Understanding and Applying Gerrit, Part 1

The Basic Code Review Workflow

Brent Laster

In this series of articles, we'll explore the basics of Gerrit – it's structure, *roles, and workflow, as well as it's integration abilities. We'll see how to use it for simple code-reviews and how to integrate it with Jenkins. We'll touch on it's extensibility as well through hooks and finally through Prolog programs that can govern many aspects of it's submit behavior (the final gate before code is merged into the remote repository).*

Introduction

Welcome to this series on Gerrit. Most people who have heard of Gerrit tend to think of it just as a code review tool. In fact, Gerrit itself is frequently referred to as Gerrit Code Review or even in some older contexts, just the formalized Code Review.

However, once you understand how Gerrit works and how it can be used, there is a lot more you can do with the tool. Like being able to hook in any tool or application you currently use as an automatic check that runs before your code is merged into your remote Git repository. The code in your remote repository will always be clean because it's already been validated. Additionally, you have a permissions and project structure that wraps each Git repository on the remote and can be inherited by other projects. And, finally, yes – there is that code review thing.

What is Gerrit?

At its core, Gerrit is a Git interface – an interface to remote Git repositories. It wraps Git and provides an extensive permissions layer organized by groups against references. (References essentially equate to git branches.) You add users to groups who have the permissions you want. Additionally, Gerrit holds changes between the local and remote repositories to allow validations and code review before things are merged into the remote.

Figure 1 below shows how Gerrit fits in with Git. **Part A** represents a typical Git system "stack" – with the working directory, staging area, and local and remote repositories. Code is moved through the various levels by the associated commands. **Part B** shows where Gerrit fits in – it provides an access layer and support for code review and validations – all in a project that wraps around a remote Gerrit repository.

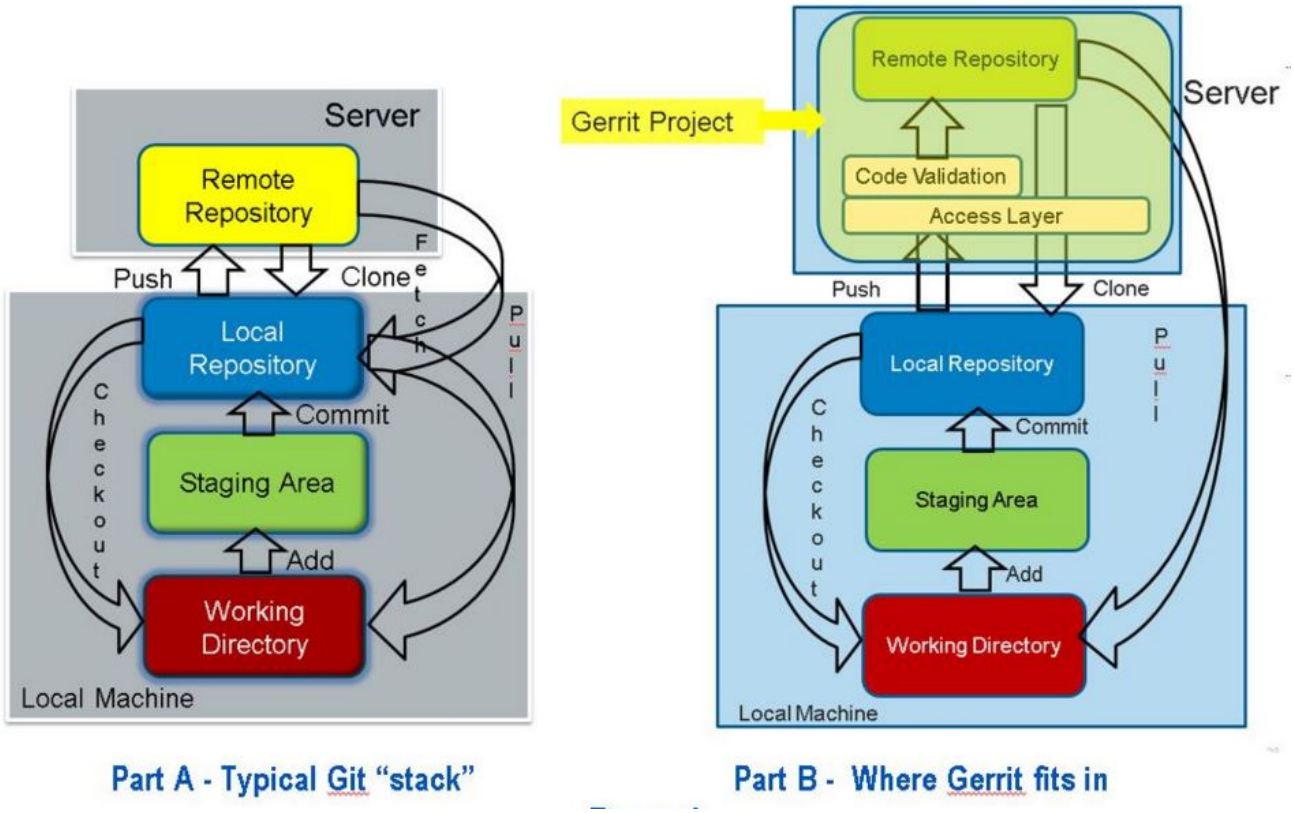

Figure 1. Where Gerrit fits in with Git

Getting Started

Getting Gerrit up and running from scratch is a bit of an involved process and beyond the scope of this series of articles. However, if you want to understand a little more about this, see the section on "Running out of the box" below. For the things we'll talk about here, I've setup a pre-configured very basic Gerrit system that you can download from GitHub to play with.

Alternate approach: Running out of the box

If you want a binary distribution for Gerrit, you may have to search a bit. A good place to look is at https://gerrit-releases.storage.googleapis.com/gerrit-<version>.war. As of this writing, the latest version available is 2.11.4. If you prefer instead to build it, be prepared for a bit of work. Since the 2.8 release, building Gerrit requires the build system Buck. ***Buck*** is a build system adapted from the Google proprietary "Blaze" system by ex-Google employees at Facebook. Buck is typically faster and more intelligent (as far as knowing what changed and what needs to be built) than Maven (the old way of building Gerrit). However, there don't seem to be binary distributions of Buck freely available, so be prepared to first build Buck with ANT. The process isn't necessarily tedious, but can take the better part of several hours getting everything setup and executed the first time through. For more information on how to accomplish all of this, see https://gerrit-review.googlesource.com/Documentation/dev-buck.html . From there, it's a matter of firing up the war file and running through an *init* sequence that creates the basic config file and template site and setting up any external database installations, etc. Then you can create users (the first user to log in is the Administrator), groups to provide access to projects, and the projects themselves.

For this series, you can grab the preconfigured Gerrit system at https://github.com/brentlaster/gerritworkshop/releases/download/1.1/gerritws2.zip to play with.

WARNING	**(Standard disclaimer – this is only for educational purposes – this is not intended to be used for any production work.)**

Just copy the file, and unzip its contents to a directory. From there, take a look at the ***gerrit_readme.txt*** file in the root directory. This file notes prerequisites – mainly Git and Java that you'll need. But also note the parts about the ssh keys and configuring your Java location in the config file.

IMPORTANT	For ssh access which the sample commands are based on, **you'll need to be setup for ssh access** as outlined in the readme file.

Once you are ready, you can start Gerrit either with the ***startgerrit.bat*** file on Windows or the ***bin/gerrit.sh start*** command on a Linux or Mac system. (On Windows, the session where startgerrit is initiated will need to be left running.)

If you see errors about *GuiceFilter*, you can ignore those – for our purposes things will still work. Beyond that, if you need to change any basic configuration items, most of them are in ***gerrit_reviewsite/etc/gerrit.config***. If you look at the file in ***gerrit_reviewsite/etc/gerrit.config*** one thing that may look a little strange is the authentication method –

DEVELOPMENT_BECOME_ANY_ACCOUNT. This is normally a debug setting for Gerrit development, but we're going to use it to allow us to experience all of the roles in the Gerrit Code Review process on our local instance. More about this in a moment. Once you start up Gerrit, you can go to the URL – http://localhost:8081 - to see the new instance.

Gerrit's Code Review Process

Roles

Typically you might have a login screen first and then after logging in, you'd see your userid on the far upper right-hand side. In this case, we instead have the word "*Become*" there. This is tied to the **DEVELOPMENT_BECOME_ANY_USER** setting. Essentially this will let us switch between any of several predefined users we'll use in learning about Gerrit. Click on the *Become* link and notice that we have four predefined users here:

Local Administrator
Spock (Contributor)
McCoy (Reviewer)
Kirk (Committer)

Local Administrator is pretty self-explanatory – administrator for the site. This is the user that has permissions to create projects, branches, etc. The other three correspond to predefined roles for Code Review that Gerrit uses. A ***Contributor*** is anyone that can contribute code for review or validation in Gerrit. A ***Reviewer*** (as the name implies) is anyone that can review code and provide feedback on it. And a ***Committer*** is someone who has the final say on whether the code is approved or vetoed. A ***Committer*** ultimately decides whether to move the change along and ask Gerrit to merge it in to the production code base in the remote repository – or to exclude it.

Clicking on any one of the users will switch to being that user in Gerrit. To switch to being someone else, click on the user name in the upper right corner and select the ***Switch Account*** link. To modify settings for a user, click on the user name and then ***Settings***. There's a bit of minimal setup we need to do for user settings for ***Spock***, ***McCoy***, and ***Kirk***. Namely, we need to give them email addresses.

Email Addresses and User Settings

Email addresses turn out to be remarkably important to Gerrit. Matching email addresses is how Gerrit ties a user who makes a commit to a user defined in Gerrit. More significantly, Gerrit is **CASE-SENSITIVE** for email addresses! So the email address a user sets locally when you do a

```
$ git config user.email <email address>
```

must match **EXACTLY** the email address provided for the user in Gerrit.

For our purposes here, we're actually going to take advantage of this eccentricity to allow us to play with all three roles. Here's how that works. ***Spock*** will be our main user here – the one who makes the commits locally in Git that become changes in Gerrit – the ***contributor***. We want ***Spock*** to have the exact same email address on the Gerrit side as the email address that will be used in doing commits locally.

IMPORTANT	Switch to being Spock.

Then click on his name in the upper right corner, and select ***Settings***. Click on the ***Contact information*** on the left hand side menu and click the ***Register New Email*** button. Enter the email address that matches exactly the email address you'll be using for commits in Git. Click the ***Register*** button. While you're here, you can select the ***SSH Public Keys*** menu item and notice that there's already a key here. If we needed to add another one, you could just dump out the public key contents in a terminal and paste them in to the text box. Or you could add one via the command line ssh interface (as part of a user setup) like so:

```
$ cat <public key file> | ssh –p 29418 Spock@<gerrit-server> gerrit create-account –full-
name "'<name>'" –-email <email address> --ssh-key - <userid>
```

NOTE	(Note the single quotes inside of double quotes around the full name string since it contains a space.)

Also on this page at the bottom of the side menu is a way to create an http password if you will be using Gerrit over http instead of/in addition to via ssh. A word of caution here – using this method to generate an http password results in a password that is a random combination of letters. To have something reasonable, set an http password through the ***–http-password*** option of the ssh ***create-account*** command, or if after the initial creation, there's a corresponding ***set-account*** command you can use to update settings.

Now we need to provide email addresses for ***McCoy*** and ***Kirk***. For these two, we'll use the fact that Gerrit is case-sensitive to trick it into using our one email address for all three users. This will allow us to see what kinds of email Gerrit sends for each role.

IMPORTANT	Switch to being McCoy.

TIP	Via the *Switch Account* option off of the popup when you click on the user.

Select ***Settings***, then ***Contact Information*** again, and ***Register a new email address*** for him. Here's the trick: use the same email address as you did for Spock, **but** make it a different case. For example if you used some mixed case on the one for Spock, you can make the one for ***McCoy*** all lower-cased. Make a note of this one as we will need it later.

IMPORTANT Switch to being Kirk.

Do the same thing for ***Kirk*** but use a third version of the email address that differs from the other two by case (for example, perhaps all upper-cased). Make a note of this one too.

Creating Projects

Now that the users are setup in the different roles, we just need a project to work with. In Gerrit, a project maps to one and only one Git repository. Additionally, a Gerrit project has access controls defined, and metadata about changes. It's also possible to have a project without a Git repository in Gerrit. Such projects are typically used to define a set of permissions that other projects can inherit from - a template.

There is a default project named ***All-Projects*** that exists in every Gerrit system. This is the starting project that provides default accesses for other projects to inherit from. As well, this project is the one that allows for setting accesses for system administration functions such as database access. These can be administered by the first user to log into the Gerrit system. That user gets *admin* permissions by default. (More about permissions to come in a future article.)

It's worth taking a moment at this point to talk about the main Gerrit screen and interface. (If not already there, jump back to http://localhost:8081) Other than an initial login screen, the first screen you will see is the Gerrit *dashboard*. (See Figure 2 below.) On this screen, you'll see things presented in terms of reviews, since Gerrit is best known for code reviews. You can think of the different sections here as being like your email. ***Outgoing*** are things you're working on or things you've sent to someone else requesting them to review. ***Incoming*** are things someone else has requested you to look at. ***Recently closed*** are any changes that have either been successfully merged all the way into the remote repository **or** ones that were abandoned for some reason (code became obsolete, too many negative comments, you changed your mind, etc.).

There are also the menus across the top. For some items, selecting an item on the top menu line causes the second line to change (*My* vs. *All* for example).

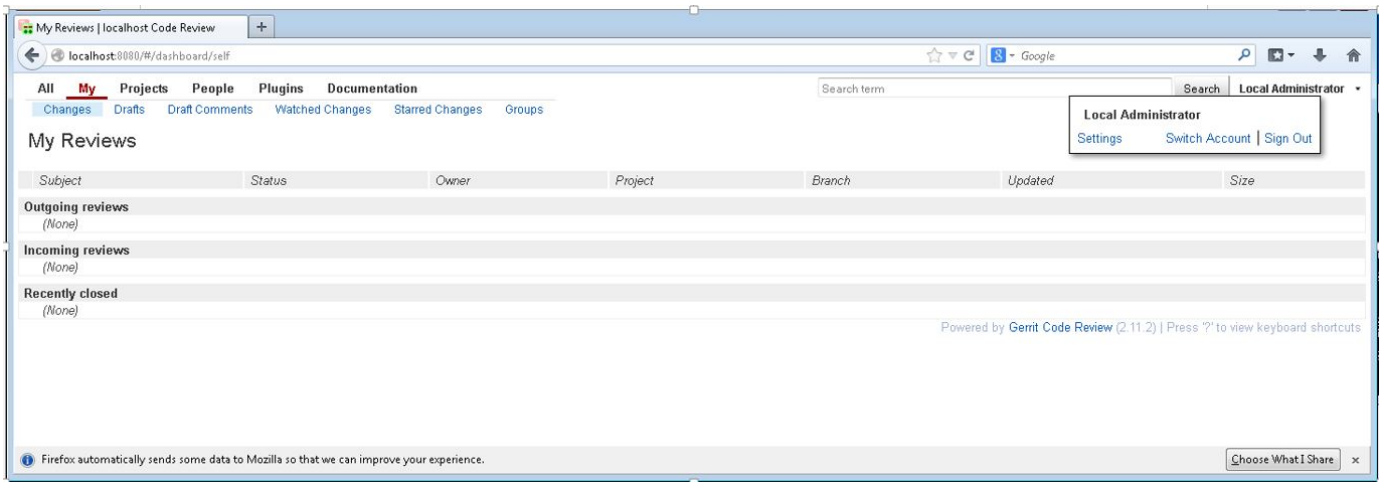

Figure 2. The Gerrit "dashboard"

NOTE

The Gerrit Interface

While it has gotten better in recent releases, Gerrit does not have the most intuitive interface. Gerrit was originally created from earlier systems for the Google Android project. Over the years, a number of other companies have contributed to it. The change interface got a major facelift in recent versions allowing everything to fit on one page.

In this preconfigured instance, the ***Local Administrator*** is the only user that can create projects.

IMPORTANT Become (switch to) the Local Administrator.

Select ***Projects***. Click on the ***Create New Project*** link. Type in a name for the project and select ***Gerrit Basics Template project*** for the parent project for ***Rights Inherit From***. (That's just an empty project(no Git repository) that we've setup to have some useful permissions to inherit.) Also click the ***Create initial empty commit*** checkbox. This will give us a starter commit in the project when we clone it. The reason not to do that would be if we were were going to migrate an existing Git repository into the Gerrit system. Now click the ***Create Project*** button.

Interacting with Gerrit through the local Git environment

Local Setup

IMPORTANT Switch to being Spock.

Clone this project down to your local system. Gerrit helpfully provides the clone command on the ***General*** page of the project we just created. Go to the project's General page at http://localhost:8081/#/admin/projects/<project_name>.

Near the top of the page, you'll find a line starting with "clone" that provides the actual commands you need to clone the project. Select "**clone with commit-msg hook**". An important part of this is choosing the right protocol further to the right in that line. For our purposes here, choose ***SSH***. **Copy the command that Gerrit shows in the line below that one and paste it into a terminal to clone the project down to our local system.** After the clone completes, change into the cloned project's directory.

Figure 3. Project's General Page - note prefilled clone command

Now, we're ready to start creating content for Gerrit. From the point of view of Git, Gerrit is a remote repository that we push to. We can switch to a terminal and configure the git username and email. Be sure here to set the email address to the exact same one used for *Spock* in Gerrit.

```
$ git config --global user.name "<first last>"
```

```
$ git config --global user.email <email address used for Spock>
```

Change-Ids

We're now almost ready to start pushing content. Before we do, there's one more piece we need to talk about that most Gerrit instances expect – a ***Change-Id***. A *Change-Id* is a special string inserted as the last line of each **commit message**. Its purpose is to let Gerrit know if a commit should create a new change in Gerrit **or** if the commit is meant to be an update to an existing one.

NOTE

How does a Change-Id work?

Requiring a *Change-Id* is a configurable option in Gerrit, but its one that is almost always used. It looks a lot like a SHA1. It is inserted by a special hook that must be "installed" in the hooks area of each local repository you want to use with Gerrit. The hook is the ***commit-msg*** one that might normally be used to validate commit messages. In the use case for Gerrit, whenever a new message is supplied, the hook gets invoked and puts a new ***Change-Id*** line at the bottom of the commit message. Then when the commit is pushed to Gerrit, Gerrit examines the ***Change-Id*** line to determine if it needs to create a new change or update an existing one.

So, we need to have this hook installed locally for **each** repository we want to use with Gerrit. You may have noticed we used the "***clone with commit-msg hook***" entry from the project's ***General*** page

earlier. This option prepopulates the commands to do the clone from Gerrit and the copy command to get the hook "installed" immediately after the clone. But if you had just done the clone or ever find yourself needing to do this by hand, here's the separate command to copy the hook from the gerrit server to your local area.

```
$ scp –p –P 29418 Spock@localhost:hooks/commit-msg .git/hooks/
```

Gerrit Workflow

Creating a Change

Now, we have all the setup done. In your terminal session, create a new branch to work in. We do this as a best practice because we don't want to work in the same branch we're going to be pushing for. The reason can be generally stated as wanting to keep the corresponding local branch "clean" so we can sync with the Git remote. If we were to develop in the same target branch locally and then our changes in Gerrit didn't pass or got held up for review, we could be in a difficult place if we wanted/needed to sync up with the target branch on the remote. So instead, we develop in a different local branch, push our changes for the target branch, and, once they are reviewed and approved, rebase them back to our dev branch.

```
$ git checkout –b devbranch
```

Now, create a file, stage it, and commit it to your local Git repository. Here's a shortcut way to do this – we use the echo simply to have some content in a file.

```
$ echo content > file1.txt && git add . && git commit –m "first change"
```

Do a **git log -1** to look at that commit. You should have a change id line at the bottom of the commit message.

Pushing to Gerrit

The next step in the normal git workflow would be to push this over to our Git remote. This is where we make a change to steer the push to Gerrit first. The way we do this is by pushing to a special target location – **refs/for/<destination branch>**. Here's the command (using master as the designated "target" branch this time):

```
$ git push origin HEAD:refs/for/master
```

Here's how to read this command: **"git push origin"** is our standard push to the remote reference **"origin"** that we cloned the project from. **"HEAD"** on the left side of the **":"** here refers to a pointer for

a SHA1 that is the latest commit on the current branch. This is just a shortcut for pushing the last thing we committed in the current branch. The ***refs/for/master*** portion on the right side of the **":"** refers to the remote branch. Normally we would just have a single name that corresponded to the remote branch here. But, by saying **refs/for/master**, the remote push is detected as intended for Gerrit. In fact, it is interpreted as "I'm pushing this to a temporary reference (branch) in Gerrit and if it passes all of the checks and code review in Gerrit, then it can be submitted for merging into the remote's master branch."

CAUTION	A ":<remote branch>" without anything on the left-hand side of the ":" is Git syntax for "delete the remote branch". So, if you happen to have permission to do that kind of operation, be careful with your typing.

Go ahead and do the push if you haven't. You should see a response from Gerrit that indicates your push was successful and shows you a URL for the new change that got created. Essentially, Gerrit has now captured this commit and created a new change with one revision (called a **"patchset"**). Gerrit is holding it, pending a successful code review, before allowing it to be merged into the master branch in the remote repository.

The Change Screen

Back in the browser, open up this change either by going directly to the URL that Gerrit provided in the push output or by selecting **My** and then **Changes** in the menus. This will take you to a change screen for the change you just created (similar to Figure 4 below.) There are a number of things to note on this screen, such as the overall status of the change in the upper left corner, the list of files associated with the change near the middle of the screen, the running log of activity at the bottom, and, near the center of the screen, under the "Reply..." button, a place to add Reviewers.

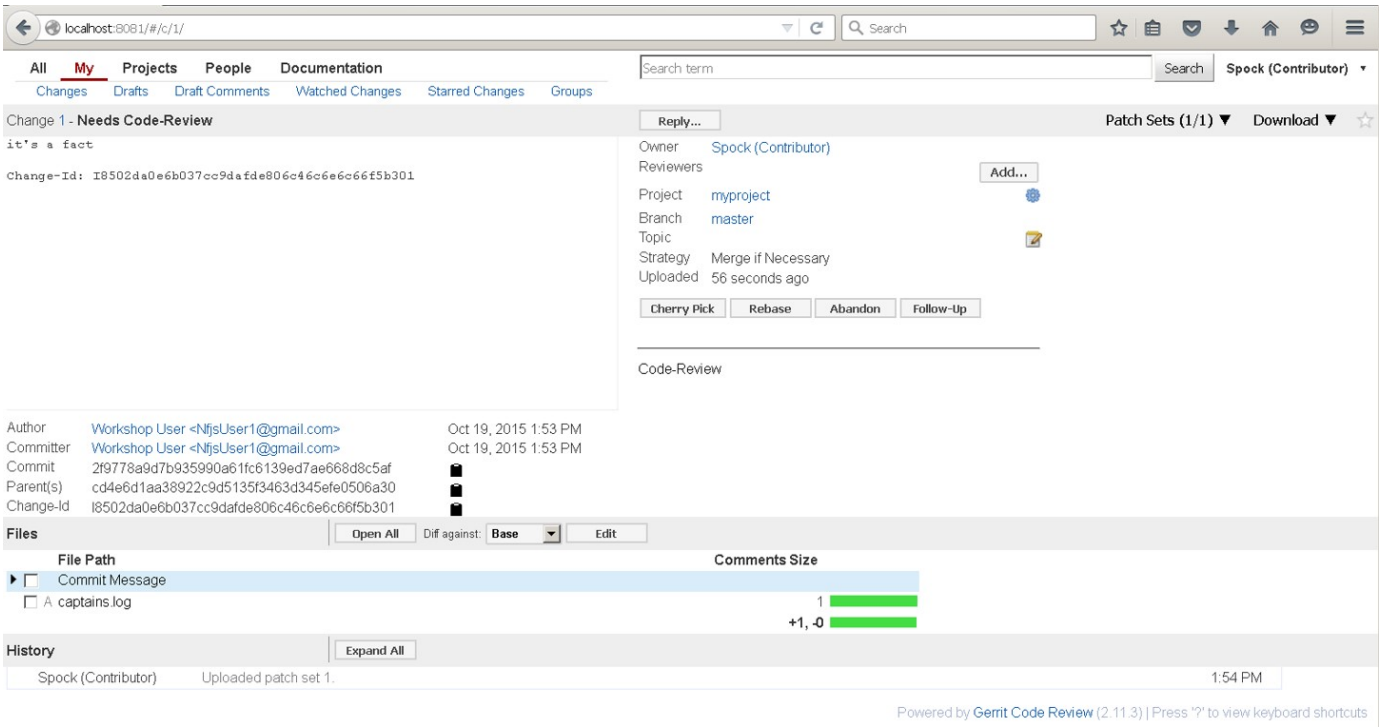

Figure 4. Example Gerrit Change screen

Review Labels

Below that section, near the center of the screen, you'll see the words **"Code-Review"**. This is an example of what Gerrit calls a ***Review Label***. You can think of a review label as a category of check that we want to do against the code in Gerrit. Out of the box, Code-Review is the only default label that is included. As the name implies, this is the indicator that the code needs to be reviewed in Gerrit in order to be submitted for merging with the production branch. The other one that is typically setup is **"Verified"**. This is used for functional verification through builds and, optionally, quick tests of the code. Most commonly this translates to having a Jenkins system hooked into Gerrit such than any code pushed to Gerrit is automatically built by Jenkins. (We'll see how to hook up Jenkins in another article in this series.) Other custom labels for desired checks can be added.

NOTE

Review Labels

Review labels such as **Code Review** and **Verified** define categories of checks/validations to be done in Gerrit. This implies that these validations can pass or fail. Gerrit uses a numeric scoring scheme to indicate pass or fail. The outcome of each check with positive values indicates a result is good, negative values indicate a result is bad, and 0 generally is used to pass back information without a judgement of success or failure.

For the Verified label, if Jenkins is able to successfully build the latest code pushed into Gerrit, it tells Gerrit that it succeeded by posting a **+1** score in the Verified category. If the build fails, it posts a **-1** score back to Gerrit. For users designated as Reviewers (like *McCoy*) they can provide feedback in the Code Review category in the range of -1 (to indicate they think the code is not ready) to +1 (to indicate they think the code is ok). Committers (like Kirk) have an extended range for the Code Review label. For them, -2 is a veto (the code will not go in), -1 to +1 is the same as for a Reviewer, and +2 is the golden ticket that says the code is approved and is going to be submitted for merging. So, it takes a +2 from a Committer to get the code submitted for merging.

Reviewer Workflow

Go ahead and click the **Add** button in the **Reviewers** row and add *McCoy* as a Reviewer (start typing his name and select it when filled in. Click the second **Add** button if needed).

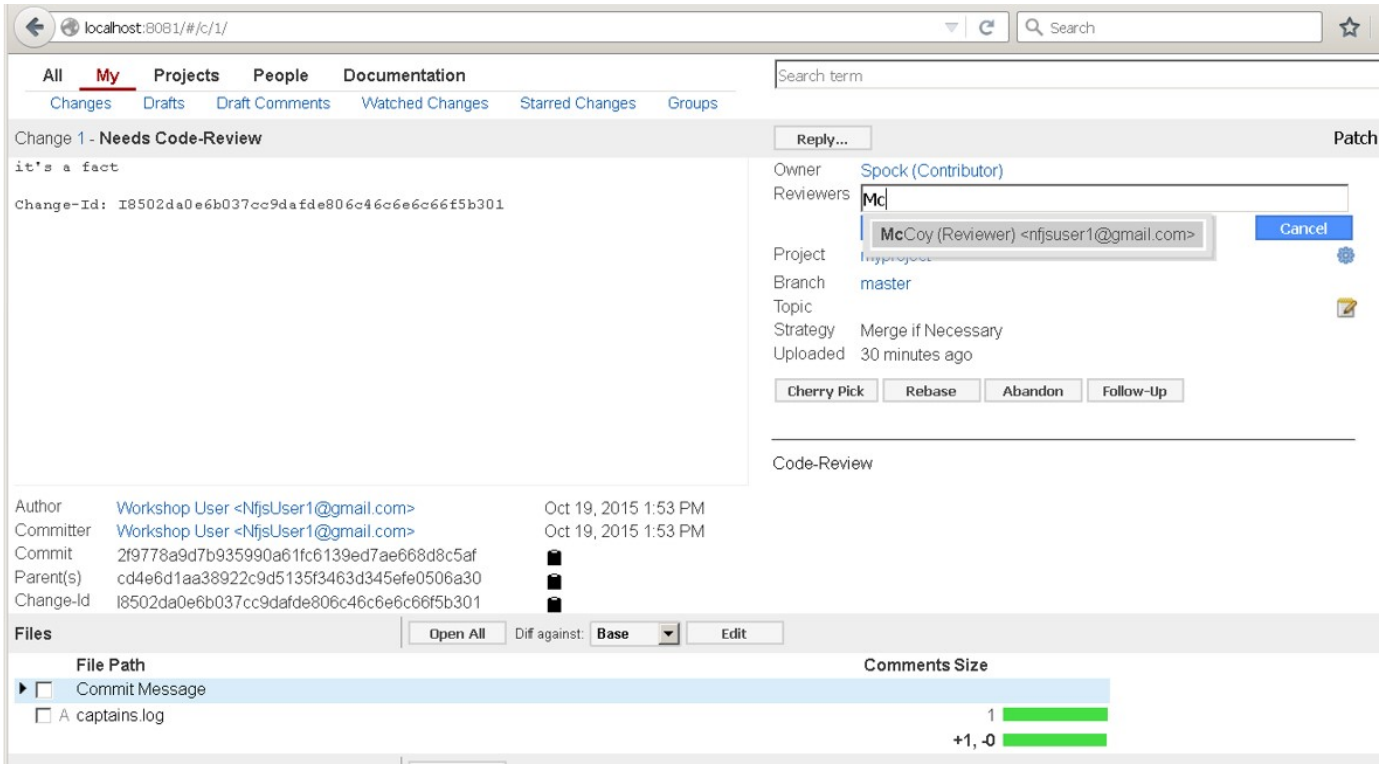

Figure 5. Adding a Reviewer

Go back to the main Changes screen **(My→Changes)** and notice that as *Spock*, this change is now in my **Outgoing** area – going out to *McCoy* for review.

IMPORTANT	Switch users (upper right corner → Switch Accounts) to become *McCoy* .

Notice that as *McCoy* , the review shows up in your **Incoming** area. Open up the incoming review by clicking on it. As *McCoy* , our job is to review *Spock's* change. We'll simulate a simple review here that finds a problem. Double-click on the name of the file in the file list to open up the code review/diff screen. Since there wasn't a previous version, there's nothing to diff against. But let's add a comment in the file to simulate feedback. You can add a comment by one of 3 methods: click on the line number in the file, use the "c" keyboard shortcut, or click the little icon that looks like a yellow post-it note with a green circle on it. In the comment box that comes up, type in some comment like "change" or whatever you want to say. Then hit the **Save** button on the dialog box to save your changes.

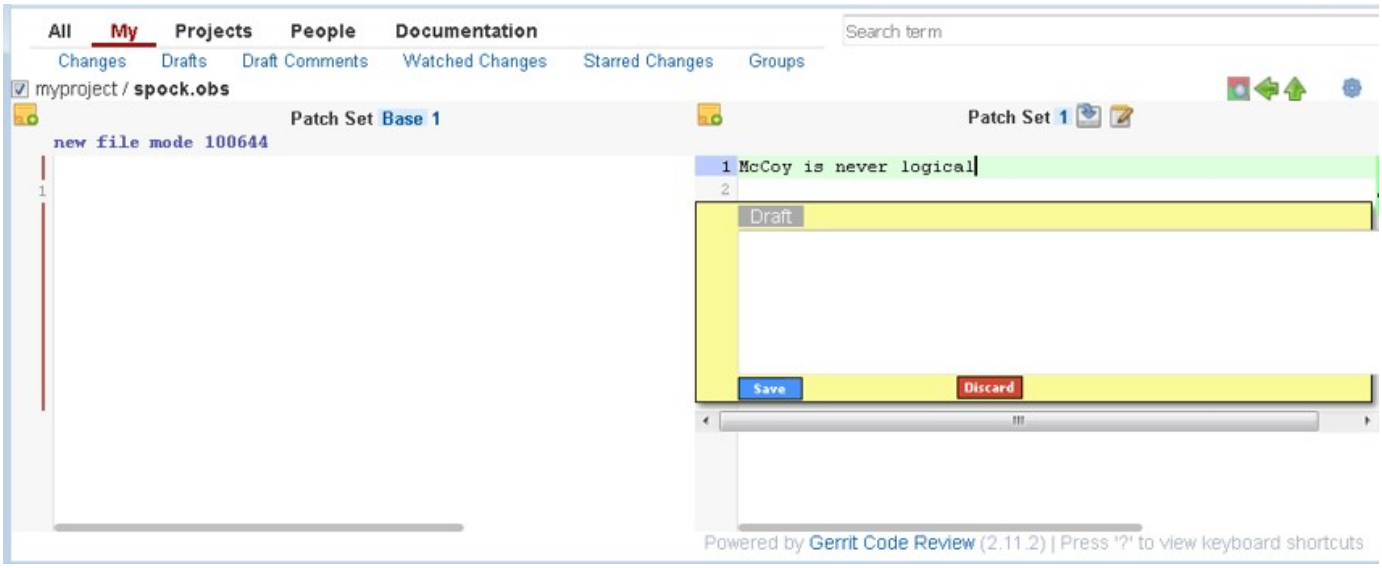

Figure 6. Creating draft comments

To get back to the main change screen, you can either use the “u” keyboard shortcut or click on the upward pointing green arrow in the upper left section of the page. (To see all keyboard shortcuts in Gerrit, just hit the "?" key.)

Back on the change screen, notice that in the line next to the file, it shows there is 1 draft comment in red text. Any comments you make are considered as drafts until you indicate you’re done with your overall review and Post your feedback. To post your feedback, click on the **Reply...** button near the center of the gray bar at the top. We’re going to see how to deal with a negative review, so in the scoring area, select -1 for the value. You can put in a general overall comment in the box if you want. Then select **Post** to finish the review. This will make your comments published instead of draft.

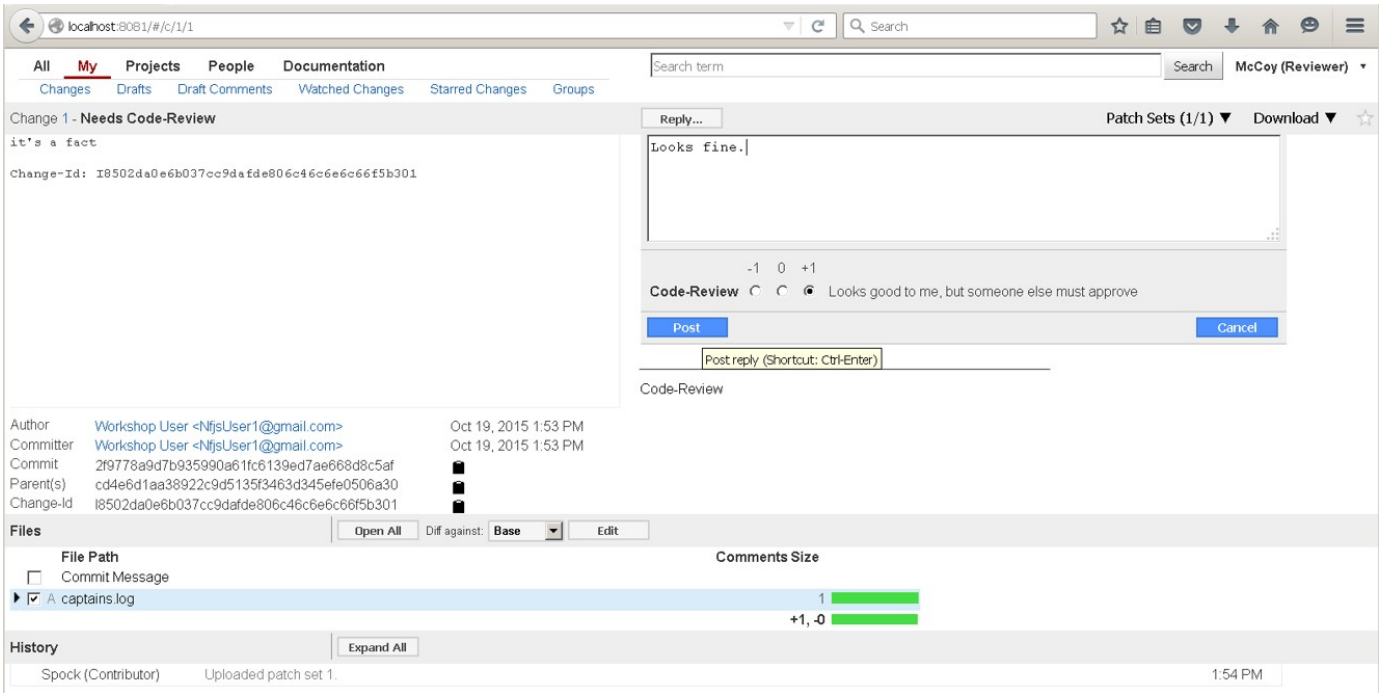

Figure 7. Finalizing a review and publishing comments

Responding to a Review

If you haven't already, this is a good point to open up your email and look at the kind of emails that Gerrit sends out. Notice it can be a bit chatty.

IMPORTANT	Switch back to being Spock, and open up the Incoming change again.

Since *McCoy* didn't like the content, we'll make an update. Click on the file name, then click on the comment, and hit **Reply**. Fill in some text and save it. Then go back to the change window, hit **Reply**, and **Post** your comments. (You can use a score of 0.)

Dealing with Dependencies - Multiple Changes vs. Multiple Patchsets

Now we need to update the actual content. Go back to the terminal, and make anotjher change, stage and commit it and push it out.

```
$ echo update > file1.txt && git commit –am "change 2" && git push origin
HEAD:refs/for/master
```

Notice the output you get from Gerrit. It indicates another Gerrit change was created. Switch to the browser, and refresh it or open up the newest change. Notice on the right-hand side of the screen, Gerrit says we have **"Related Changes"**. This isn't really what we wanted because we have now introduced a dependency on the first (bad) change. We would have to do something with that first change before we could get the newest one through.

What we really should have done here was create an updated revision (new patchset) for the existing change. To correct this situation, let's get rid of this latest change – not the first one. To do that in Gerrit, find the **"Abandon"** button on the **most recent** change screen (change 2) and click it. Add a comment if you wish and click the second **"Abandon"** button on the dialog. The change will now be abandoned in Gerrit and will show in the **"Recently Closed"** section on the dashboard. Abandoned changes can be restored if needed.

We're back to our original change in Gerrit, but we're still one ahead on the local Git side. Let's correct that. Switch back to the local Git session. Before we do anything, do a **"git log"** and take a look at the commit messages for the last two commits. Notice that they each have different Change-Ids in them. This is the reason Gerrit created a new change instead of updating the existing one. And the new *Change-Id* got inserted because we committed a second time supplying a message (**git commit –am**) which caused the commit-msg hook to fire and insert a new Change-Id. We need to "rollback" our local repository to the state before the last commit. In Git, we can effectively do a rollback by using the **reset** command. Execute this:

```
$ git reset --hard HEAD^
```

This command says move the HEAD pointer back to one before the current HEAD and update the

staging area and working directory (--hard) to be consistent with the new HEAD. You can do a **git log** here to confirm where we're at. Make a simple change and stage it, BUT don't commit it yet.

```
$ echo update > file1.txt  &&  git add .
```

Now, we'll do the commit in a way to cause Gerrit to create a new revision (patchset) for the existing change rather than a completely new change that introduces a dependency. We do this by using the ***--amend*** option on the commit. This is the trick. By using the **--amend**, Git will not fire the commit-msg hook (since no message is supplied). Rather, the amend essentially says "update the last commit with whatever's in the staging area and bring up the editor to let me type in a new comment". This pulls in whatever change in content we have put in the staging area AND allows us to keep the same *Change-Id* from the original commit in the editor. This is the key – not changing the Change-Id. By keeping the same one as before, Gerrit recognizes that this commit is just an update for the existing change and not a new one.

```
$ git commit --amend
```

Make any changes to the part of the commit message that you want **EXCEPT to the *Change-Id*** line. Save the file, close the editor. Push with the same command as before.

```
$ git push origin HEAD:refs/for/master
```

Switch back to the Gerrit interface, go to the screen for the original change (change 1), refresh if needed, and notice that there's a patchset section in the upper right quadrant. We now have 2 patchsets for this change instead of 2 separate changes. (You can switch between patchsets using this dropdown.) Gerrit will preserve all patchset associated with a change for purposes of diffing and comparison, but only the latest one makes sense to review and ultimately try to merge. Ensure you are on the screen for patchset 2. (URL should indicate something like localhost:8081/#/c/1/2).

The Committer Workflow

We could send this back to *McCoy* for review again, but for simplicity here, since we've already addressed his concern, we'll just add *Kirk* as a committer so he can hopefully move this along. In the Reviewers row, add *Kirk* as you did *McCoy* earlier. Ensure this is completed before proceeding.

IMPORTANT	Switch accounts to become Kirk.

As Kirk, open up the review. Open up the single file in the change and look at it. If you want, you can make comments in it. When done, use the keyboard shortcut or the green up arrow to go back to the main change window. As a committer, *Kirk* can override anyone else's scoring and either veto or approve this change. In this case, *Kirk* is ok with the code as-is. We can again go through the **Reply** and **Post** process and supply a **+2** as our score. However, as a committer, Gerrit provides a shortcut – a

button labeled **“Code-Review +2”** in the gray bar at the top. To expedite things we’ll use that. So, click on that button. By having a Committer issue a score of +2, we are indicating that this change is approved. Now click on the **Submit** button. In Gerrit, submit refers to telling Gerrit to try and merge the code into the remote repository. Note that just as with any push to a remote repository that is not a fast-forward, there can be merge issues. If there are, then essentially, the developer has to create a new patchset for the change that addresses the merge issues and the review process starts over again with that patchset.

In this case though, since this is our first change, there shouldn’t be any merge conflicts. So notice that now our change shows a status of **“Merged”** in the top left. It has now been merged into the remote repository’s master branch. If we switch back to being Spock, we’ll see that the change shows up in the **“Recently Closed”** section now on our dashboard.

Conclusion

We’ve now gone through the process of getting a local instance of Gerrit up and running with multiple user roles and taking a change with multiple revisions through the review process to being merged into the production code base. In our next installment, we’ll talk about merge types in Gerrit and dealing with multiple commits.

About the Author

I’ve been involved in the software industry for over 25 years, holding various technical and management positions. Over my time in the industry, much has changed, but one constant is the need for those in the business to grow their skills and keep up with ever-changing technologies and paradigms.

To that end, I’ve always tried to make time to learn and develop both technical and leadership skills and share them with others. In the early days, I taught community college classes on topics like Lotus and early versions of Windows while working as a software developer by day.

Fast forward quite a few years and more recently, I’ve been fortunate enough to have a chance to explore and train others in technologies like Git, Gerrit, Gradle and Jenkins as part of my job managing a group focusing on developer productivity and emerging technologies. Regardless of the topic or technology, there’s no substitute for the excitement and sense of potential that come from providing others with the knowledge they need to help them accomplish their goals.

In my spare time, I hang out with my wife Anne-Marie, 4 children and a small dog in Cary, North Carolina and volunteer in local Cub Scout and Boy Scout organizations.