

Increased Regression Efficiency with Jenkins Continuous Integration Before You Finish Your Morning Coffee

Thomas Ellis
Mentor Graphics Corp.
8005 SW Boekman Rd.
Wilsonville, OR 97070

Abstract- As verification engineers, we are always looking for ways to automate otherwise manual tasks. In case you have not heard, we are constantly trying to do more with less. Continuous Integration is a practice which has been widely, and successfully used in the software realm for many years. Deploying a continuous integration server such as Jenkins not only provides a way to automate the running of jobs, and collection of results, but it also allows for teams to reap the benefits of a continuous integration practices, which ultimately leads to a cleaner repository, with less integration headaches. Among many other benefits, Jenkins also provides a web dashboard to view and analyze results in a common place, regardless of how spread out your team may be. Its open source, has a strong community behind it, and you can start seeing the benefits by getting it up and running a regression in your environment before you even finish your morning cup of coffee.

I. INTRODUCTION

The topic of Continuous Integration (CI) is one which has started to become more and more common in the world of verification. For those unfamiliar with CI, it is a concept often associated with Agile programming practices, and it runs off the basic principle that the longer a branch of code is checked out, the more it begins to drift away from what is stored in the repository. The more the two diverge, the more complicated it becomes to eventually merge in changes easily. Ultimately leading to what is commonly referred to as “integration hell”. To avoid this, and ultimately save engineers time, CI calls for integrating regularly and often (typically daily).

Regular check-ins are of course, only half the equation, you need to be able to verify their changes quickly as well, otherwise many small check ins over several days, is no different than one large check in at weeks end. Commonly, in a Continuous Integration environment, a CI server monitors the source control for check in's, which in turn triggers a CI process (time-based triggers are also common). This process will then build the necessary design files, and run the requisite integration tests. Once complete, the results of the tests are reported back to the user, and assuming everything passed, can now be safely committed to the repository.

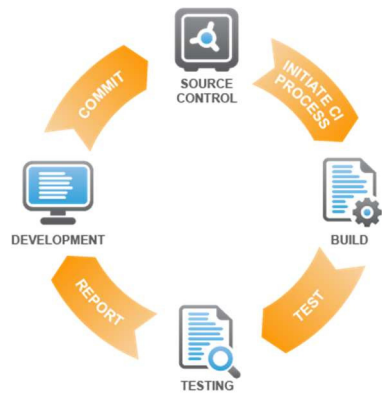


Figure 1. Continuous Integration Flow

By following this model, issues can be caught earlier in the development process, and can be resolved quicker as there is less variance between check ins.

There are several options to choose from when it comes to CI, however, far and away the most common solution today is Jenkins. So what is Jenkins, and why is it the favorite CI tool of so many users?



Figure 2. Jenkins CI

II. MEET JENKINS

Jenkins was originally created by Kohsuke Kawaguchi while he was working for Sun Microsystems back in 2004. At the time, however, the project was called Hudson. It was not until several years later in 2010, after Oracle had acquired Sun, issues began to arise between the community working on Hudson, and Oracle. At that time, a vote was taken as to whether development would continue, or if the community would break ties with Oracle, and fork the project. After an overwhelming vote, the project was forked, and Jenkins was born. A majority of those working on and using Hudson at the time, eventually migrated over to Jenkins. Today there are over 130k active installations of Jenkins

worldwide [1].

ZeroTurnaround is a development company, which amongst other things, conducts an annual global survey of Java developers, and produces a report of the tools and technologies most commonly used by the industry. Just recently, they released their 2016 survey results [2] which polled over 2000 respondents on various topics, one of which was their use of CI. Of those respondents, 60% were stated to be using Jenkins, the second place tool was Bamboo, used by only 9% of the respondents. Oh, and remember Hudson, it came in at only 3%.

Jenkins has wide appeal to many developers. Firstly, it is open-source and freely available. With an incredibly active and dedicated community, issues are resolved very quickly, and members are eager to help. As a tool, it is easy to install, and highly configurable for any environment via, all through its convenient web interface. While Jenkins itself has a lot to offer, one of its biggest benefits is that it is so highly extensible via plugins to the tool. At present, it boasts nearly 1400 plugins from close to 600 contributors. If there is something Jenkins does not do natively, chances are, there is a plugin to enable that functionality.

To get started to get a feel for all that Jenkins has to offer, you really need to try and use the tool. After all, I said it is easy to setup, and I think you will agree once you try it for yourself.

III. INSTALLING JENKINS

Ok, enough background, it is time to take a look at actually using Jenkins. The process of getting Jenkins running for the first time is remarkably simple. It should be noted, there are several ways to install Jenkins, from leveraging a Tomcat instance, to taking advantage of the built-in packages provided by some OS's (such as Red Hat and Windows) [3], however for this paper we will focus on the most general method which can be used regardless of the OS you're running on, which is to use built-in Jetty servlet container that comes with Jenkins.

The first step is to download the Jenkins war file from the Jenkins website [4]. Once downloaded, all you need to do is type the following command:

```
java -jar jenkins.war > jenkins.log &
```

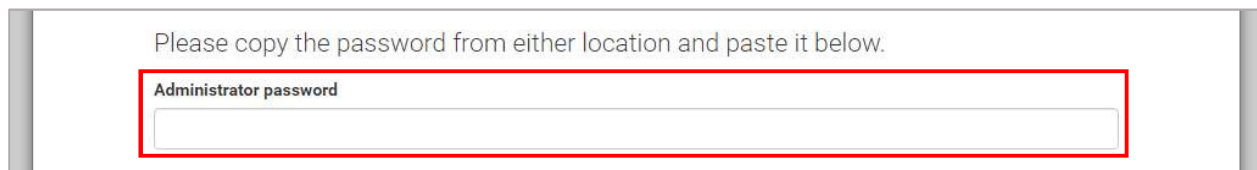
If Jenkins fails to start, look in the log file for an error message, most commonly if you have an issue at this stage, it is because there is another process already using the default http port that Jenkins runs on which is 8080. If that happens, try starting Jenkins by running on port that is not currently in use, you can do that by adding ‘--httpPort=<port>’ to the above command.

Now that Jenkins is up and running, you should be able to access the server by navigating to the following address in your browser (where <servername> is the name of the machine you are running on):

```
http://<servername>:8080
```

You should be presented with a ‘Getting Started’ screen. There are a few setup items that Jenkins will ask for before we can start using Jenkins.

The first thing Jenkins will ask for is a password to unlock the instance. This is an alphanumeric string that was printed out to the log file when you initially started Jenkins, copy and paste that string into Jenkins.

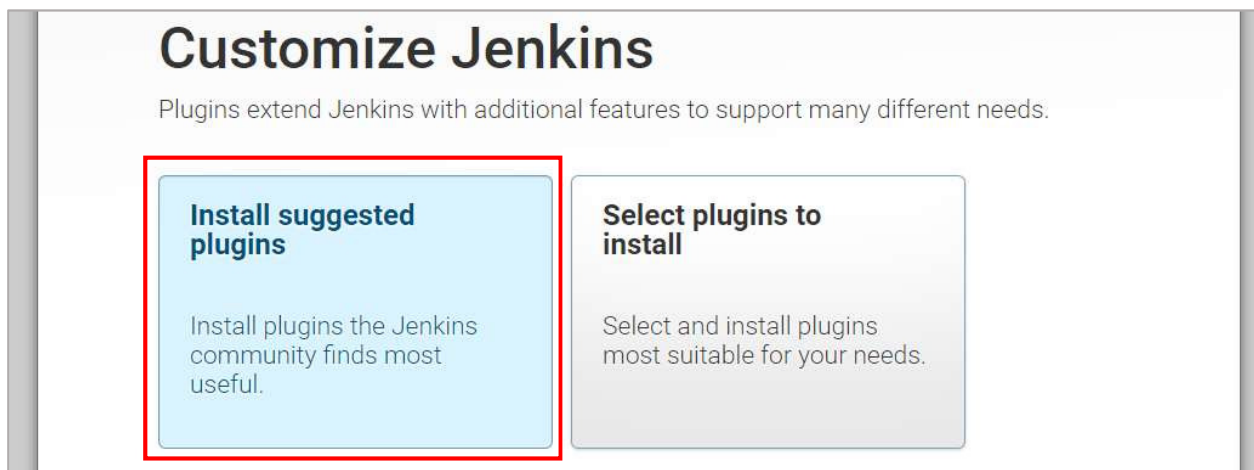


Please copy the password from either location and paste it below.

Administrator password

Figure 3. Admin Password

Next Jenkins will ask what plugins you want installed initially. You will have a chance later to install any additional plugins you wish, so for now, just click ‘Install suggested plugins’:



Customize Jenkins

Plugins extend Jenkins with additional features to support many different needs.

Install suggested plugins

Install plugins the Jenkins community finds most useful.

Select plugins to install

Select and install plugins most suitable for your needs.

Figure 4. Installing Plugins

It will take a few moments for each of the plugins to install, once the process is completed, you will be presented with the option to setup admin users for the Jenkins instance. By default, Jenkins requires users to create accounts and login to use Jenkins (you can disable this later if you wish to have an open access policy). This also allows Jenkins to keep track of additional information on a per user basis, such as who started a particular build. Fill out the form to create your first user account, and click 'Save and Finish'.

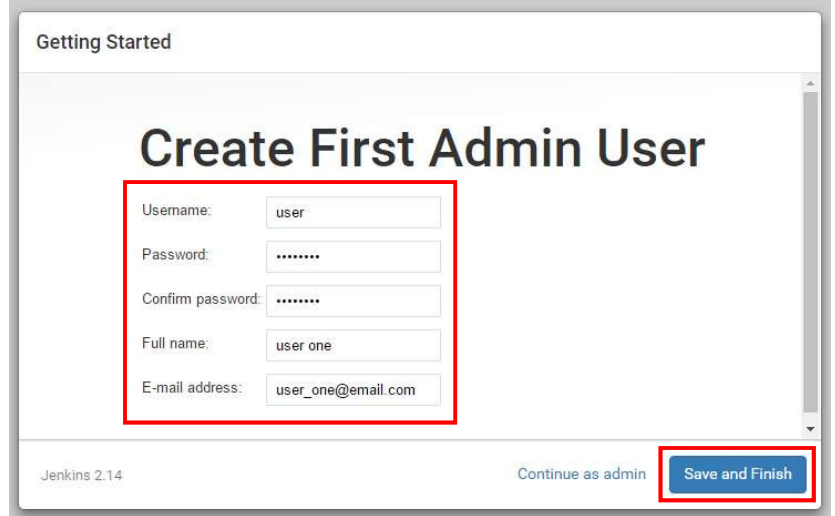


Figure 5. Create Admin User

Everything should now be setup, so click 'Start Using Jenkins' to be taken to the main page of your Jenkins instance. In the future, this is the page you will be presented with when you navigate to your Jenkins instance (even if you stop and start the Jenkins process).

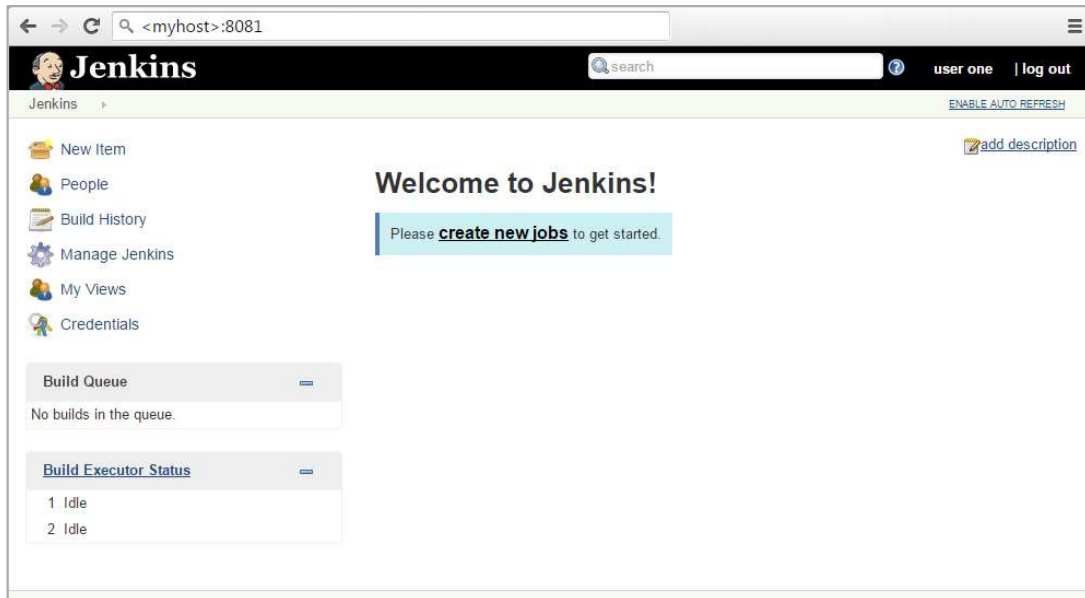
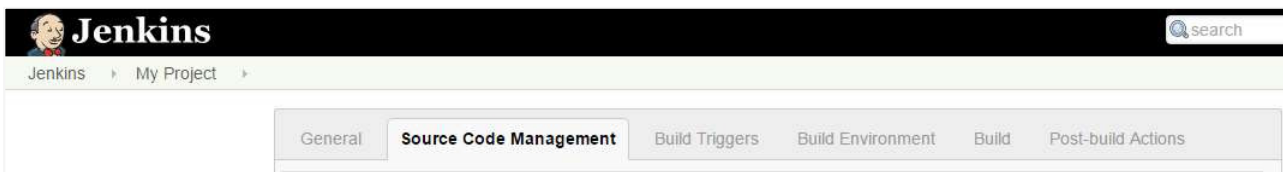


Figure 6. Jenkins Start Page

IV. RUNNING A REGRESSION IN JENKINS

It is time to dive in, and take a look at what running a job in Jenkins looks like. Clicking the 'create new jobs' link will allow us to create a new job, which Jenkins refers to as a 'project'. There are a couple project types which Jenkins supports, for our



case however, the 'Freestyle' project will be the best fit. Once you give your project a name, and click 'Ok', you will be taken to the configuration page, which will look like this:

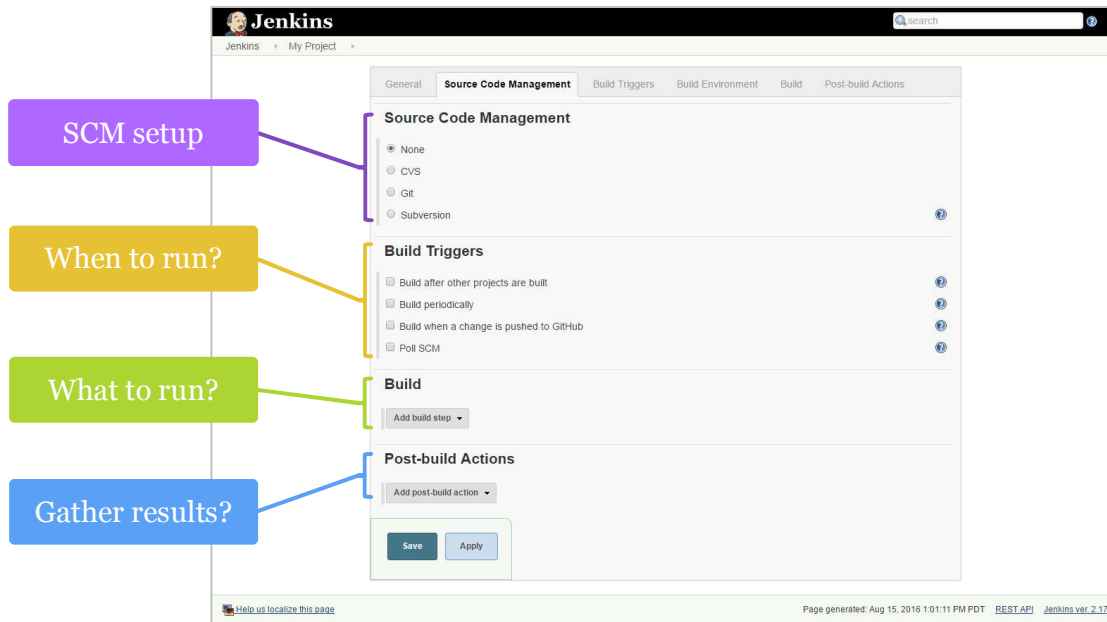


Figure 7. Jenkins Project Setup

Here you can see the basic steps for configuring a project in Jenkins. Tasks in Jenkins are represented by builds. A build could be a complete regression, it could be the running of unit tests, or any other task you may wish Jenkins to automate.

Breaking down each of the sections in the configuration page; first you have the option to setup your source code management (SCM). This is not a requirement of course, but this will allow Jenkins to poll your SCM for changes to initiate builds. Everyone has experienced at one time or another, experience the situation where multiple check-ins were made before a regression was ran, and then inevitably the regression failed, and you had to try and figure out which check-in broke the build. By allowing Jenkins to monitor your SCM system, you can always ensure each check-in is regressed and verified, helping to ensure a clean repository.

Out of the box, Jenkins supports CVS, Git, and Subversion (SVN), however if your team uses a different tool, there are plugins available for most common SCM tools. Figure 8 shows a simple example of setting up Jenkins to monitor a SVN repository for check-ins. You can get more elaborate by creating a remote repository with credentials, however, since this will be different for every reader, this simple example will suffice for the purposes of this paper.

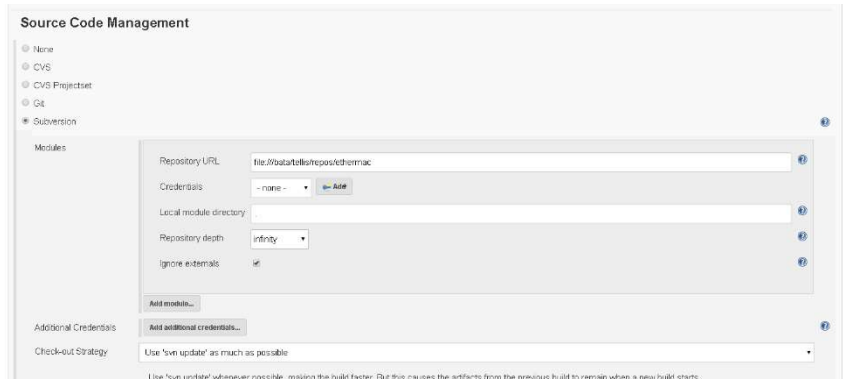


Figure 8. SCM Setup

Next, you have to tell Jenkins when to initiate a build. The syntax here is essentially the same as that of the UNIX

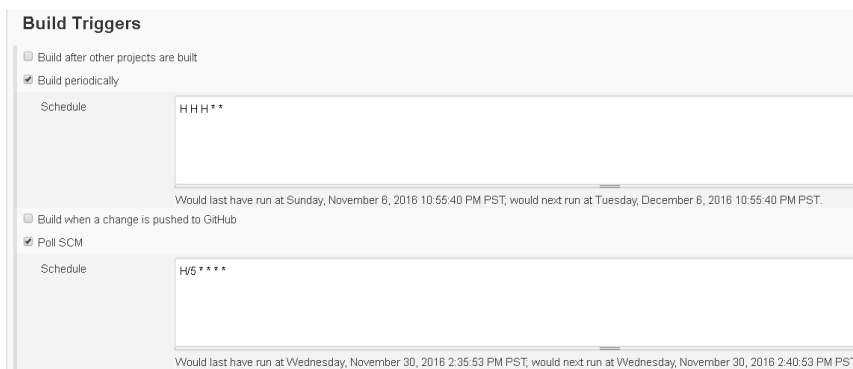


Figure 9. Build Triggers

cron command. In this example, a build will be initiated periodically once a day. In addition, Jenkins will also poll the SCM every 5 minutes to check for changes. Typically, it is a good idea to give a little window of time between polling, to grant a short grace period in case an engineer committed an erroneous check-in, or similar situation.

In the following ‘Build’ section, you tell Jenkins what to do when the triggers occur. Jenkins is capable of running just about anything you can think to throw at it, which for this example, will be to launch a set of regression tests. Most commonly you will be able to simply use the “Execute Shell” build step, and simply type the commands you wish to execute directly. Being able to setup builds and triggers is really Jenkins’ bread and butter. Additionally, you can manually execute a build anytime you would like by simply clicking a button on the project page, which you will see later.

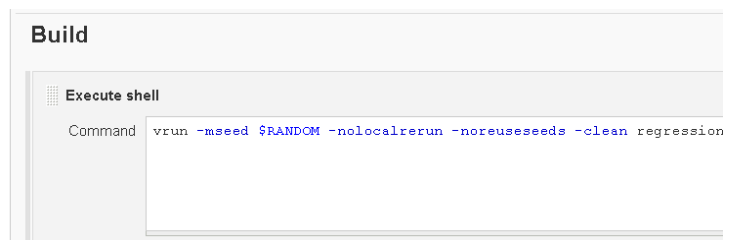


Figure 10. Build Command

Out of the box, Jenkins will give you basic pass/fail information, meaning, if you launch a script to run your regression, it will tell you whether or not your script passed or failed. It will also keep track of the history of your runs, including information such as the last stable build and changes made between builds (if you are using some form

of source code management. The final configuration step allows you to tell Jenkins to do something additional with the results of the build, once execution has completed. We will return to this later in the paper, for now this basic setup is all that is needed.

The project will now have a project page associated with it, and you can either wait for a build to trigger based on one of the build triggers, or manually by clicking the 'Build Now' button on the project page. Figure 11 shows this simple project, after it has been ran twice.

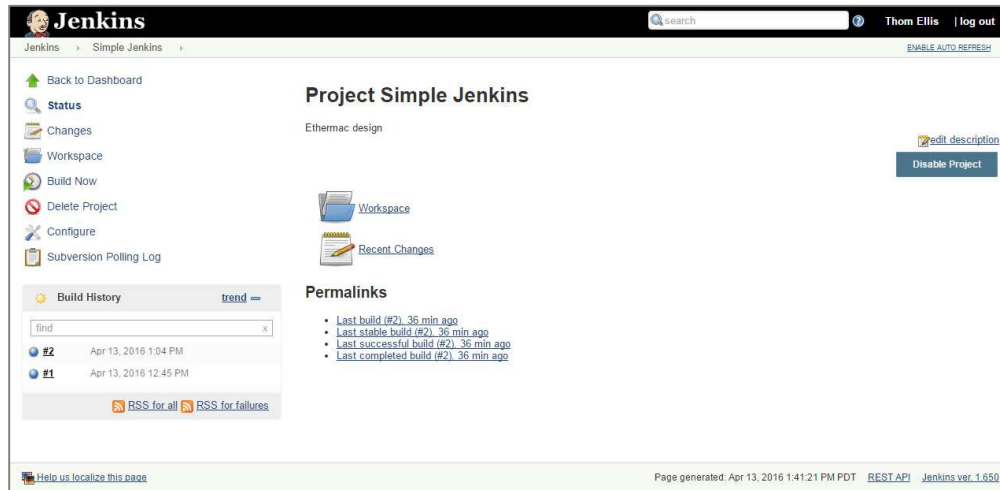


Figure 11. Jenkins Project Page

Straight away you can see that you get some basic information about each build. The ball to the left of the build number represents the status of that build. In Jenkins, blue is a pass (we will look at that a bit more later as well), yellow is unstable, and red is a failure. Access to some common information is also easily accessible. You can quickly get to the console output of the job, as well as see what triggered the build, and some other information on the build by clicking the corresponding links on the left-hand side of the project page.

By inspecting the console output, we can also see that this build was triggered by an SCM change, which shows both that our connection to our repository is setup correctly, and Jenkins is checking out those files successfully. As well as the fact that the SCM trigger that was set up is working correctly, so future check-ins will trigger a build, allowing us to start to leverage the benefits of Continuous Integration in our project.



Figure 12. SCM Triggered Build

Jenkins has built-in facilities to also send e-mails to users upon completion of a build so that users can be notified of the results. This way if there is an issue with the build, users can immediately be notified, and since check-ins are

happening early and often, and Jenkins is automating the builds, it makes it much easier to find when a new issue was introduced, and get the proper parties involved.

Now that we have something running, let us take a look at how we can customize the Jenkins instance to get more out of our results, and begin to better use Jenkins as a dashboard for our verification

V. CUSTOMIZING JENKINS

First, something simple to get us introduced to Jenkins plug-ins. Jenkins has its reasons for using blue balls to represent passes [5], however, there are enough people out there looking for green balls, that the community developed a plugin to allow a user to use green balls to represent passes instead.

We will use this simple plugin as our first example of how we can extend Jenkins' functionality via plugins. To do this, first return to the Jenkins dashboard and click on the 'Manage Jenkins' link to take you to the 'Manage Jenkins' page. Here you will see several links to customize and setup Jenkins specifically to your needs. If there is an update to Jenkins itself which has yet to be installed, you may also see a notification of that here as well.



Figure 13. Manage Jenkins

Next click on the 'Manage Plugins' link to open up the plugin manager. Here you can get a list of all the available plugins, as well as those which you already have installed. You may recall, during the initial setup, Jenkins installed a few common plugins for you already. Additionally, you can check here for updates to plugins which have been installed.

Navigate to the 'Available' tab, and type "green" in the filter box to locate the 'Green Balls' plugin:

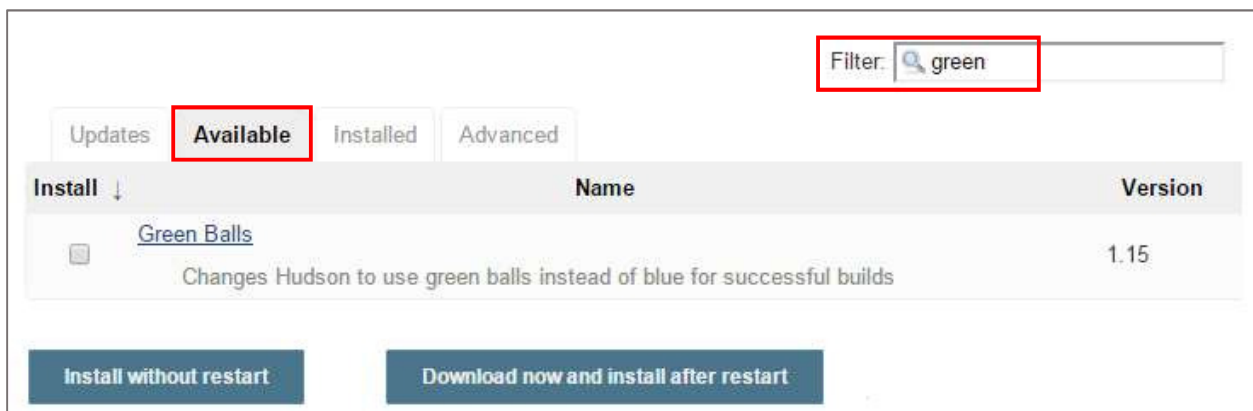


Figure 14. Installing a Plugin

Click the checkbox next to the plugin name to select it for installation, then click, “Download now and install after restart”. Jenkins will get the requisite files, and install them, and automatically restart your Jenkins instance for you. Once that is complete, when we navigate back to the project we created earlier, we can see that in fact, we now have green balls instead of blue to represent passing builds.



Figure 15. Green Balls Plugin

This plugin may not necessarily make your day-to-day job easier, but it serves the purpose to show you that installing plugins is very easy. Some will require a little more additional setup than the Green Balls plugin, however the installation process is always the same. One of the biggest areas we can benefit from a plugin, is by leveraging a post-build plugin. You may recall earlier, that this is the one step we skipped when configuring the project.

For our first post-build plugin, we will look at one with a bit more utility which will help to analyze regression results more quickly. Trying to read through the console output from a regression to find key messages, such as errors, can be quite burdensome, as often those types of messages do not really jump out to the user. We often add special characters, or formatting to try to make this easier, but in the end you are still left looking through a screen of text.

The Log Parser plugin [7] is a very useful plugin for this scenario. It is used to parse the console log output (the output from the job itself, not the individual tests), and enables the ability to highlight particular lines in the log, such as errors and warnings, so that at a glance we can see where problems occurred. It also can be used to display a summary of the errors, including links back to the location of those messages in the log, which means you do not have to go hunting back through the output manually. Finally, it also supports the ability to tag sections in your log, so that you can split up the log into logical chunks, for example you might have a compile and simulate section, with easy links to navigate directly to those areas in the log output.

The Log Parser plugin installs just like the Green Balls (only we search for ‘Log Parser’), however it operates slightly differently, in that it adds an additional option to the post-build actions section in the project configuration

page.

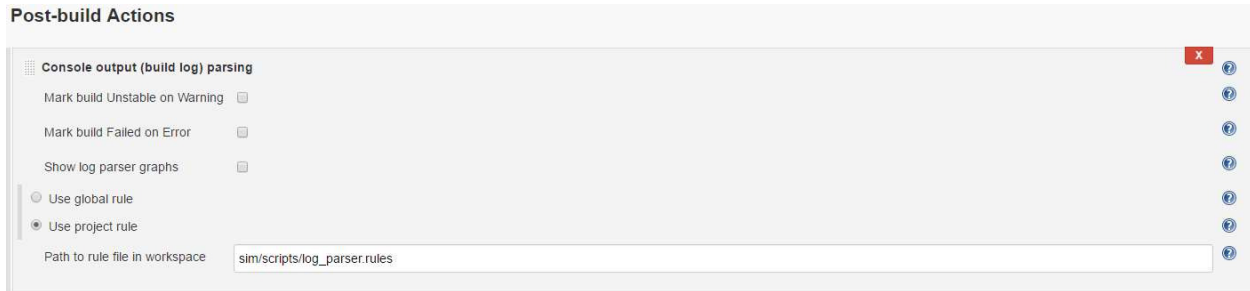


Figure 16. Log Parser Plugin

Based on the results of the parsed output, you can also have the Log Parser plugin modify the result of the build. For example if you want the presence of any error in a regression suite to mark a build as a failure, you can do that here (by default it will be marked as ‘unstable’ with a yellow ball). You can also graph a count of the messages that are parsed on the project page.

The main input required here is to specify a link to the rules files for which you want to parse the output with. The rules themselves are simply regular expressions which can match any line in the log output. Each expression also is assigned a “level” or type, which determines what color the message is highlighted, as well as how it is linked in the summary. For typical messages, there is “ok” (blue), “warn” (yellow), and “error” (red). There are additionally two other types, “info” and “start”, both of which are also blue, and create quick access links to areas in the report, for example to a final summary report in the log.

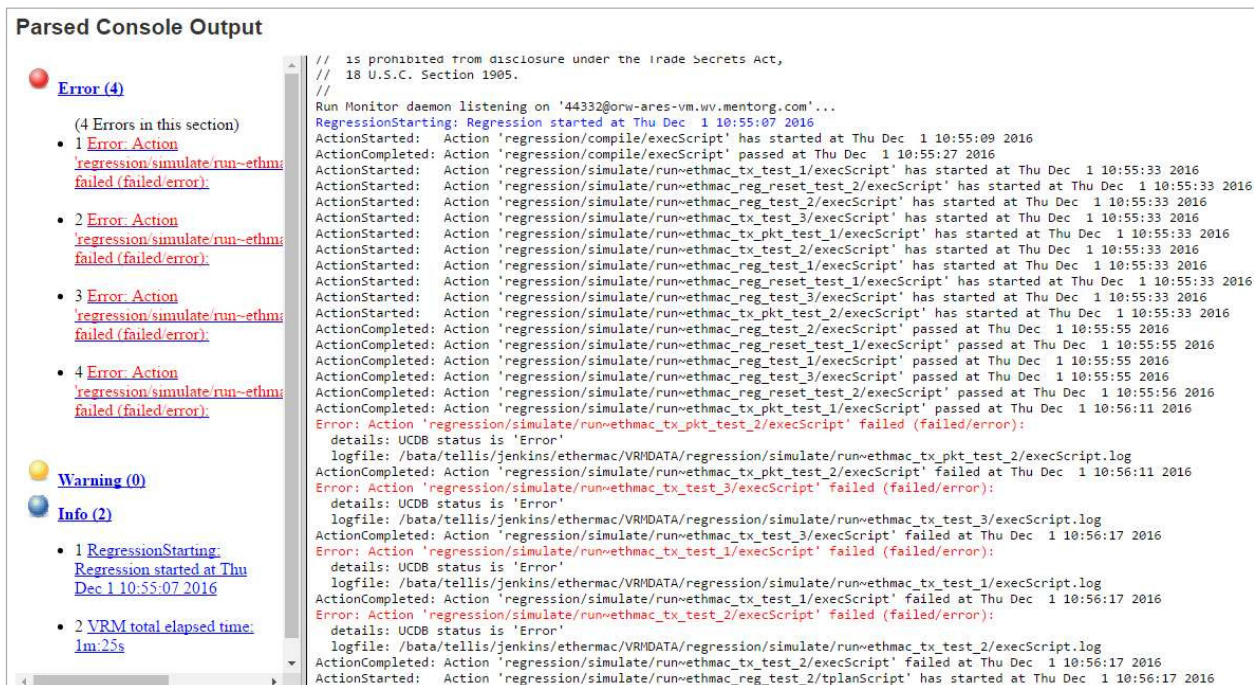


Figure 17. Parsed Console Output

Once setup, the log output now not only is highlighted making certain messages such as errors stand out, but we can easily get a list of all message types in the log, and quickly jump around to inspect the various errors should they occur.

One of the things you surely noticed from earlier, is that once we ran a regression, outside of basic pass/fail information, we did not get a lot of insight into the regression itself. For example, we do not know the results of individual tests (without inspecting log files, etc.). If we are in a coverage stage of our project, we cannot see how our coverage is progressing. The reason for this is that Jenkins itself, does not know how to interpret the results from a simulation, formal or emulation run. Being Java based, it does have some ability to understand Java test results, but for the most part, Jenkins is not focused on understanding the results of any tool it could come across. Instead, it provides a highly extensible infrastructure, which allows for plugins to be built to leverage the power of Jenkins.

There are many plugins which focus on this area, so chances are, there could already be one developed for the tool you are using. In our case, the regression we are launching is leveraging Questa's Verification Run Manager (VRM) tool, and there already happens to be a plugin.

Essentially, what the Questa VRM plugin does, is it understands the output of the regression. It can interpret coverage numbers, understand what constitutes a test, etc., so that it can collect that pertinent information, and package it up in a way that can be displayed in the Jenkins dashboard. It can dictate to create graphs of certain metrics, tables for other; link off to relevant reports, and a many other capabilities. The end result is getting much better visibility into the regression results themselves, including the ability to analyze trends, and dig into why tests failed.

Once you install the Questa VRM plugin [6] for Jenkins, the project configuration page will now have an additional option available in the post-build actions drop-down list to "Publish Questa VRM Regression Results":



Figure 18. Questa VRM Plugin

In this case, a little additional configuration is required, but it is fairly minimal. VRM is doing the bulk of the work still, in that it runs all the tests, merges the coverage results, etc. With that all being done, all that is required is that you tell Jenkins where the results of the regression will be placed, so that the plugin can locate the results, and package them in a manner that is consumable by Jenkins. Optionally, you can also tell the plugin that you would like to create

a link to a coverage HTML report on the project page, and/or if you would like Jenkins to keep trends of your coverage results as well.

You will not notice the changes until the next time you run the regression to trigger the plugins first execution, however, once you run again, you should see significantly more information on the project page for your regression. Here you can see all the additional information that becomes available once the plugin is installed.

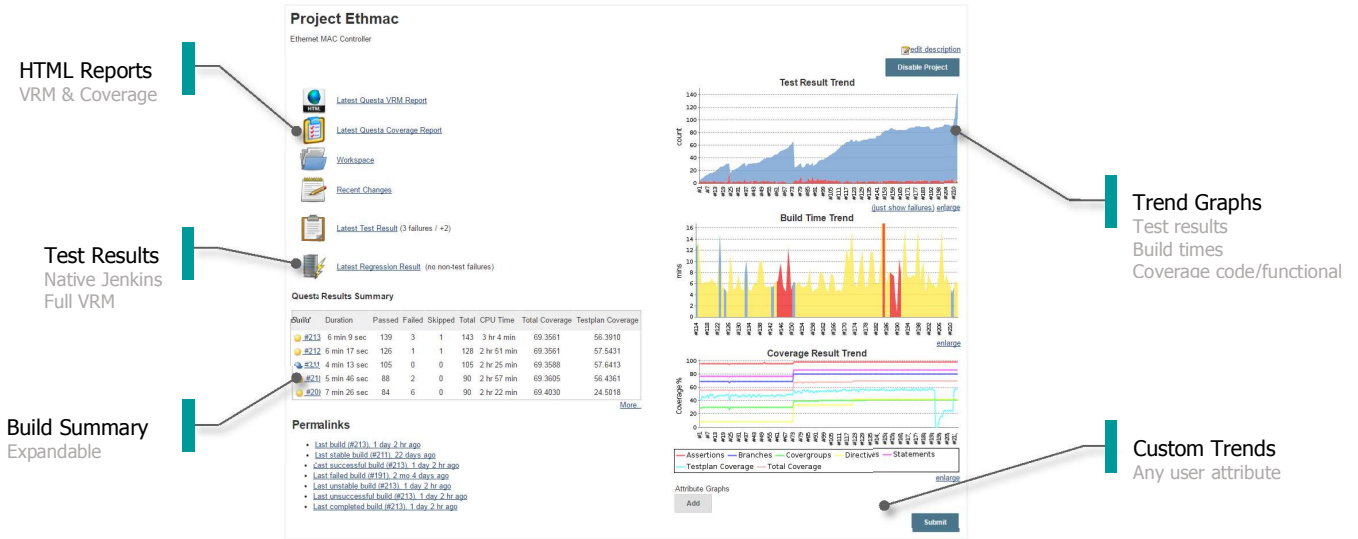


Figure 19. Project Page with VRM Plugin

Now all of the data that was already present from the regression, can be brought to the forefront. Coverage and test trends are easily visible, and can be tracked over the life of the project. Additionally, if we were trending any other metrics with the project, i.e. bug count, lines of code, etc., we could now add that as a customer trend graph to the project page.

In addition to the graphs, the plugin also produced an HTML report of both the regression and coverage results, which can be easily accessed directly from within the Jenkins dashboard, which allows for more efficient analysis. The project page now gives us a great view of the overall state of the project, however, the plugin also grants much more granularity to analyze our results on both a per-regression, and per-test basis.

Diving down into an individual regression, we can see the results of tests that were ran, as well as the coverage for that regression.

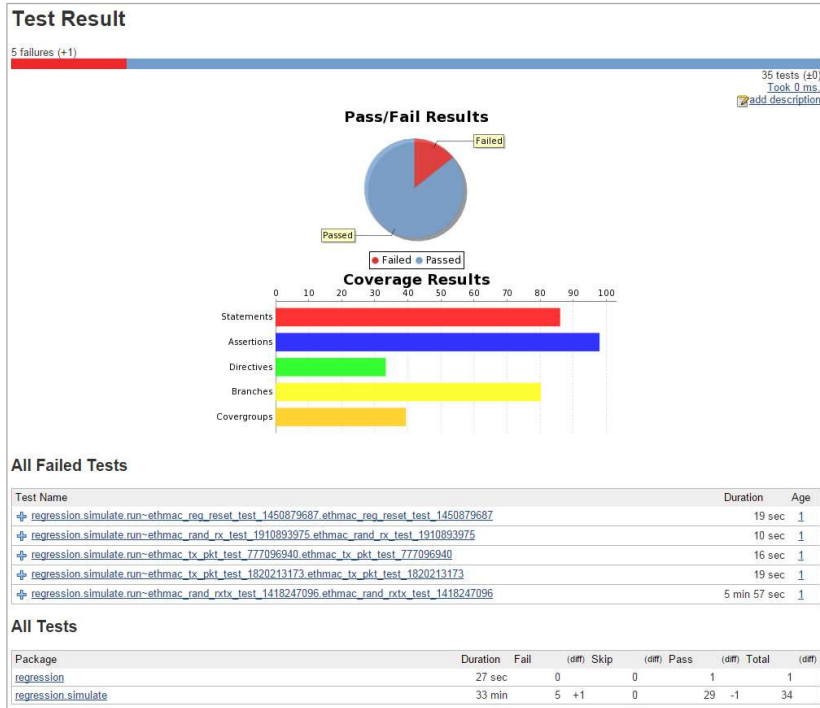


Figure 20. Regression Results

Failing tests are also broken out so that you can quickly and easily find what tests are having an issue. Clicking that test will dive down into the test itself, and you can quickly get access to the reason why that test failed. You can also inspect the log file for the test, as well as simulation statistics, attributes and coverage for the test.

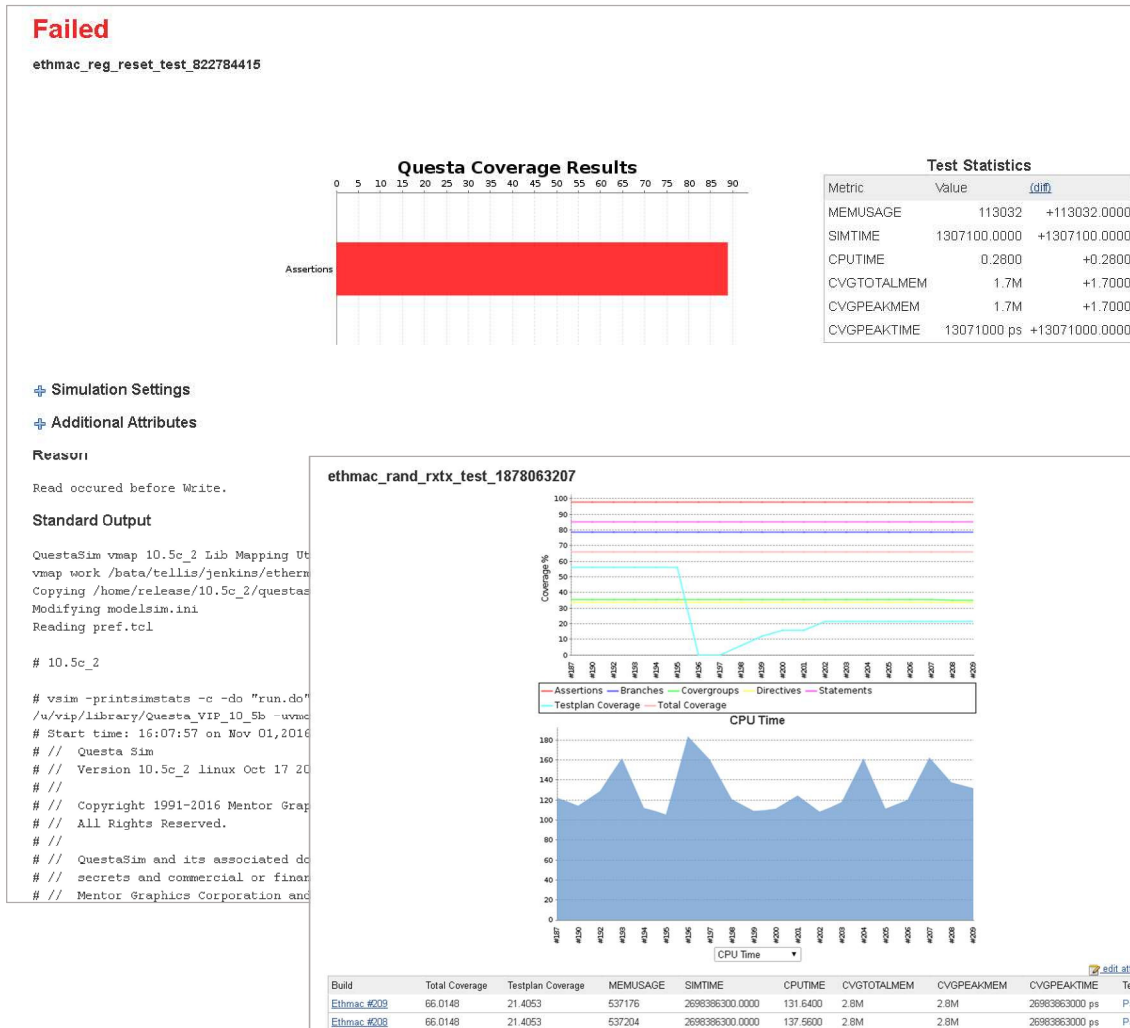


Figure 21. Test Result

Furthermore, not only is all of the data for each regression now available, but it will also be tracked historically, so you can see when a given tests started failing, when coverage dropped off, when simulation time doubled, etc. By simply adding this plugin, you can do a fair amount of analysis directly from within the Jenkins dashboard.

The last plugin we will look at, is the Dashboard View plugin []. Up to this point, everything we have looked at is with respect to one Jenkins project. However, what if we have several we need to monitor, and we do not want to always have to inspect each project independently, but rather get an overall view of where we stand. That is exactly what the Dashboard View plugin allows us to do.

The dashboard view plugin is setup slightly differently, in that it is accessed via the Jenkins start page, so that is where we need to set it up. On the start page, simply click the '+' to create a new view. Without the Dashboard View plugin installed, you



Figure 22. Adding a Dashboard View

would only have one option to create a new “List View”, however, with the plugin, a new option is available to create a “Dashboard”. Simply provide a name, and select the “Dashboard” option, and click “Ok” to begin setup.

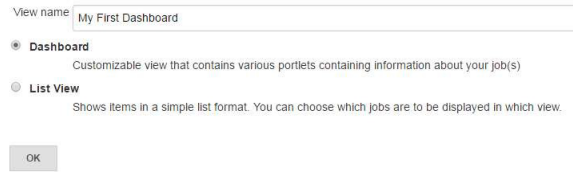


Figure 23. Creating a Dashboard

This will bring you dashboard configuration screen. There are a couple key sections to point out here. The first is the Job Filters section. This section determines what projects will be listed in the dashboard. You can selectively choose those which you are interested in, or optionally setup a regular expression to match against job names as well. Multiple filters can be created for more complex scenarios.

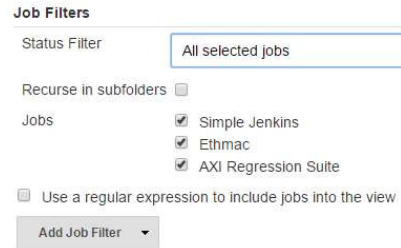


Figure 24. Jobs to Display in Dashboard

The next section is pretty self explanatory, it allows you to choose what information will be listed for each of the jobs selected (via columns in a table). Following that is the portlet section. This is where you can really customize what is displayed on your dashboard, and where it will go. The dashboard page is split up into the sections shown in figure 25. Each section can have multiple items displayed within it.

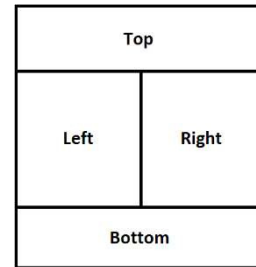


Figure 25. Dashboard Sections

There are several portlets which come with Jenkins by default which you can choose to display in your dashboard. In addition, other plugins can also provide portlets which can be displayed in the dashboard. The VRM plugin we looked at early is one example of this. It adds an additional “Questa Coveage Portlet” portlet as an option in the portlet view drop-down list. Each portlet will have different configuration options available as well which you can use to customize the look and feel to your personal preference. I would encourage you to try the various options, however, for the purpose of this paper, we will keep it simple, and use four portlets, all with their default configuration options set. The name of the portlets used becomes the title of that portlet, they are also highlighted in this figure for clarity.

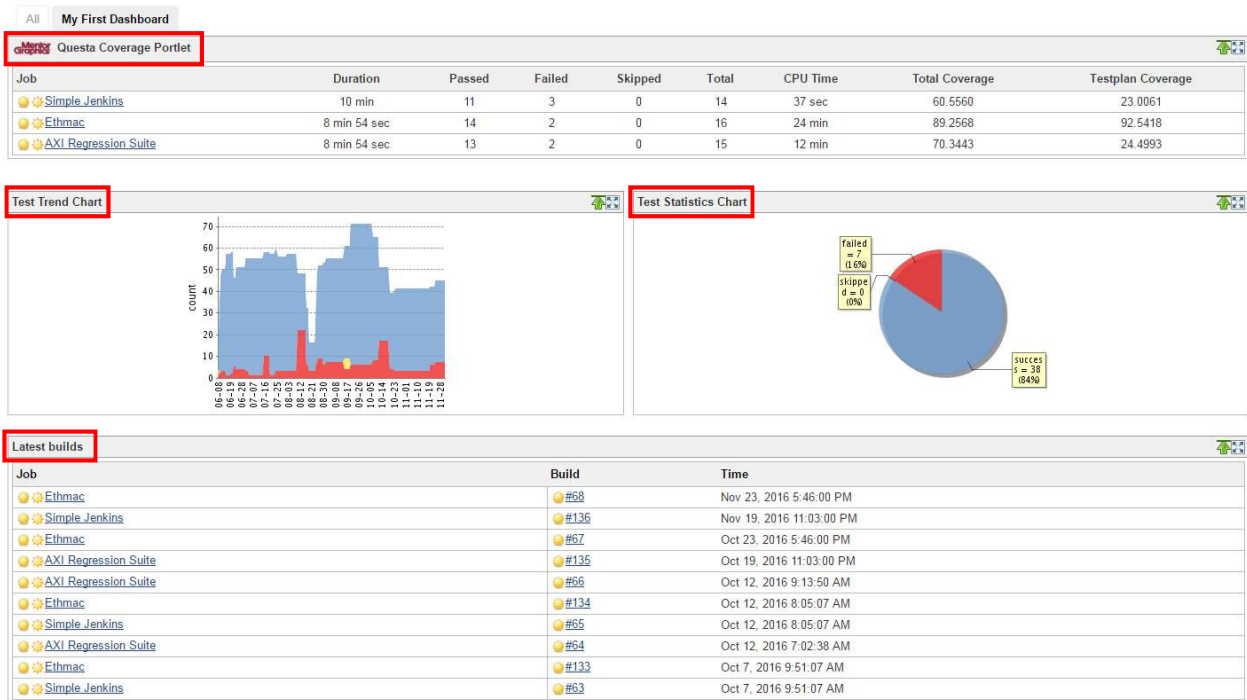


Figure 26. Completed Dashboard View

With this dashboard now complete, we can easily get a view of all projects in one location, providing more of a management level view. Clicking any link will allow us to dig down into the given project if we wish to do analysis on a given project. If we wanted to, we could also include the standard Jenkins list at the top of the page, which would additionally give us the ability to launch any given job from the dashboard as well. As you can see, this becomes a very powerful tool for viewing overall results.

At this point, we have really only scraped the surface of the over 1400 plugins Jenkins has available. Obviously, not all plugins will be applicable to all situations, but there are many that can help with productivity, analysis and debug of jobs.

VI. SUMMARY

Continuous Integration usage is on the rise, and Jenkins CI is one of the best tools available today to help you get started using CI today. It is incredibly easy to get setup, and running a regression quickly with minimal effort. Thanks to its vast library of plugins, you can easily customize Jenkins to suite your needs, and get the most out of your regression environment, allowing for verification engineers to make the most efficient use of the time they are given even in the tightest of schedules.

REFERENCES

- [1] Jenkins-stats, <http://stats.jenkins.io/>
- [2] Java Tools and Technologies Landscape Report 2016, <http://zereturnaround.com/rebellabs/java-tools-and-technologies-landscape-2016/>
- [3] Installing Jenkins, <https://wiki.jenkins-ci.org/display/JENKINS/Installing+Jenkins>

- [4] Jenkins, <https://jenkins.io/>
- [5] Why Does Jenkins Have Blue Balls?, <https://jenkins.io/blog/2012/03/13/why-does-jenkins-have-blue-balls/>
- [6] Questa VRM Plugin, <https://wiki.jenkins-ci.org/display/JENKINS/Questa+VRM+Plugin>
- [7] Log Parser Plugin, <https://wiki.jenkins-ci.org/display/JENKINS/Log+Parser+Plugin>
- [8] Dashboard View Plugin, <https://wiki.jenkins-ci.org/display/JENKINS/Dashboard+View>