Run the container, effectively "docker run postgres"
2. Any number of steps can go inside the closure
3. When the closure completes, the container will be destroyed

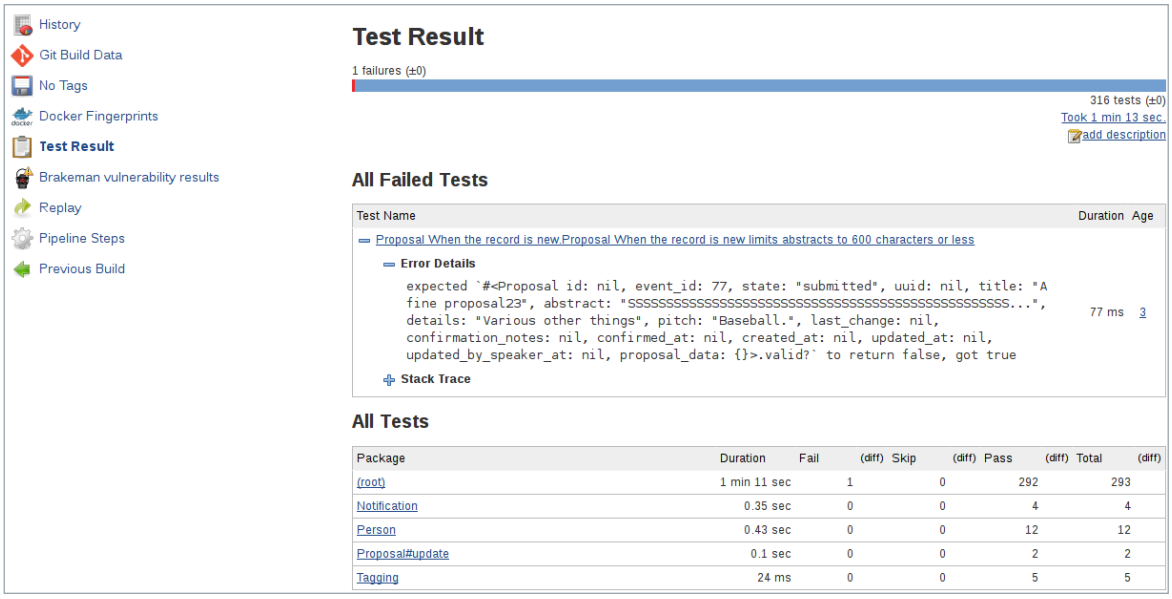
Running the Tests

Combining these two snippets of Jenkins Pipeline code highlights where the power of the DSL shines. With this done, the basics are in place to consistently run the tests for cfp-app in fresh Docker containers for each execution of the pipeline.

```
node('docker') {
  docker.image('postgres').withRun { container ->
      docker.image('rtyler/rvm:2.3.0')
          .inside("--link=${container.id}:postgres") { // <1>
        stage ('Install Gems') {
          rvm "bundle install"
        }

        stage ('Invoke Rake'){
          withEnv(
                ['DATABASE_URL=postgres://postgres@postgres:5432/']
              ) { // <2>
            rvm "bundle exec rake"
          junit 'spec/reports/*.xml' // <3>
        }
      }
    }
  }
}
```

Notes:

1. By passing the "--link" argument, the Docker daemon will allow the RVM container to talk to the PostgreSQL container under the host name postgres.
2. Use the withEnv step to set environment variables for everything that is in the closure. In this case, the cfp-app DB scaffolding will look for the DATABASE_URL variable to override the DB host/user/dbname defaults.
3. Archive the test reports generated by ci_reporter so that Jenkins can display test reports and trend analysis.



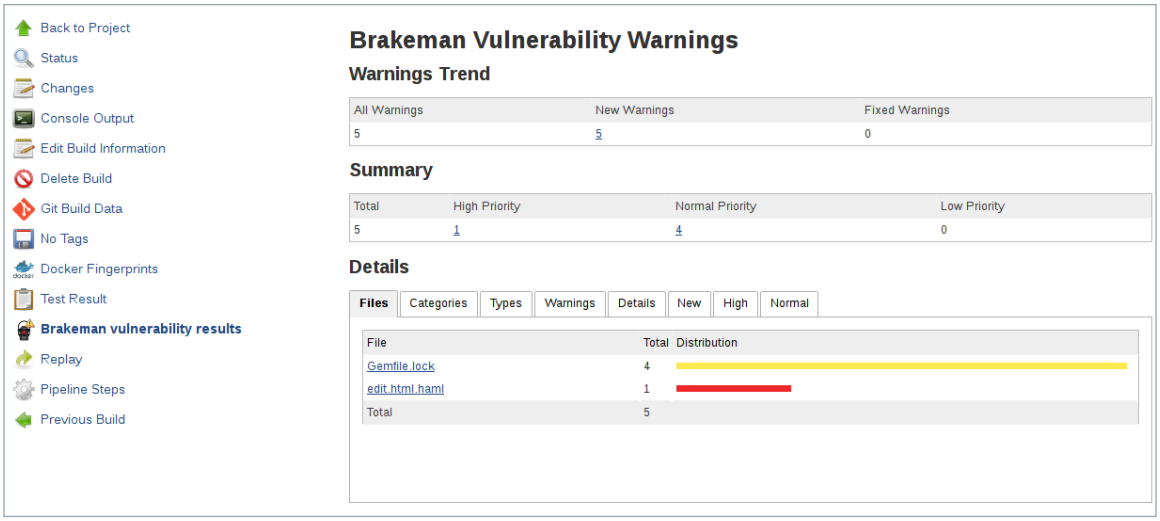
Security Scanning

Using Brakeman, the security scanner for Ruby on Rails, is almost trivially easy inside of Jenkins Pipeline, thanks to the Brakeman plugin which implements the `publishBrakeman` step. Building on our example above, we can implement the "Security scan" stage:

```
node('docker') {
  /* --8<--8<-- snipsnip --8<--8<-- */
  stage('Security scan') {
    rvm 'brakeman -o brakeman-output.tabs' +
     ' --no-progress --separate-models' // <1>
    publishBrakeman 'brakeman-output.tabs' // <2>
    /* --8<--8<-- snipsnip --8<--8<-- */
  }
}
```

Notes:

1. Run the Brakeman security scanner for Rails and store the output for later in brakeman-output.tabs
2. Archive the reports generated by Brakeman so that Jenkins can display detailed reports with trend analysis



Deploying

Once the tests and security scanning are all working properly, we can start to set up the deployment stage. Jenkins Pipeline provides the variable currentBuild, which we can use to determine whether our pipeline has been successful thus far or not. This allows us to add the logic to only deploy when everything is passing:

```
node('docker') {
  /* --8<--8<-- snipsnip --8<--8<-- */
  stage('Deploy') {
    if (currentBuild.result == 'SUCCESS') {   // <1>
        sh './deploy.sh' // <2>
    }
    else {
      mail (
        subject: "Something is wrong with " +
          "${env.JOB_NAME} ${env.BUILD_ID}",
        to: 'nobody@example.com',
        body: 'You should fix it')
    }
    /* --8<--8<-- snipsnip --8<--8<-- */
  }
}
```

Notes:

1. currentBuild has the result property which would be 'SUCCESS,' 'FAILED,' 'UNSTABLE,' 'ABORTED'
2. Only if currentBuild.result is successful should we bother invoking our deployment script (e.g. git push heroku master)

Conclusion

Here is a thoroughly commented full Jenkinsfile, which we hope is a useful summation of the example outlined above. The consistency provided by Docker and Jenkins Pipeline above shows how Pipeline can improve project delivery time. There is still room for improvement however, which is left as an exercise for the reader. For example, preparing new containers with all their dependencies built-in, instead of installing them at run-time, or utilizing the parallel step for executing RSpec across multiple Jenkins agents simultaneously.

The beautiful thing about defining your continuous delivery (and continuous security) pipeline in code is that you can continue to iterate on it!

Sauce On-Demand for UI Testing

Introduction

Testing web applications across multiple browsers on different platforms can be challenging even for smaller applications. With Jenkins and the Sauce OnDemand plugin, you can wrangle that complexity by defining your Pipeline as Code. For this section we'll use the Sauce OnDemand plugin and Nightwatch.js to run Selenium tests on a sample project.

Required plugins

- JUnit plugin (v1.19 or greater)
- Sauce OnDemand plugin (v1.159 or greater)

Starting from Framework

We will start off by following Sauce Labs' instructions on "Setting up Sauce Labs with Jenkins." We'll install the JUnit and Sauce OnDemand plugins, create an account with Sauce Labs and add our Sauce Labs credentials to Jenkins.

Next, let's use one of the sample projects in "saucelabs-sample-test-frameworks" on GitHub, which demonstrates how to integrate Sauce Labs with various test frameworks. For this section, we'll use a JavaScript-based

framework called Nightwatch.js.

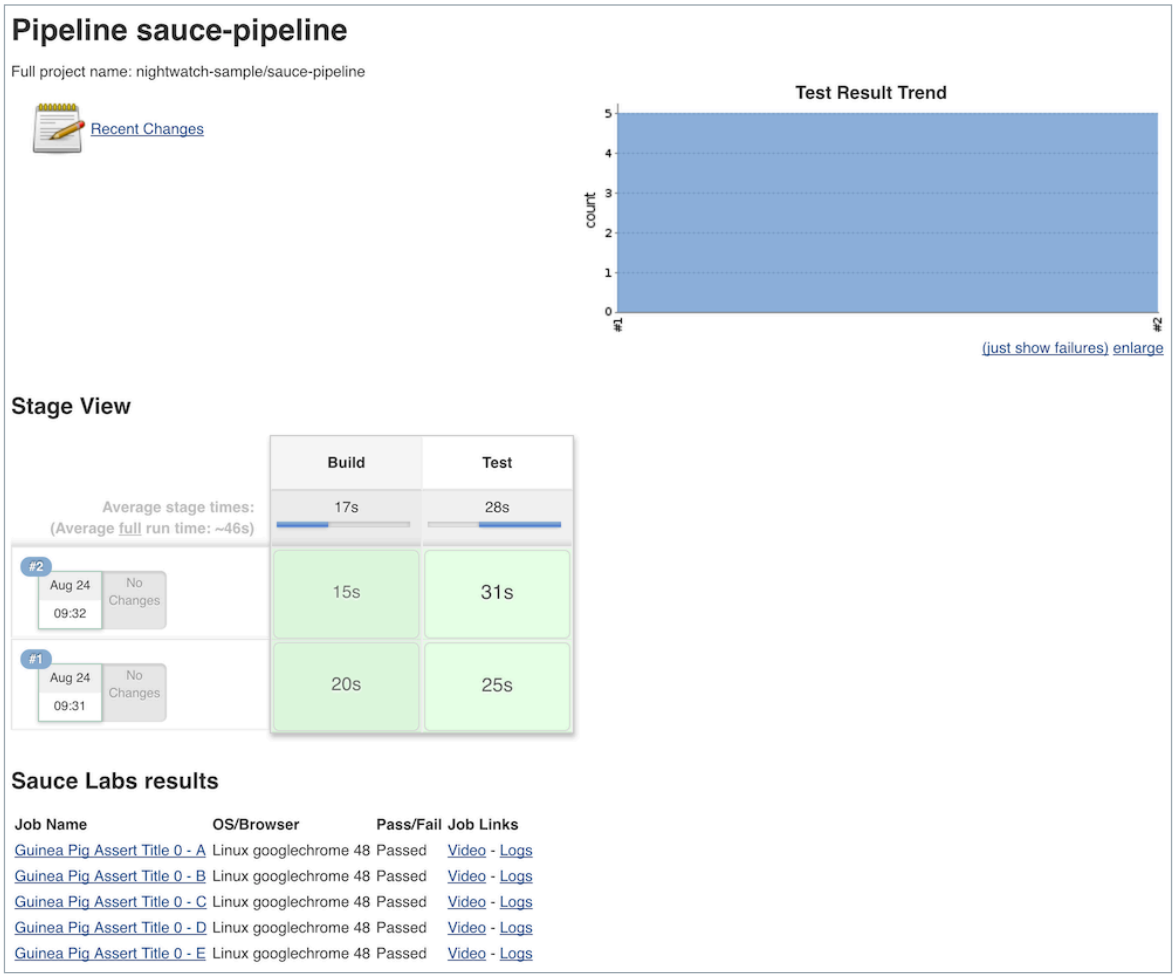
We will fork saucelabs-sample-test-frameworks/JS-Nightwatch.js and start by adding a Jenkinsfile. Between the sample and the Sauce Labs instructions, we'll be able to write a pipeline that runs five tests on one browser via Sauce Connect:

```
node {
  stage('Build') {
    checkout scm
    sh 'npm install' // <1>
  }

  stage('Test') {
    sauce('f0a6b8ad-ce30-4cba-bf9a-95afbc470a8a') { // <2>
      sauceconnect(options: '',
          useGeneratedTunnelIdentifier: false,
          verboseLogging: false) { // <3>
        sh './node_modules/.bin/nightwatch' +
          '-e chrome --test tests/guineaPig.js || true' // <4>
        junit 'reports/**' // <5>
        step([$class: 'SauceOnDemandTestPublisher']) // <6>
      }
    }
  }
}
```

Notes:

1. Install dependencies
2. Use previously added Sauce credentials. This ID string will be different on your Jenkins instance
3. Start up the Sauce Connect tunnel to Sauce Labs
4. Run Nightwatch.js
5. Use JUnit to track results and show a trend graph
6. Link result details from Sauce Labs

If we run this job a few times, the JUnit report will show a trend graph. Also, the sample app generates the



Job Name	OS/Browser	Pass/Fail	Job Links
Guinea Pig Assert Title 0 - A	Linux googlechrome 48	Passed	Video - Logs
Guinea Pig Assert Title 0 - B	Linux googlechrome 48	Passed	Video - Logs
Guinea Pig Assert Title 0 - C	Linux googlechrome 48	Passed	Video - Logs
Guinea Pig Assert Title 0 - D	Linux googlechrome 48	Passed	Video - Logs
Guinea Pig Assert Title 0 - E	Linux googlechrome 48	Passed	Video - Logs

appropriate `SauceOnDemandSessionID` for each test, enabling the Jenkins Sauce OnDemand plugin's result publisher to link results to details Sauce Labs captured during the run.

Adding Platforms

Next, we'll add a few more platforms to the matrix. This will require changing both the test framework configuration and the pipeline. We will need to add new named combinations of platform, browser and browser version (called "environments") to the Nightwatch.js configuration file, and modify the pipeline to run tests in those new environments.

This is another perfect example of the power of Pipeline as Code. If we were working with a separately configured pipeline, we would have to make the change to the test framework, then change the pipeline manually. With our pipeline checked in as code, we can change both in one commit, preventing errors resulting from pipeline configurations getting out of sync with the rest of the project.

I added three new environments to `nightwatch.json`:

```
"test_settings" : {
  "default": { /*----8<----8<----8<----*/ },
  "chrome": { /*----8<----8<----8<----*/ },

  "firefox": {
    "desiredCapabilities": {
      "platform": "linux",
      "browserName": "firefox",
      "version": "latest"
    }
  },
  "ie": {
    "desiredCapabilities": {
      "platform": "Windows 10",
      "browserName": "internet explorer",
      "version": "latest"
    }
  },
  "edge": {
    "desiredCapabilities": {
```

```
        "platform": "Windows 10",
        "browserName": "MicrosoftEdge",
        "version": "latest"
      }
    }
}
```

And we'll modify the Jenkinsfile to call them:

```
//----8<----8<----8<----8<----8<----8<----
sauceconnect(options: '',
    useGeneratedTunnelIdentifier: false,
    verboseLogging: false) {
  def configs = [ // <1>
      'chrome',
      'firefox',
      'ie',
      'edge'
  ].join(',')
  // Run selenium tests using Nightwatch.js
  sh "./node_modules/.bin/nightwatch" +
    " -e ${configs} --test tests/guineaPig.js || true"  // <2>
} //----8<----8<----8<----8<----8<----8<----
```

Notes:

1. Using an array to improve readability and make it easy to add more platforms later
2. Changed from single-quoted string to double-quoted to support variable substitution

NOTE: Test frameworks have bugs too. Nightwatch.js (v0.9.8) generates incomplete JUnit files, reporting results without enough information in them to distinguish between platforms. A fix has been implemented for this and submitted a PR to Nightwatch.js. This section shows output with that fix applied locally.

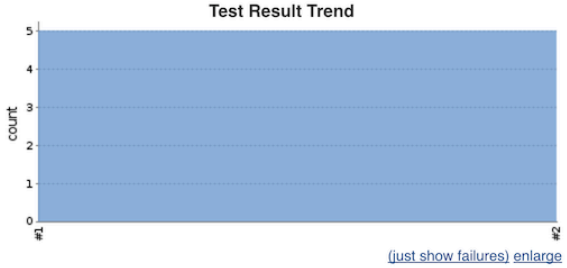
Sauce Labs results

Job Name	OS/Browser	Pass/Fail	Job Links
Guinea Pig Assert Title 0 - A	Linux googlechrome 48	Passed	Video - Logs
Guinea Pig Assert Title 0 - B	Linux googlechrome 48	Passed	Video - Logs
Guinea Pig Assert Title 0 - C	Linux googlechrome 48	Passed	Video - Logs
Guinea Pig Assert Title 0 - D	Linux googlechrome 48	Passed	Video - Logs
Guinea Pig Assert Title 0 - E	Linux googlechrome 48	Passed	Video - Logs
Guinea Pig Assert Title 0 - A	Linux firefox 45	Passed	Video - Logs
Guinea Pig Assert Title 0 - B	Linux firefox 45	Passed	Video - Logs
Guinea Pig Assert Title 0 - C	Linux firefox 45	Passed	Video - Logs
Guinea Pig Assert Title 0 - D	Linux firefox 45	Passed	Video - Logs
Guinea Pig Assert Title 0 - E	Linux firefox 45	Passed	Video - Logs
Guinea Pig Assert Title 0 - A	Windows 10 iexplore 11	Passed	Video - Logs
Guinea Pig Assert Title 0 - B	Windows 10 iexplore 11	Passed	Video - Logs
Guinea Pig Assert Title 0 - C	Windows 10 iexplore 11	Passed	Video - Logs
Guinea Pig Assert Title 0 - D	Windows 10 iexplore 11	Passed	Video - Logs
Guinea Pig Assert Title 0 - E	Windows 10 iexplore 11	Passed	Video - Logs
Guinea Pig Assert Title 0 - A	Windows 10 microsoftedge 13	Passed	Video - Logs
Guinea Pig Assert Title 0 - B	Windows 10 microsoftedge 13	Passed	Video - Logs
Guinea Pig Assert Title 0 - C	Windows 10 microsoftedge 13	Passed	Video - Logs
Guinea Pig Assert Title 0 - D	Windows 10 microsoftedge 13	Passed	Video - Logs
Guinea Pig Assert Title 0 - E	Windows 10 microsoftedge 13	Passed	Video - Logs

Pipeline sauce-pipeline

Full project name: nightwatch-sample/sauce-pipeline

Recent Changes



(just show failures) enlarge

Stage View

	Build	Test
Average stage times: (Average full run time: ~46s)	17s	28s
#2 Aug 24 09:32 No Changes	15s	31s
#1 Aug 24 09:31 No Changes	20s	25s

Sauce Labs results

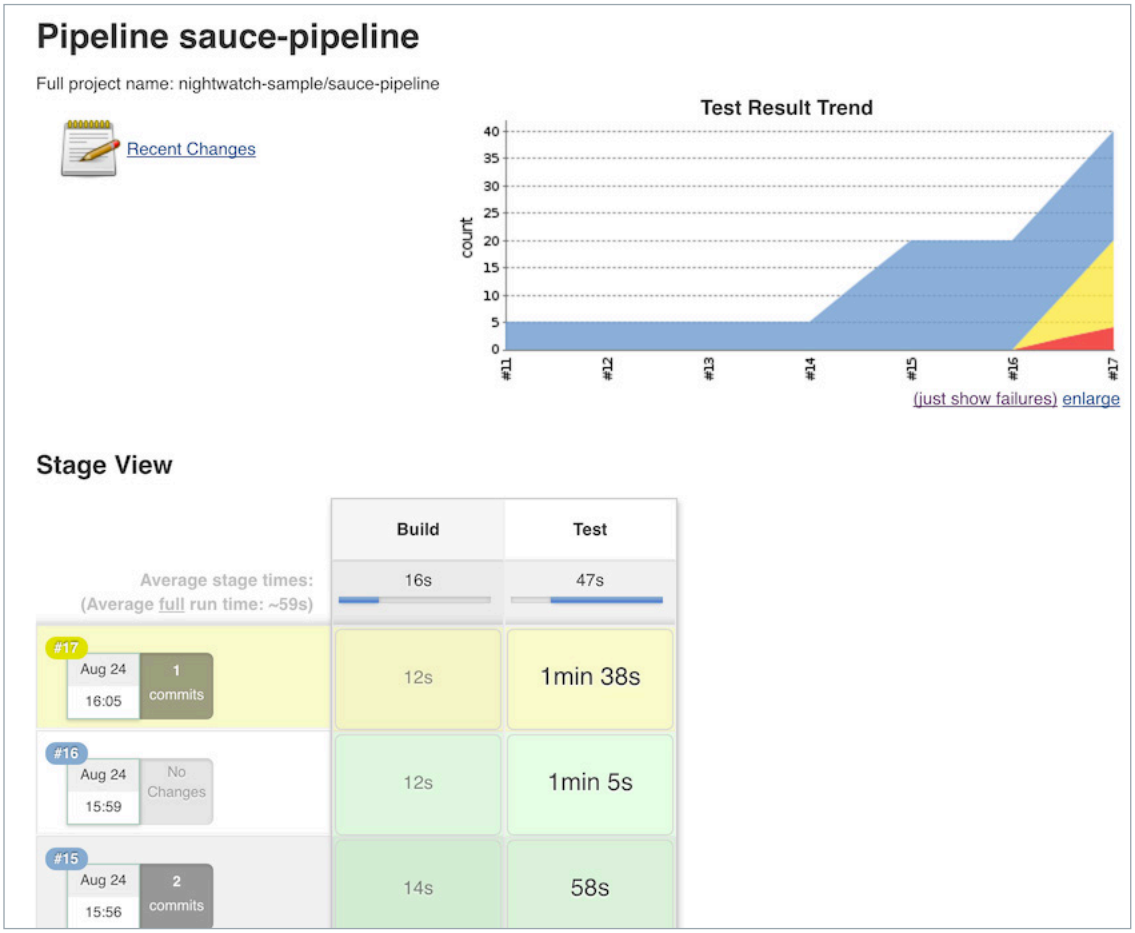
Job Name	OS/Browser	Pass/Fail	Job Links
Guinea Pig Assert Title 0 - A	Linux googlechrome 48	Passed	Video - Logs
Guinea Pig Assert Title 0 - B	Linux googlechrome 48	Passed	Video - Logs
Guinea Pig Assert Title 0 - C	Linux googlechrome 48	Passed	Video - Logs
Guinea Pig Assert Title 0 - D	Linux googlechrome 48	Passed	Video - Logs
Guinea Pig Assert Title 0 - E	Linux googlechrome 48	Passed	Video - Logs

As expected, Jenkins picked up the new pipeline and ran Nightwatch.js on four platforms. Sauce Labs recorded the results and correctly linked them into this build. Nightwatch.js was already configured to use multiple worker threads to run tests against those platforms in parallel, and our Sauce Labs account supported running them all at the same time, letting us cover four configurations in less than twice the time; the added time was mostly due to individual new environments taking longer to complete.

Conclusion

Considering the complexity of the system, it is rather easy to integrate Jenkins with Sauce OnDemand to start testing on multiple browsers. The plugin worked flawlessly with Jenkins Pipeline. Below, we'll go ahead and run some additional tests to show that failure reporting also behaves as expected.

```
//----8<----8<----8<----8<----8<----8<----
```



```
    sh "./node_modules/.bin/nightwatch -e ${configs} || true"  // <1>
//----8<----8<----8<----8<----8<----8<----
```

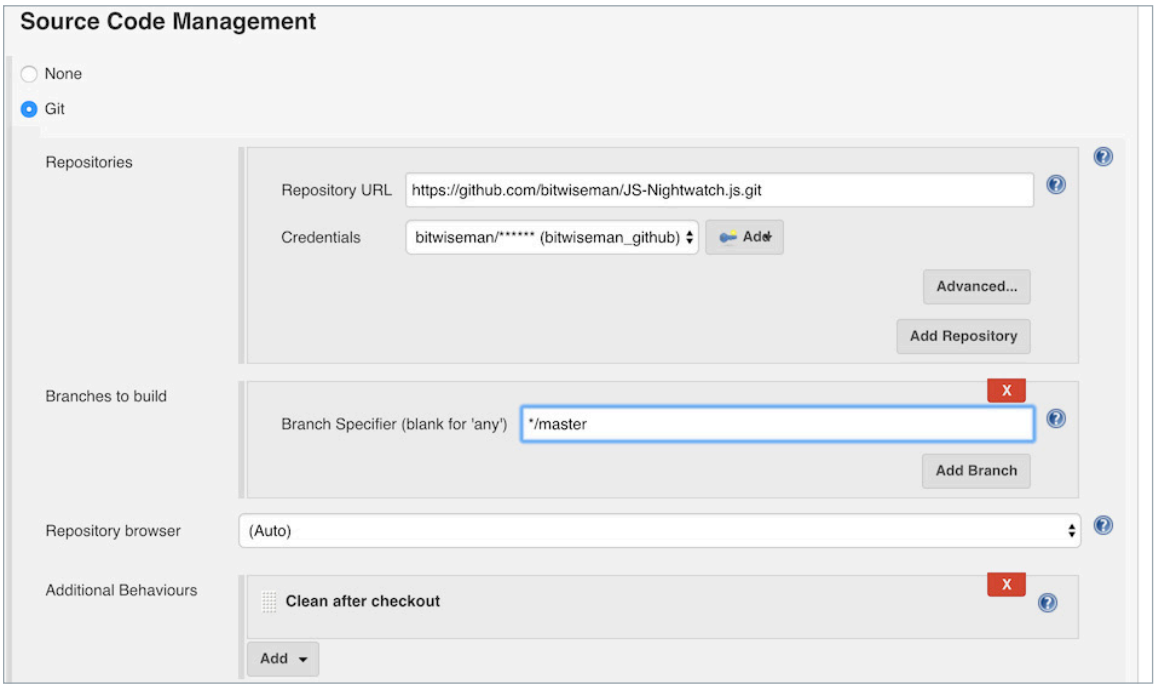
Notes:

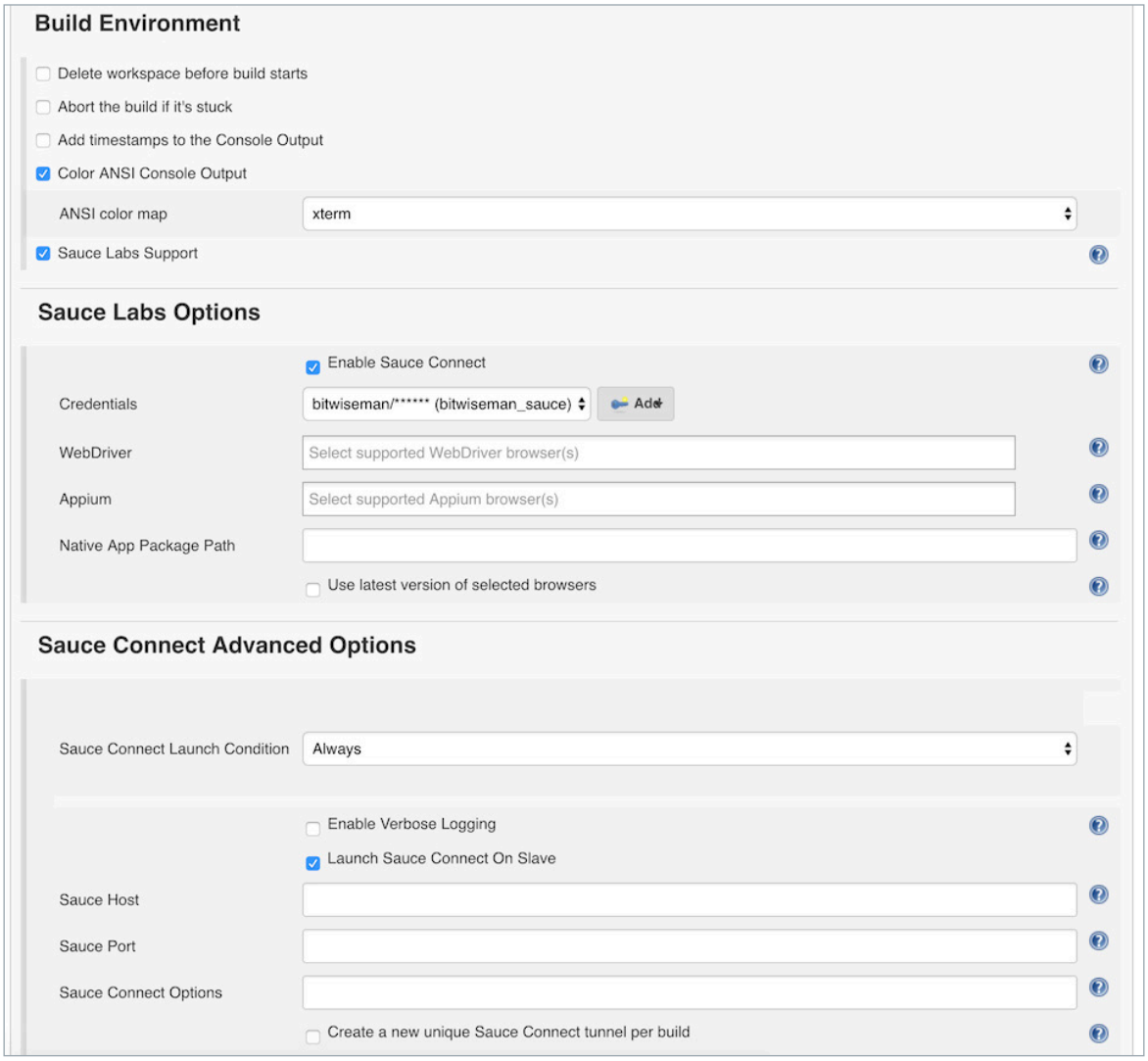
1. Removed – test filter to run all tests

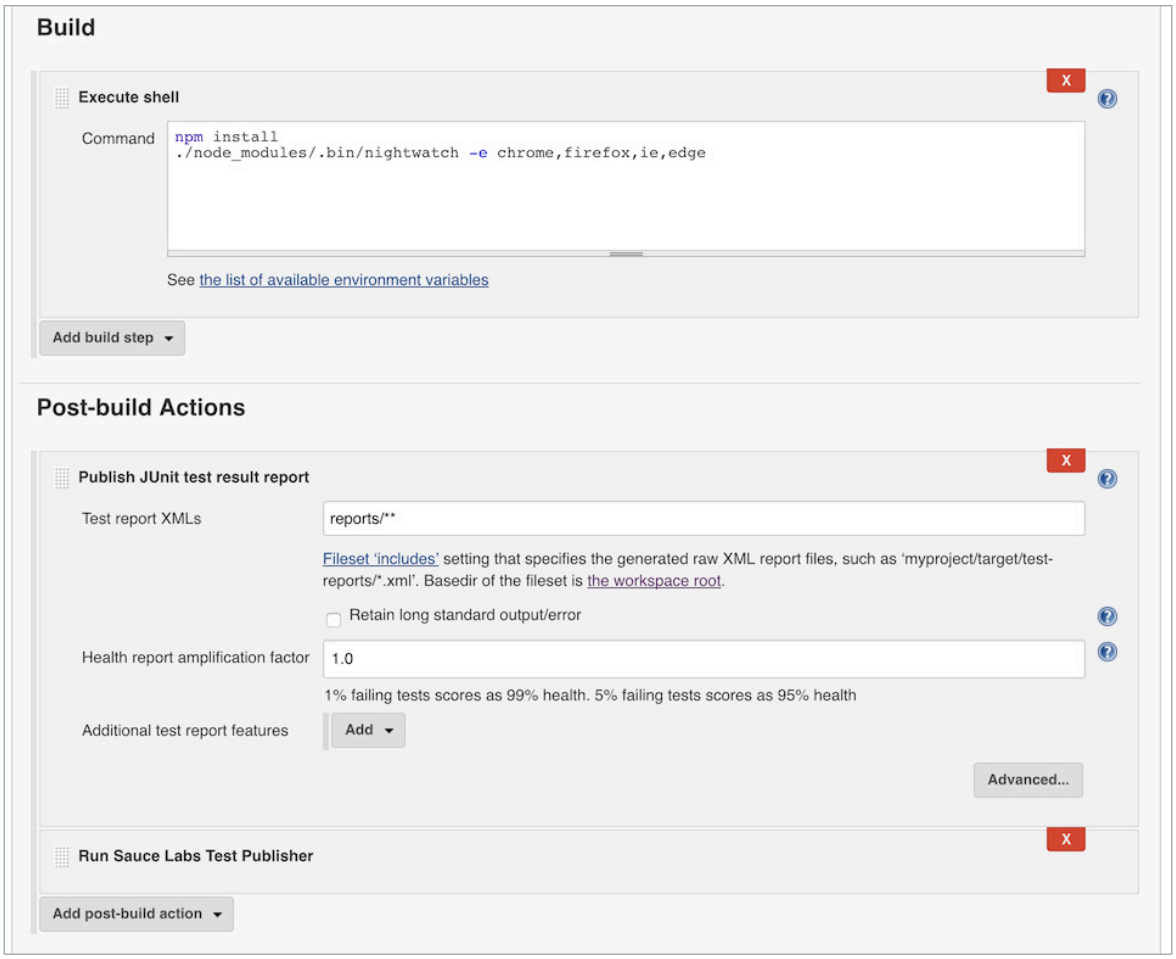
Epilogue: Jenkins Pipeline vs. Freestyle

Just for comparison, here's the final state of this job in Freestyle UI versus fully-commented Jenkins Pipeline code. NOTE: This includes the AnsiColor Plugin to support Nightwatch.js' default ANSI color output.

Freestyle




Build Environment
Delete workspace before build starts
Abort the build if it's stuck
Add timestamps to the Console Output
Color ANSI Console Output
ANSI color map
xterm
Sauce Labs Support
Sauce Labs Options
Enable Sauce Connect
Credentials
bitwiseman/****** (bitwiseman_sauce)
Add
WebDriver
Select supported WebDriver browser(s)
Appium
Select supported Appium browser(s)
Native App Package Path
Use latest version of selected browsers
Sauce Connect Advanced Options
Sauce Connect Launch Condition
Always
Enable Verbose Logging
Launch Sauce Connect On Slave
Sauce Host
Sauce Port
Sauce Connect Options
Create a new unique Sauce Connect tunnel per build



Jenkins Pipeline

```
node {
  stage('Build') {
    checkout scm

    // Install dependencies
    sh 'npm install'
  }

  stage('Test') {

    // Add sauce credentials
    sauce('f0a6b8ad-ce30-4cba-bf9a-95afbc470a8a') {
        // Start sauce connect
        sauceconnect(options: '',
            useGeneratedTunnelIdentifier: false,
            verboseLogging: false) {
```

```
            // List of browser configs we'll be testing against.
            def configs = [
                'chrome',
                'firefox',
                'ie',
                'edge'
            ].join(',')

            // Nightwatch.js supports color output
            // wrap this step for ansi color
            wrap(
              [$class: 'AnsiColorBuildWrapper', 'colorMapName': 'XTerm']) {

              // Run selenium tests using Nightwatch.js
              // Ignore error codes. The junit publisher will
              // cover setting build status.
              sh "./node_modules/.bin/nightwatch -e ${configs} || true"
            }

            junit 'reports/**'

            step([$class: 'SauceOnDemandTestPublisher'])
        }
    }
  }
}
```

Not only is the Pipeline as Code more compact, it also allows for comments that further clarify what is being done. As noted earlier, changes to this pipeline code are committed the same as changes to the rest of the project, keeping everything synchronized, reviewable and testable at any commit. In fact, you can view the full set of commits for this blog post in the blog/sauce-pipeline branch of the bitwiseman/JS-Nightwatch.js repository.

Links

- » saucelabs-sample-test-frameworks/JS-Nightwatch.js
- » bitwiseman/JS-Nightwatch.js
- » Sauce Connect

Using xUnit to Publish Results

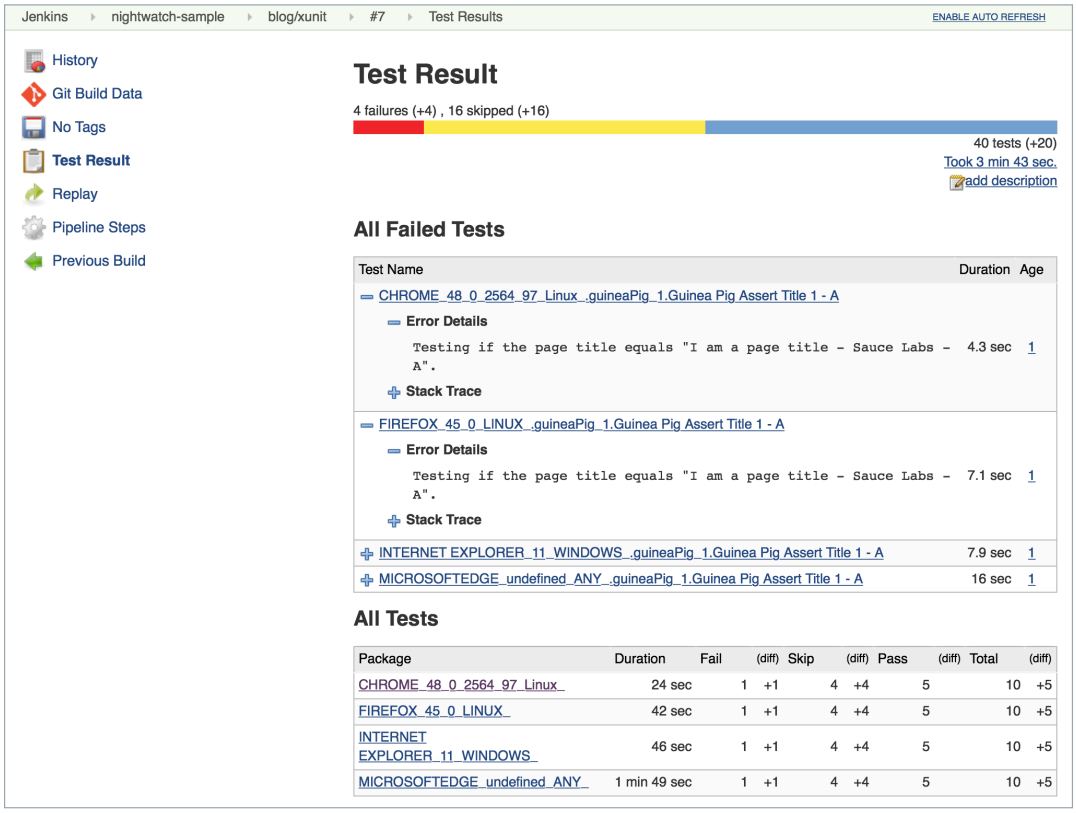
The JUnit plugin is the go-to test result reporter for many Jenkins projects, but it is not the only one available. The xUnit plugin is a viable alternative that supports JUnit and many other test result file formats.

Required plugins

» xUnit plugin (v1.102 or greater)

Introduction

No matter the project, you need to gather and report test results. JUnit is one of the most widely supported formats for recording test results. For scenarios where your tests are stable and your framework can produce JUnit output, the JUnit plugin is ideal for reporting results in Jenkins. It will consume results from a specified file or path, create a report and if it finds test failures, it will set the the job state to “unstable” or “failed.”



There are also plenty of scenarios where the JUnit plugin is not enough. If your project has some failing tests that will take some time to fix, or if there are some flaky tests, the JUnit plugin's simplistic view of test failures may be difficult to work with.

No problem, the Jenkins plugin model lets us replace the JUnit plugin functionality with similar functionality from another plugin. Jenkins Pipeline lets us do this in safe, stepwise fashion where we can test and debug each of our changes.

In this section, we'll cover how to replace the JUnit plugin with the xUnit plugin in Jenkins Pipeline code to address a few common test reporting scenarios.

Initial Setup

We'll use the JS-Nightwatch.js sample project from the previous section to demonstrate a couple of common scenarios that xUnit handles better. We will need to have the latest JUnit plugin and xUnit plugin installed on our Jenkins server. We can keep changes in the same fork of the JS-Nightwatch.js sample project on GitHub as the previous section, but use the blog/xunit branch.

Here is what the Jenkinsfile looked like at the end of the previous section and what the report page looks like after a few runs:

```
node {
  stage('Build') {
    checkout scm

    // Install dependencies
    sh 'npm install'
  }

  stage('Test') {

    // Add sauce credentials
    sauce('f0a6b8ad-ce30-4cba-bf9a-95afbc470a8a') {
        // Start sauce connect
        sauceconnect(options: '',
            useGeneratedTunnelIdentifier: false,
            verboseLogging: false) {

          // List of browser configs we'll be testing against.
          def configs = [
              'chrome',
              'firefox',
              'ie',
              'edge'
          ].join(',')
```

```
            // Nightwatch.js supports color output
            // wrap this step for ansi color
            wrap(
              [$class: 'AnsiColorBuildWrapper', 'colorMapName': 'XTerm']) {

              // Run selenium tests using Nightwatch.js
              // Ignore error codes. The junit publisher will
              // cover setting build status.
              sh "./node_modules/.bin/nightwatch -e ${configs} || true"
            }

            junit 'reports/**'

            step([$class: 'SauceOnDemandTestPublisher'])
        }
      }
    }
}
```

```
edge      SKIPPED:
edge    - Guinea Pig Assert Title 1 - B
edge    - Guinea Pig Assert Title 1 - C
edge    - Guinea Pig Assert Title 1 - D
edge    - Guinea Pig Assert Title 1 - E
edge
+ true
[Pipeline] }
[Pipeline] // wrap
[Pipeline] step
Recording test results
[Pipeline] step
Starting Sauce Labs test publisher
Finished Sauce Labs test publisher
[Pipeline] }
```

Switching from JUnit to xUnit

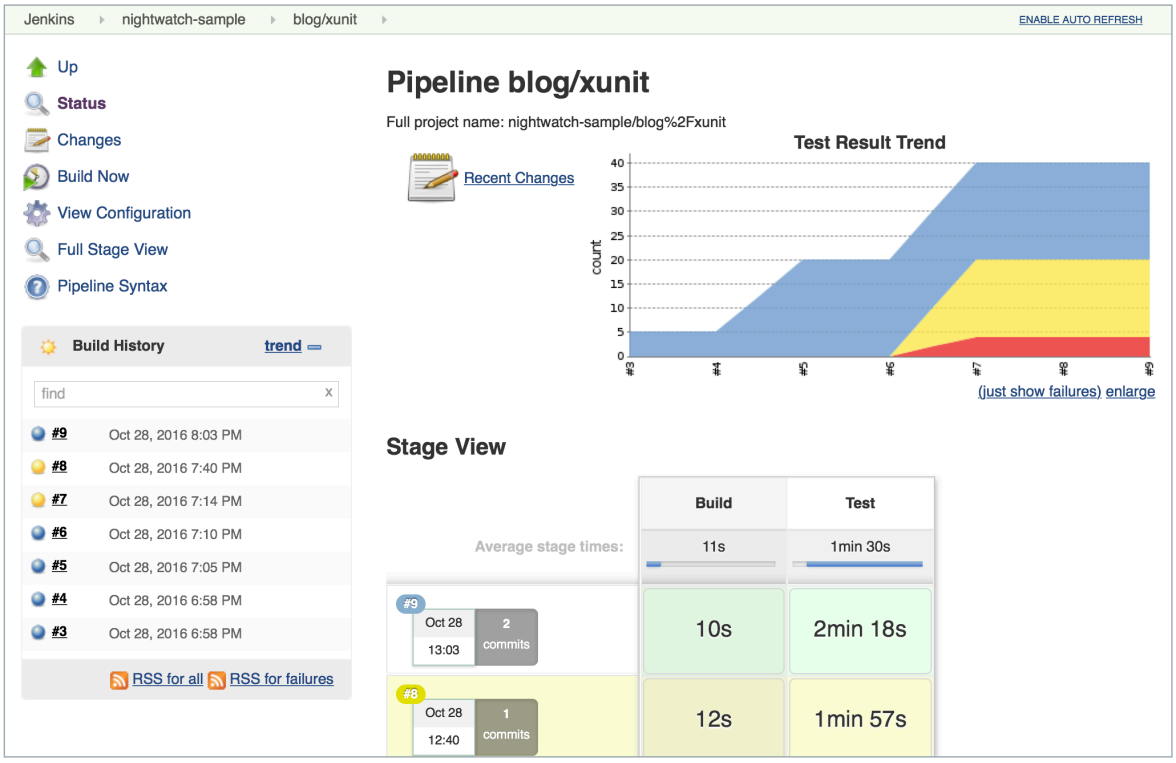
We'll start by replacing JUnit with xUnit in our pipeline. We will use the Snippet Generator to create the step with the right parameters. The main downside of using the xUnit plugin is that while it is Jenkins Pipeline compatible, it still uses the more verbose step() syntax – and it has some very rough edges around that, too.

```
// Original JUnit step
junit 'reports/**'

// Equivalent xUnit step - generated (reformatted)
step([$class: 'XUnitBuilder',
  testTimeMargin: '3000', thresholdMode: 1,
```

```
    thresholds: [
      [$class: 'FailedThreshold',
        failureNewThreshold: '', failureThreshold: '',
        unstableNewThreshold: '', unstableThreshold: '1'],
      [$class: 'SkippedThreshold',
        failureNewThreshold: '', failureThreshold: '',
        unstableNewThreshold: '', unstableThreshold: '']],
    tools: [
      [$class: 'JUnitType', deleteOutputFiles: false,
        failIfNotNew: false, pattern: 'reports/**',
        skipNoTestFiles: false, stopProcessingIfError: true]]
])

// Equivalent xUnit step - cleaned
step([$class: 'XUnitBuilder',
    thresholds: [[$class: 'FailedThreshold', unstableThreshold: '1']],
    tools: [[$class: 'JUnitType', pattern: 'reports/**']]])
```

If we replace the `junit` step in our Jenkinsfile with that last example above, it produces a report and job result identical to the JUnit plugin but using the xUnit plugin. Easy!

```
node {
  stage('Build') { /* ... */ }

  stage('Test') {

    // Add sauce credentials
    sauce('f0a6b8ad-ce30-4cba-bf9a-95afbc470a8a') {
        // Start sauce connect
        sauceconnect( /* ... */ ) {
          // ... snip ...

          // junit 'reports/**'
          step([$class: 'XUnitBuilder',
            thresholds: [
              [$class: 'FailedThreshold', unstableThreshold: '1']],
            tools: [[$class: 'JUnitType', pattern: 'reports/**']]])

          // ... snip ...
      }
    }
  }
}
```



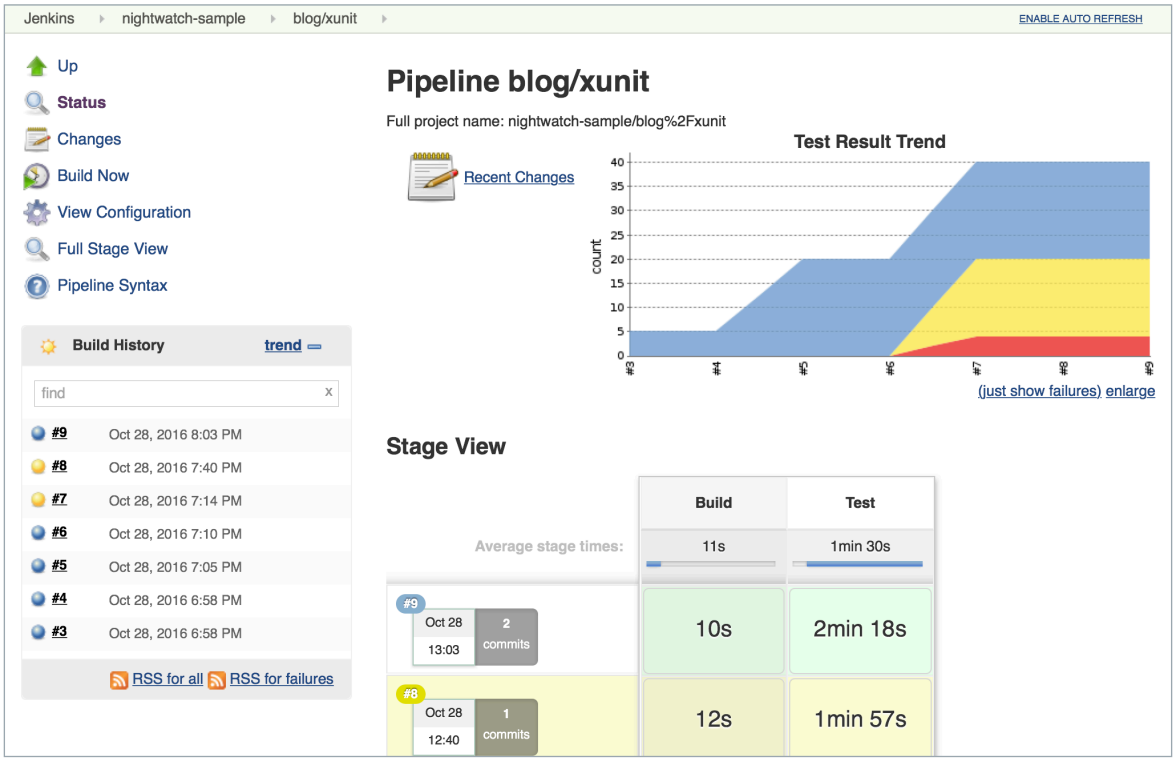

```
edge      SKIPPED:
edge    - Guinea Pig Assert Title 1 - B
   - Guinea Pig Assert Title 1 - C
edge    - Guinea Pig Assert Title 1 - D
   - Guinea Pig Assert Title 1 - E
edge
+ true
[Pipeline] }
[Pipeline] // wrap
[Pipeline] step
[xUnit] [INFO] - Starting to record.
[xUnit] [INFO] - Processing JUnit
[xUnit] [INFO] - [JUnit] - 8 test report file(s) were found with the pattern
'reports/**' relative to
'/Users/bitwiseman/jenkins/agents/osx_mbp/workspace/ightwatch-sample_blog_xunit-
QF55R3KGV2ZYWCASCYDEDD6WMNYDYLRLC6JD5C7UNXW5Q33MMFVA' for the testing framework
'JUnit'.
[xUnit] [INFO] - Check 'Failed Tests' threshold.
[xUnit] [INFO] - The total number of tests for this category exceeds the specified
'unstable' threshold value.
[xUnit] [INFO] - Setting the build status to UNSTABLE
[xUnit] [INFO] - Stopping recording.
[Pipeline] step
Starting Sauce Labs test publisher
Finished Sauce Labs test publisher
[Pipeline] }
```

Accept a Baseline

Most projects don't start off with automated tests that will pass or even will run. They start with developers hacking and prototyping, and eventually they start to write tests. As new tests are written, having tests checked-in, running and failing can provide valuable information. With the xUnit plugin, we can accept a baseline of failed cases and drive that number down over time. Now that we've switched to the xUnit Plugin we can modify our pipeline to fail jobs only if the number of failures is greater than an expected baseline – in this case, four failures. When we run the job with the following change, the reported numbers will remain the same, but the job will be marked as passing.

Jenkinsfile

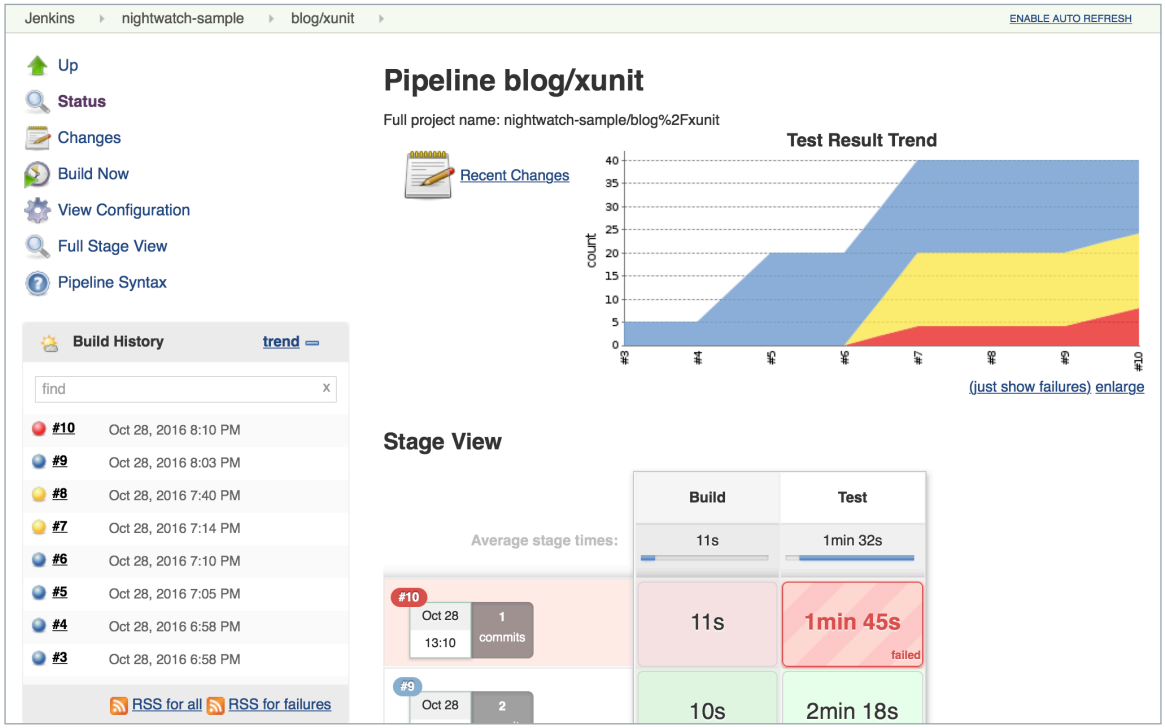
```
// The rest of the Jenkinsfile is unchanged.
// Only the xUnit step() call is modified.
step([$class: 'XUnitBuilder',
  thresholds: [[$class: 'FailedThreshold', failureThreshold: '4']],
  tools: [[$class: 'JUnitType', pattern: 'reports/**']]])
```



Next, we can also check that the plugin reports the job as failed if more failures occur. Since this is sample code, we'll do this by adding another failing test and checking that the job is marked as failed on the next run.

tests/guineaPig.js

```
// ... snip ...
    'Guinea Pig Assert Title 0 - D': function(client) { /* ... */ },

    'Guinea Pig Assert Title 0 - E': function(client) {
      client
        .url('https://saucelabs.com/test/guinea-pig')
        .waitForElementVisible('body', 1000)
        //.assert.title('I am a page title - Sauce Labs');
        .assert.title('I am a page title - Sauce Labs - Cause a Failure');
    },

    afterEach: function(client, done) { /* ... */ }
// ... snip ...
```



In a real project, we'd make fixes over a number of commits, progressively bringing the number of failures down and adjusting our baseline to match. Since this is a sample, we'll just make all tests pass, and set the job failure threshold for failed and skipped cases to zero.

Jenkinsfile

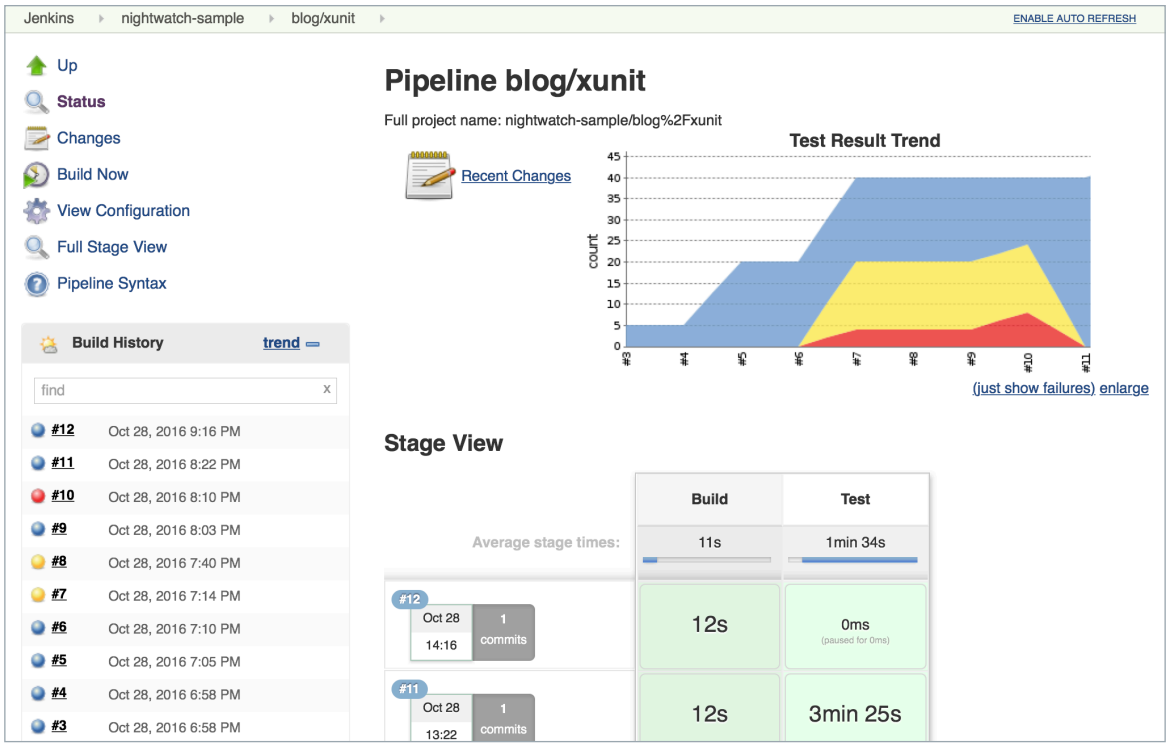
```
// The rest of the Jenkinsfile is unchanged.
// Only the xUnit step() call is modified.
step([$class: 'XUnitBuilder',
  thresholds: [
    [$class: 'SkippedThreshold', failureThreshold: '0'],
    [$class: 'FailedThreshold', failureThreshold: '0']],
  tools: [[$class: 'JUnitType', pattern: 'reports/**']]])
```

tests/guineaPig.js

```
// ... snip ...
    'Guinea Pig Assert Title 0 - D': function(client) { /* ... */ },

    'Guinea Pig Assert Title 0 - E': function(client) {
      client
        .url('https://saucelabs.com/test/guinea-pig')
        .waitForElementVisible('body', 1000)
        .assert.title('I am a page title - Sauce Labs');
    },

    afterEach: function(client, done) { /* ... */ }
// ... snip ...
```

tests/guineaPig_1.js

```
// ... snip ...
    'Guinea Pig Assert Title 1 - A': function(client) {
      client
        .url('https://saucelabs.com/test/guinea-pig')
        .waitForElementVisible('body', 1000)
        .assert.title('I am a page title - Sauce Labs');
    },
// ... snip ...
```



Allow for Flakiness

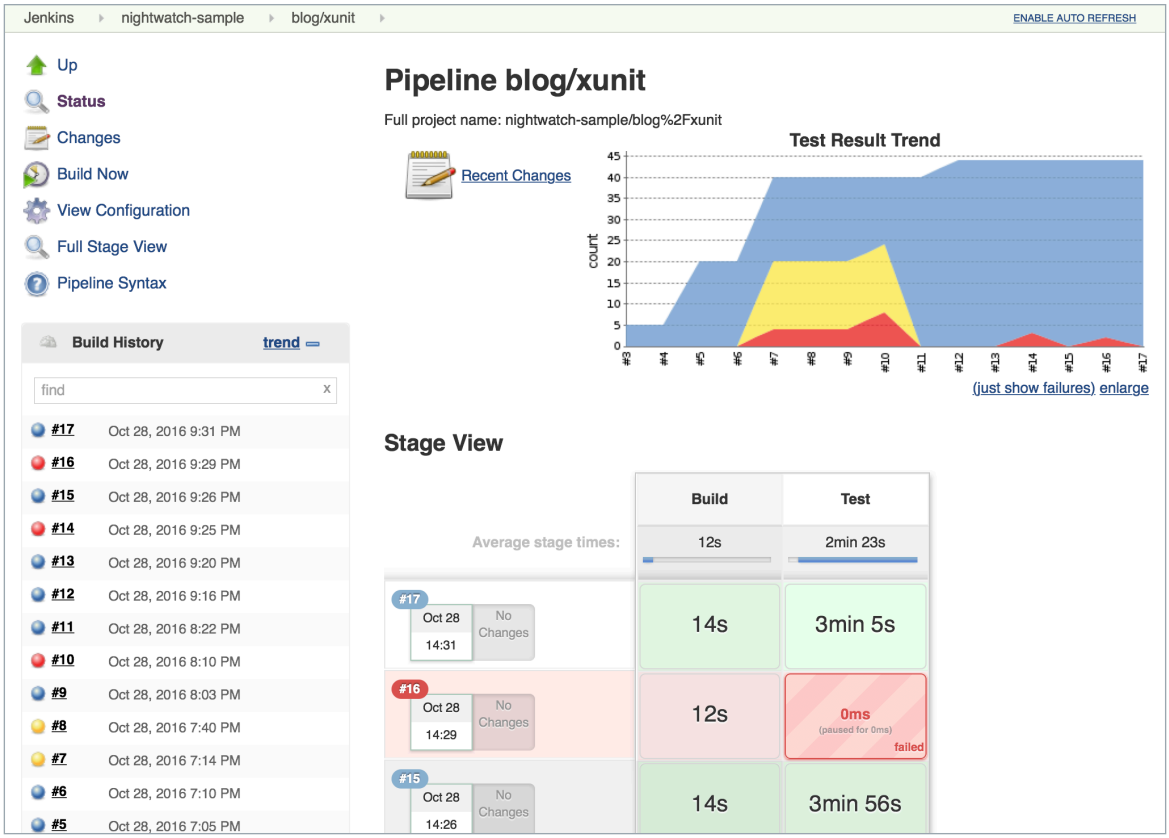
We have all known the frustration of having one flaky test that fails once every ten jobs. You want to keep it active so you can work on isolating the source of the problem, but you also don't want to destabilize your CI pipeline or reject commits that are actually okay. You could move the test to a separate job that runs the flaky tests, but that just leads to a job that is always in a failing state and a pile of flaky tests that no one looks at.

With the xUnit plugin, we can keep the flaky test in our main test suite, but still allow the job to pass. Let's add a sample flaky test. After a few runs, we can see that the test fails intermittently and causes the job to fail, too.

tests/guineaPigFlaky.js

```
// New test file: tests/guineaPigFlaky.js
var https = require('https');
var SauceLabs = require("saucelabs");

module.exports = {
  '@tags': ['guineaPig'],
  'Guinea Pig Flaky Assert Title 0': function(client) {
    var expectedTitle = 'I am a page title - Sauce Labs';
    // Fail every fifth minute
    if (Math.floor(Date.now() / (1000 * 60)) % 5 === 0) {
      expectedTitle += " - Cause failure";
    }
```

```
    client
      .url('https://saucelabs.com/test/guinea-pig')
      .waitForElementVisible('body', 1000)
      .assert.title(expectedTitle);
  }

  afterEach: function(client, done) {
    client.customSauceEnd();

    setTimeout(function() {
      done();
    }, 1000);
  }
};
```



You can almost hear your teammates screaming in frustration, just looking at this report. To allow specific tests to be unstable but not others, we will add a guard "suite completed" test to the suites that should be stable, and keep the flaky test on its own. Then we'll tell xUnit to allow a number of failed tests, but no skipped ones. If any test fails other than the ones we allow to be flaky, it will result in one or more skipped tests and will fail the build.

Jenkinsfile

```
// The rest of the Jenkinsfile is unchanged.
// Only the xUnit step() call is modified.
step([$class: 'XUnitBuilder',
  thresholds: [
    [$class: 'SkippedThreshold', failureThreshold: '0'],
    // Allow for a significant number of failures
    // Keeping this threshold so that large failures are guaranteed
    //     to still fail the build
    [$class: 'FailedThreshold', failureThreshold: '10']],
  tools: [[$class: 'JUnitType', pattern: 'reports/**']]])
```

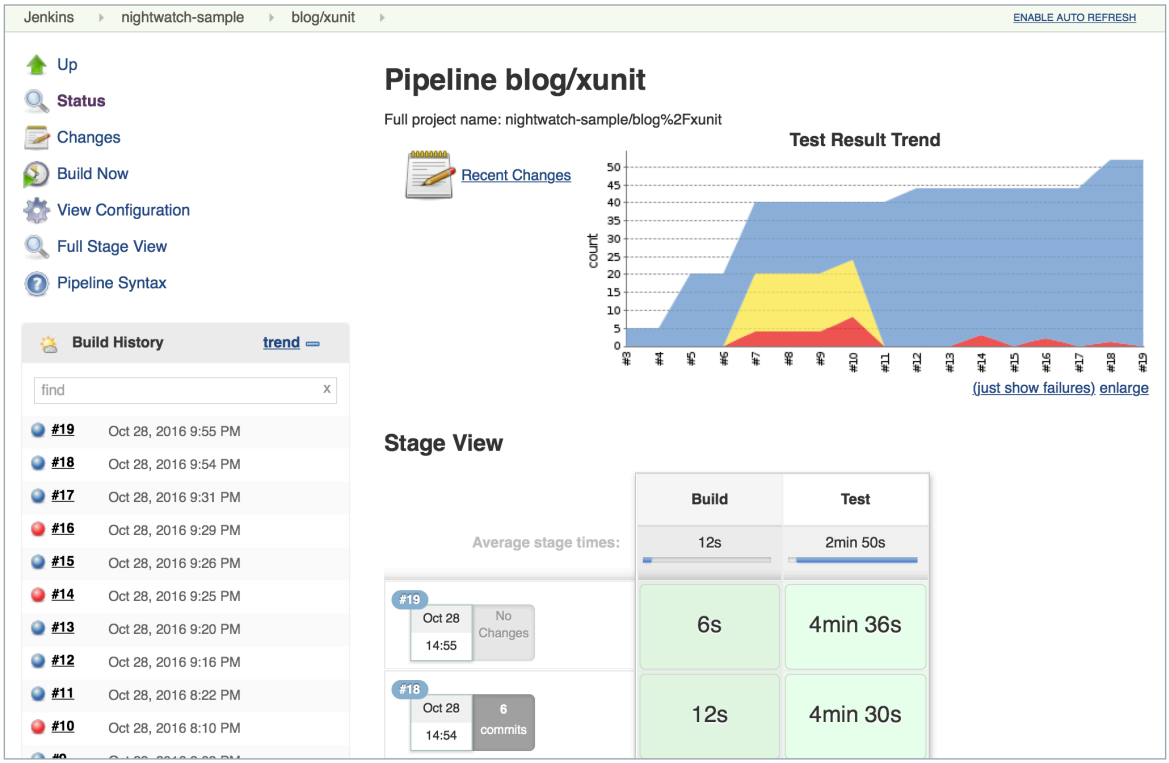
tests/guineaPig.js

```
// ... snip ...
  'Guinea Pig Assert Title 0 - E': function(client) { /* ... */ },

  'Guinea Pig Assert Title 0 - Suite Completed': function(client) {
    // No assertion needed
  },

  afterEach: function(client, done) { /* ... */ }
// ... snip ...
```

tests/guineaPig_1.js

```
// ... snip ...
  'Guinea Pig Assert Title 1 - E': function(client) { /* ... */ },

  'Guinea Pig Assert Title 1 - Suite Completed': function(client) {
    // No assertion needed
  },

  afterEach: function(client, done) { /* ... */ }
// ... snip ...
```

After a few more runs, as you can see in build number 18, the flaky test is still being flaky, but it is no longer failing the build. Meanwhile, if another test fails, it will cause the "suite completed" test to be skipped, failing the job. If this were a real project, the test owner could then instrument and eventually fix the test. When they were confident they had stabilized the test, they could add a `suite_completed` test after it to enforce passing without having to make changes to other tests or framework.



Conclusion

This section has shown how to migrate from the JUnit plugin to the xUnit plugin on an existing project in Jenkins Pipeline. It also covered how to use the features of the xUnit plugin to get more meaningful and effective Jenkins reporting. Not covered was how many other formats xUnit supports – from CCPUnit to MSTest. You can also write your own XSL for result formats not on the known/supported list.

Links

- xUnit plugin
- bitwiseman/JS-Nightwatch.js
- Saucelabs-sample-test-frameworks

Summary

In this guide, we've shown a number of use cases for Jenkins plugins and Jenkins Pipeline.

Starting with an outline of stages, we have:

- Constructed a pipeline from scratch, iterating on it to build a continuous delivery pipeline using Docker containers that interact with each other
- Added an HTML report to an existing pipeline
- Made Jenkins notify us when builds start, succeed or fail
- Implemented cloud-based and parallelized browser testing
- Improved result processing with tolerance for flaky tests

We hope you have found it informative and helpful.

For additional resources, please visit: www.cloudbees.com/devops/continuous-delivery/pipeline

CloudBees, Inc.
4 North Second Street | Suite 1270
San Jose, CA 95113
United States
www.cloudbees.com
info@cloudbees.com