



DIFFY

The Kung Fu Review Cuckoo

Community Experience Distilled

Learning Gerrit Code Review

Leverage the power of Gerrit Code Review to make software development more cooperative and social

Luca Milanesio

[PACKT] open source*
PUBLISHING community experience distilled

Learning Gerrit Code Review

Leverage the power of Gerrit Code Review to make software development more cooperative and social

Luca Milanesio



BIRMINGHAM - MUMBAI

Learning Gerrit Code Review

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: August 2013

Production Reference: 2260813

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78328-947-9

www.packtpub.com

Cover Image by Sarah Owens (sarato@inkylabs.com)

Credits

Author

Luca Milanesio

Project Coordinator

Amigya Khurana

Reviewers

Edwin Kempin

Fredrik Luthander

Proofreader

Sarah Heath

Acquisition Editor

Pramila Balan

Indexer

Tejal R. Soni

Commissioning Editor

Mohammed Fahad

Production Coordinator

Conidon Miranda

Technical Editors

Dipika Gaonkar

Sonali S. Vernekar

Cover Work

Conidon Miranda

About the Author

Luca Milanesio is the Director and cofounder of GerritForge, the leading Git and Gerrit competence center for the enterprise. His background includes over 20 years of experience in development management, software configuration management, and software development lifecycle in large enterprises worldwide. Just prior to GerritForge LLP, Luca was the Technical Director and Senior Product Executive of the Security and Compliance platform for Electronic Payments at Primeur in Italy and UK. Since starting GerritForge LLP, Luca contributed to the Gerrit community and allowed the introduction of enterprise Code Review workflow in large enterprises worldwide including major telecoms, banks, and industries.

Luca Milanesio is the author of *Git Patterns and Anti-Patterns*, an essential guide for scaling Git to the enterprise with 16 patterns and anti-patterns, including Hybrid SCM, Git champions, blessed repository, per-feature topic branches, and ALM integration.

I would like to thank Shawn Pearce and Google Inc. for having created and funded the Gerrit Code Review project, which allowed the Android operating system to be really the result of a collaborative effort of multiple companies and developers, and inspired many more development teams in the adoption of Code Review. Many thanks to the people that worked around the writing and publication of this book: Edwin and Fredrik for the extensive content review, Sarah for the copyediting, Mohammed the Commissioning Editor, Amigya the Project Coordinator, and all the other Packt staff that participated in the making of this book.

I would like to give a special thanks, and dedicate this book to my beloved wife Emilija and my daughter Nina, who have inspired and supported me throughout the days and nights while writing, and for making every single day of my life worth living.

About the Reviewers

Edwin Kempin is one of the maintainers of the Gerrit Code Review project. He joined the Gerrit development in 2010.

Since 2004 Edwin is working as a software engineer at SAP AG. As a member of the Java development infrastructure team, his main focus is the adoption of Git and Gerrit at SAP AG.

I would like to thank the Gerrit community for all the fantastic collaboration. Each day it's exciting to work on this project!

Fredrik Luthander has been working with software configuration management since 2000 and with Gerrit since 2009. He has worked mainly as an SCM systems admin and trainer. He has also been involved in designing and teaching/rolling out CM practices with both large and small software projects.

Fredrik is currently a self-employed SCM consultant, with prior experience from two enterprise-sized companies and their software projects. He is involved as a contributor in the Gerrit open source project since 2010.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read, and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Chapter 1: Introducing Code Review	7
Benefits of Code Review	7
Build stability	8
Knowledge sharing	9
External early feedback	10
Shared code style	11
Team engagement	11
Qualitative code selection	12
Code Review roles	13
Contributor	13
Reviewer	13
Committer	14
Maintainer	14
Review labels and roles customization	15
Review terms and workflow	15
Project	15
Change	16
Label Code Review	17
Submit change	17
Merge change	18
Abandon change	18
Summary	19
Chapter 2: Setting Up and Quick-start	21
Pre-requisites	21
Gerrit download	22

Running Gerrit initial setup	22
Installation completed	26
Log in and create user profile	26
Generate HTTP password for Git	28
Create and clone your first project	30
Summary	32
Chapter 3: User Authentication	33
How Gerrit user authentication works	33
Git versus Gerrit UI authentication	34
Gerrit internal accounts	35
Authenticating over the Internet through OpenID	36
OpenID SSO	39
Using in-house private Gerrit authentication (LDAP)	41
LDAP configuration steps	41
Read-only LDAP user profile	44
Active Directory	44
Third-party authentication options	45
Example – Apache HTTP frontend	46
Setting up Gerrit behind a reverse proxy	46
Enabling Gerrit HTTP authentication	47
User profile lookup	47
Summary	48
Chapter 4: SSH and HTTPS Access	49
Enabling strong security on Gerrit	49
Installing Bouncy Castle Security	50
Using SSH with Gerrit	51
Git/SSH Client keys	51
Adding SSH keys to a Gerrit user profile	52
Cloning a repo over Git/SSH	53
Enabling HTTPS	55
HTTP/S reverse-proxy to Gerrit	55
HTTP/S Support in Gerrit	57
Summary	58
Chapter 5: Editing Your Project Permissions	59
Understanding the Gerrit permission scheme	59
Configuring project permissions	60
All-Projects permissions	61
Git and Gerrit references	62
Git permissions	62
Code Review permissions	63

Managing groups	64
Custom internal groups	65
External groups (group backends)	66
Using group hierarchy effectively	67
Organizing project security templates	68
Summary	70
Chapter 6: Changes and Code Review Workflow	71
Gerrit Code Review roles and workflow	71
Review workflow step-by-step	72
Roles overview	72
Workflow in ten steps	73
Gerrit branch namespace for code review	74
Setting topics	76
Adding reviewers	76
Review labels	77
Review	78
Commenting and scoring changes	78
Review etiquette	79
Publish review and scoring	80
Amending code under review	81
Summary	83
Chapter 7: Submit Types and Concurrency	85
Submit types	85
Fast forward only	86
Rebase if necessary	86
Merge if necessary / always merge	87
Cherry pick	88
Concurrent code reviews	89
Changes dependency tracking	89
Real-life change dependencies with Gerrit	90
Navigating through the Gerrit change dependency graph	92
Managing change graph updates	93
Summary	94
Appendix A: Using Gerrit with GitHub	95
GitHub workflow	95
GitHub controversy	96
GitHub authentication	96
Building the GitHub plugin	97
Installing GitHub OAuth library	98
Installing GitHub plugin	98

Register Gerrit as a GitHub OAuth application	98
Running Gerrit init to configure GitHub OAuth	99
Using GitHub login for Gerrit	100
Replication to GitHub	101
Configure Gerrit replication plugin	101
Authorizing Gerrit to push to GitHub	102
Start working with Gerrit replication	102
Reviewing and merging to GitHub branches	103
Using Gerrit and GitHub on http://gerrithub.io	104
Summary	105
Appendix B: Automation with Jenkins	107
Gerrit review workflow with Jenkins	107
Setting up the Jenkins Gerrit plugin	108
Installing Jenkins plugins	108
Setting up the Gerrit trigger plugin	109
Getting Gerrit ready for Jenkins	109
Gerrit trigger plugin configuration	110
Triggering a build from Gerrit	110
Configuring the Gerrit trigger	110
Configuring the Git plugin	111
Automated code validation	111
Summary	112
Appendix C: Git Basics	113
Peer-to-peer distributed version-control system	113
Git installation	113
Creating the first hello world repository	115
Hello world file archived in Git	115
Displaying repository history	116
Editing and archiving a new file change	116
Git basic concepts	117
Working with Git branches	117
Fast forward, merge, rebase, and cherry pick	119
Working with remote Git repositories	120
Pushing branches to the remote repository	121
Fetching from remote repositories	122
Summary	123
Index	125

Preface

Today, developing software is more than a globally distributed activity: agile methodologies that worked well enough with co-located teams, need to be empowered with additional tools, in order to allow the developers to share, discuss, and cooperate more socially.

With the rise of the adoption of Git, a version control system designed by *Linus Torvalds* in 2005 for the Linux open source project, major steps have been made to get developers to work and share code in a more open way. GitHub and Gerrit Code Review are the cornerstones of the social coding revolution, having driven respectively the success of the most popular open source code repository and the development of Android, the most successful mobile OS open source project.

This book looks at the Code Review workflow's benefits in an agile development team, and provides simple steps for leveraging them. It guides through the installation steps of Gerrit by showing the most typical setup and configuration schemes used in private networks and over the Internet. It includes a simple set of recipes and samples on how to configure and use it in conjunction with your existing authentication and continuous integration systems. By following this book's instruction, you will be able to install a fully configured Gerrit server ready to be used for your development team.

You will also learn how to effectively use Gerrit with GitHub, in order to have a more consistent Code Review, in addition to the social collaboration tools provided by the GitHub platform. Using the two tools together, you will be able to reuse your existing accounts and integrate your GitHub community into the development lifecycle, keeping in touch with external contributors.

Learning Gerrit Code Review will teach you everything you need to know to install, configure, and to start using Gerrit Code Review in your daily development. It will introduce the concept and the spirit of collaboration and collective code ownership with co-located and remote collaborators.

What this book covers

Chapter 1, Introducing Code Review, introduces the basic concepts and terminology of Gerrit Code Review, along with its benefits, roles, and responsibilities of the development team.

Chapter 2, Setting Up and Quick-start, explains in detail the Gerrit setup and configuration of an initial sandbox environment. We will take our first steps with Gerrit by creating an initial administrator account, one repository, and we will also start pushing our first change for review.

Chapter 3, User Authentication, explains how Gerrit integrates with existing user registries for performing authentication and user's profile registration. We will understand how the authentication process works, and what are the possibilities for configuring it to connect to LDAP, OpenID, or any other third party authentication system.

Chapter 4, SSH and HTTPS Access, explains how to enforce the security of our Gerrit installation using stronger security protocols, such as SSH and HTTPS to control the Code Review actions from a Git client.

Chapter 5, Editing Your Project Permissions, explains us how to effectively use project templates and groups, in order to organize role-based access for our repositories. In order to start organizing the projects and teams for a Code Review, we will need to explore the Gerrit permissions scheme and the association of ACLs (Access Control Lists) to Git ref-specs.

Chapter 6, Changes and Code Review Workflow, provides an overview of how Code Review workflow works with Gerrit by going step-by-step through an initial change review. We will explain the jargon used by reviewers and how to interact with the reviewers and contributors.

Chapter 7, Merge Strategies and Concurrency, analyzes the different situations of concurrent dependent changes under review, and how Gerrit helps you to manage them with the project merge strategies.

Appendix A, Using Gerrit with GitHub, provides a step-by-step example of how to connect Gerrit to an external GitHub server, and provides guidance on how to use the Gerrit Code Review workflow and GitHub side by side.

Appendix B, Automation with Jenkins, takes us through the steps needed to configure Jenkins for fetching code directly from Gerrit changes during the review phase, enabling the automation of a series of validations against the code.

Appendix C, Git basics, provides the reader with a basic knowledge of the concepts and common terms of the Git version control system, which allows beginners to quickly get set up and use Git.

What you need for this book

This book is a guide to set up and use Gerrit Code Review and assumes that you have already installed a server with the following characteristics:

- Linux server (recommended) or Windows server with Cygwin extensions
- Oracle Java JDK 6 or later
- Git version 1.6 or later
- Apache HTTP server 2.2 or later (optional)
- Jenkins 1.424 or later (optional)

Alternatively, the book can be used as a reference on how to use a Gerrit instance of an open source project available on the Internet, for example, <https://gerrit-review.googlesource.com>, or available for private and public projects on the cloud, for example, <http://gerrithub.io>.

Who this book is for

This book is for team leaders, developers, or SCM managers who are willing to improve collaboration by introducing Gerrit Code Review.

A basic knowledge of the Git version control and its commands is required. Appendix C provides the reader with the basic Git concepts and commands, while the book provides a rich set of Gerrit Code Review examples and configuration recipes ready to be used for getting started quickly and effectively.

Conventions


In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.


Code words in text are shown as follows: "We can create then our first change on the `hello-project` we cloned in our initial sandbox."

Any command-line input or output is written as follows:

```
$ echo "Change C" > change-c.txt
$ git add change-c.txt
$ git commit -m "Change C"
[...]
$ git push origin HEAD:refs/for/master
[...]
remote: New Changes:
remote:  https://myhost.mydomain.com:8443/10
remote:
To ssh://jdoe@myhost.mydomain.com:29418/hello-project
* [new branch]      HEAD -> refs/for/master
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen."

[ Warnings or important notes appear in a box like this.]

[ Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Introducing Code Review

Welcome to the world of Code Review! In this chapter we will introduce the basic concepts of Gerrit Code Review, along with the roles and responsibilities of the development team and the actions involved when reviewing the changes pushed to the repository. We will cover the benefits of the Code Review, not only for the overall quality of the development but also the positive impacts on the entire team dynamics, collaboration and interactions between its members, and even its external counterparts. Basic terminology and a glossary of Gerrit Code Review will also be introduced in order to allow an easier and smoother introduction of the Code Review lifecycle, workflow, phases, actors, and the associated actions and concepts. By the end of this chapter we will put in place the basics for making the first steps with Gerrit.

Benefits of Code Review

The words "code" and "review" are sometimes a bit misleading when we try to use them to understand the fundamentals of modern Code Review. The Wikipedia definition says "Code Review is systematic examination (often known as peer review) of computer source code". This can lead to the thinking that there are two elements characterizing this operation: systematic and examination.

Historically developers believed that by being systematic it was easily expressed in terms of "coding rules" and automated through static and dynamic code analysis. Tools such as Sonar have been designed to support the definition and automation of such rules, which are then checked and validated together with your Continuous Integration environment.

Modern Code Review, however, in addition to traditional code inspection, encourages positive dynamics amongst people and facilitates collaboration, both of which cannot be replaced by automated and defined rules and processes. The most positive result of a Code Review cycle is, in addition to higher code quality standards,, it is about getting different people to look at the version of the solution and having a constructive discussion around it.

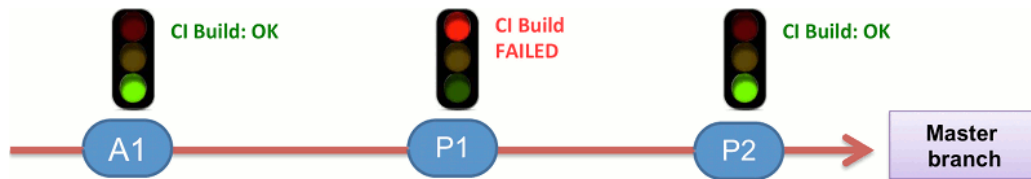
Code Review becomes part of the development process as a fundamental step and becomes a prerequisite for merging the code into its target branch.

Build stability

When adopting Continuous Integration, your goal is to get the code into your version control as soon as possible so that it can be checked in and validated with the whole project: in return you get a **RAG (red-amber-green)** status of your change. Whenever a build is not green, it is defined as "broken" and it is the team's priority to re-establish a healthy head version of the branch by fixing the build.

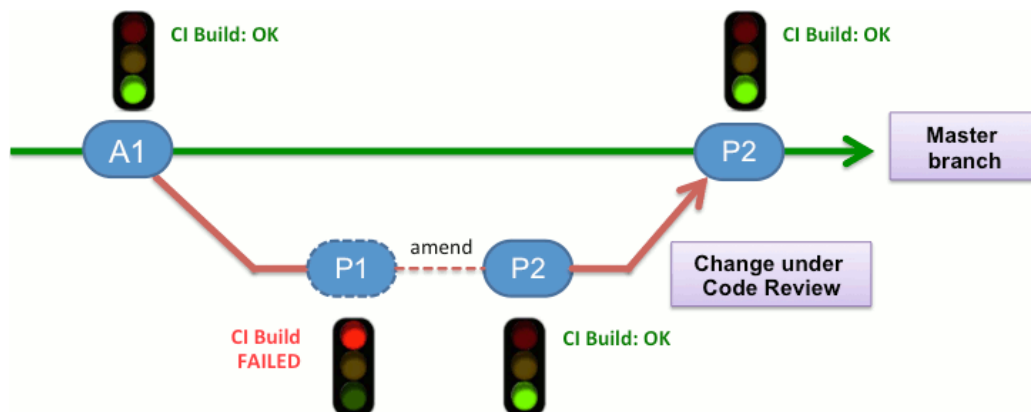
Code Review helps by reducing broken builds, thanks to the ability to link the review status to a validation step, resulting in less time wasted by the development team.

The following diagram shows Continuous Integration without Code Review:



The preceding diagram shows the typical stability of the build when every commit is directly pushed to the master branch. Whenever an incorrect patch (P1) is pushed, the build fails causing the team to intervene to fix it, which results in them pushing a new patch (P2) to re-establish a green build again. The master branch will keep a record of the failure by having both P1 and P2 in the history of the master branch.

The following diagram shows Continuous Integration with Code Review:



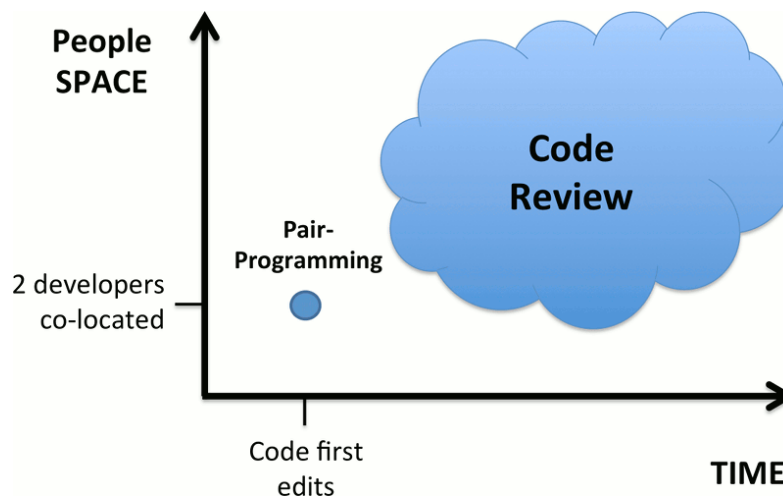
As shown in the preceding diagram, the introduction of Code Review allows you to check the sanity of the change in isolation, without interfering with the normal integration flow of the functionalities and keeping a green build. Even if a faulty patch is uploaded (P1), it causes a build failure only in the review branch: this can be fixed as before by submitting a new amended commit (P2) that will succeed and thus will be merged into the master.

The benefits are twofold: firstly, the normal branch stability has not been impacted and secondly, the Code Review branch has allowed the pre-validation and clean-up of the code for allowing a consistent history without losing track of the review activity and the continuous integration feedback.

Knowledge sharing

Whenever more than one person is looking at the code, the knowledge of the solution starts spreading among the team. Historically this was achieved by applying Pair Programming, one of the rules of Extreme Programming. The interaction between the "pair" leads to an earlier discussion on the solution from different perspectives with the result of a more shared understanding of the code base. This approach, however, is limited by time and space: the pair has to sit in front of the same desk at the same time, a condition that often is hard to achieve with distributed teams, which is a common set up with open source projects.

The following diagram shows Pair Programming versus Code Review:



Code Review, a practice of "team programming" is where the code is shared with potentially any member of the development team, each one free to look at it at different times beyond the code first edits timelines. The preceding diagram shows the difference in time, space, and people of the involvement of Pair-Programming versus Code Review.

There are two benefits introduced by this different approach:

- **Active involvement:** This review by other people is a proposed action and is not mandatory as in pair-programming, so the people participating in the review are often keener to cooperate actively.
- **Discussion retrospective:** The different opinions and discussions around the solution are stored and associated to the code: this feedback can be then analyzed and resumed at a later date, even possibly when the review is actually finished, which is particularly useful if there is a risk of team turnover.

Pair-programming however can still be used whenever a higher communication bandwidth and short turnaround of the discussion is needed, for example developing a particular complex algorithm where four focused eyes are definitely needed.

External early feedback

The possibility of having anyone reviewing your code allows external contributions from people with different mindsets and skill levels to share their views on the solution. All feedbacks are valuable, take for example, a Junior developer whose comment may be "I don't understand this fragment" – this is precious input and could prompt you to break the complexity and introduce much simpler statements. Stable code has to be simple to be maintainable: a piece of code that is obscure to a majority of developers would have more chances to be amended in the wrong way and thus to become a critical and unstable part of the project in the future! A different scenario is when we open up the review to external users or developers, including those that are accessing the code from a user perspective (that is API user, as in an Android open source project). External users can provide a different perspective on the solution providing input on its usability and the correctness and completeness of the associated documentation available. One of the fundamental aspects of Agile software development is to get feedback early in the process: with Code Review we can allow feedback when the code has not even been completed and even before having built and released it.

Shared code style

Not all programmers share the same code style, learning to read somebody else's code also means getting used to a different way of writing code and thus potentially being able to maintain it. When developers spend a significant amount of time reading other people's code, they learn not only about other parts of the project but also, consciously or not, to conform to others' coding styles.

Another more obvious way to achieve style unification is by enforcing a set of coding rules that can be defined and checked for compliance at every code build, providing an automatic negative review in case of discrepancies (Example: an open source project may require the code license to be inserted on top of each source). This type of enforcement is typically a good thing but can sometimes be seen as an additional cost or an activity that will slow down the development cycle, even if it definitely has long-term benefits.

An indirect but effective way of achieving an agreed shared code style is to get the team just spending time reading each others code: programming languages follow the rules of natural languages, we speak and write using the same constructs of the language we hear and read every day. When a project starts with Code Review from the outset, it will naturally conform to a unique type "dialect" of language syntax shared by the developers that are working on it and reviewing it on a daily basis. It is common practice, however, to have a contributor's guide that describes the code conventions representing the "dialect" of language that has been agreed over the lifetime of the project.

Team engagement

Code Review has the power to trigger a complete set of new behaviors and dynamics among the developers. Some members naturally emerge and take the lead by commenting and proposing new creative solutions, suggesting easier frameworks or simpler algorithms for achieving the same goal. The review activity provides a natural selection of the more proactive members of the team by showing their value and appreciation through comments and commit approvals. A by-product of this is that less-senior developers are likely to push themselves to improve and try to keep up with the strongest members of the team.

The Code Review process can facilitate the team members engagement in a positive race to get the code reviewed and merged into the mainstream. It is a dynamic similar to the "CI Build game" where each successful or failed build had a corresponding score that was submitted to the last committed author. Turning the code quality selection into a positive race typically provides the benefit of engaging and encouraging people to achieve their best.

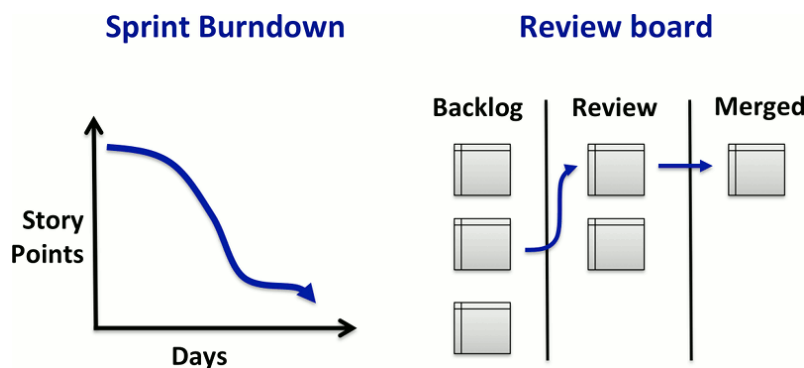
Code Review applied to open source projects becomes an excellent way to attract new team members and allows everyone to make a contribution. Having minimized the risk of breaking the build and blocking the project, allows even a new member of the project can try to express themselves by proposing a patch to the kernel of the project. Each member will also start learning from each other's feedback. The team itself could define an internal "members promotion" scheme within the project, granting new privileges based on the results of past contributions: the core team would then be automatically selected from the most involved developers, naturally elected and recognized by every member.

Qualitative code selection

Getting code under review shuffles the cards and may facilitate the adoption of a more cooperative way of planning software development cycles. Instead of forcing each change to be committed to the target code branch, you allow a natural selection and promotion of the changes that the development team collectively has judged as useful and good enough to be submitted.

This election/selection mechanism is an important new factor to the acceptance criteria of the traditional Agile planning methodologies such as scrum, where user-stories are organized in sprints, and their completion is accounted in the daily stand-up for updating the burndown chart (see the following diagram).

The following diagram shows **Sprint Burndown** chart compared to **Review board**:



When adopting Code Review, the team may decide that a completed user-story is not ready to be committed because the solution implemented does not match the quality criteria defined by its members. The flip side of the medal is that burn-down could become less predictable, as it depends on the outcome of the reviews. The methodology that emerges influences the way scrum gets executed and often drives the teams to **Kanban**, where the improvements to the project come from a collectively agreed set of small changes.

Code Review roles

In order to leverage the benefits of Gerrit, we need to assign some additional roles to the development team members. Understanding the roles, their associated duties and access permissions allows the correct insertion and responsibilities assignment in the Code Review workflow.

Contributor

With Gerrit, every user with the ability to access at least one branch of the repository in read mode and with the ability to upload a new change is known as a **contributor**. The philosophy behind Code Review is "getting early feedback from the team" and often there is no better feedback than just an alternative solution to the same problem: this is the reason why typically every user with read access is granted the ability to upload a change on the code.

Being a contributor does not necessarily provide that person with the ability to actually submit the code to any project branch, nor provide any guarantee that a specific change will be ever approved and merged by someone else.

There are times where changes are submitted as pure sample code (often prefixed with **Request For Comments (RFC)** – in their commit message) for explaining a proposal for a different implementation of an existing feature: every contribution is valuable regardless of whether the code will be effectively be used or not.

The existence of this role makes Code Review fundamentally different from the previous code analysis/inspection tools because anyone can be a contributor and provide input and value, regardless of their project history, access rights or authoritativeness about coding rules and best practices. The ability to use external contributors allows Code Review to be a fuel for innovation and project growth, especially in the open source domain.

Reviewer

In Gerrit every team member is typically allowed to express a score on a given change, expressed as a label + numeric value (positive, negative, or neutral). This role is known as a **reviewer** and can be further segmented according to the ability of the member to provide higher (positive or negative) score values to a change.

Team members and contributors are typically granted the ability to express a positive (+1) or negative (-1) review score in addition to simple comments on the code or on the entire change. Providing a score on a change should not be perceived as judging somebody else's skills: Code Review is always about exchanging opinions on code and should never be used for assessing team members' performance.



An apparently negative review (-1) should be read as an invite for improvement, as the Gerrit meaning for it is "your code looks fine, but I would prefer you to change something before submitting it".

Committer

When team members become more experienced on the code and its underlying design, they start becoming a "technical authority" for the project. They are then entitled to provide more thorough feedback on the code and they will also have the ability to have the final word (either positive or negative) on a proposed change.

These members are known as **committers** and have the ability to provide a positive or negative score up to +2 (-2) for a change. The results of these reviews are absolute: +2 enables the code to be submitted into the main branch, while -2 represents a veto for the change not to be accepted.



Bear in mind that Gerrit reviews are *not* calculative: two +1 reviews does not make a +2. Additionally, positive and negative reviews do not mutually compensate: when a -2 (veto) is given to a change it cannot be eliminated by any positive +1s or even a +2 feedback.

Typically, the number of committers is fairly limited, in the Gerrit Code Review project there are no more than six committers. This is a role that can be earned on the basis on how many changes and reviews have been successfully performed by each member during the history of the project.

Maintainer

It is sometimes useful to assign one committer the duty of administering and monitoring the project, but who then has no active role in the Code Review process. This role is called the **maintainer**. Their main activities are the administration of the user-to-role mapping and the assignment of the access control permissions of different roles to the project branches.

If we take as example the Gerrit Code Review OpenSource project, the maintainer is Shawn Pearce, the project founder. He has a couple of committers who have been delegated the administrative roles whenever needed.

Review labels and roles customization

Today Gerrit is a highly customizable Code Review system and all labels and concepts mentioned so far can be tailored to a project's needs using a powerful rule engine. Every review input is generally referred to as *labeling* with a positive/negative score. The logic behind the score can also be customized and used in project's rules for approving/rejecting the changes.

Gerrit itself uses a set of default rules for implementing the logic described so far, but a project maintainer can create additional rules and define new templates to be used in other projects.

The configuration of customized rules is beyond the scope of this book but it is covered in a set of simple "cookbook recipes" inside the Gerrit online documentation. (See <https://gerrit-review.googlesource.com/Documentation/prolog-cookbook.html>)

Review terms and workflow

Whenever a new tool is adopted, the initial problems arise when new terminologies are encountered, especially when they are referred to new concepts and methodologies. It is crucial to educate about keywords and their associated meaning in the Gerrit Code Review domain.

Project

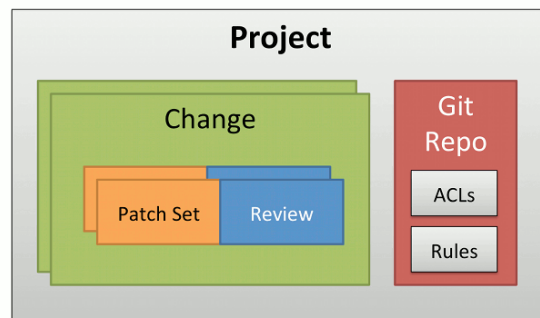
Gerrit project is a workspace consisting of the following elements:

- **Git repository:** It is used to store the merged code base and the changes under review that have not being merged yet. Gerrit has the limitation of a single repository per project. There can also be projects without any code repository associated at all (that is, Security-only projects)
- **Changes references under review:** Git commit-id (expressed as SHA-1 Hexadecimal alphanumeric string) stored in the Gerrit DB and pointing to the corresponding changes stored in the Git repository
- **Access Control Lists (ACLs):** It contains the list of roles defined for the Gerrit project and the associated access permissions to the Git repository branches.
- **Prolog rules:** It is the set of rules that govern the Code Review process for the project.
- **Additional metadata:** All the extra settings such as description, merge strategy, contributor agreements, and accessory metadata needed in order to manage the project.



Gerrit supports the ability to define fine-grained access of different roles for specific branches. This feature enables there to be specifically defined levels of review depending on the criticality of the target branch (that is, experimental versus production branch).

The following diagram shows Gerrit Code Review project elements:



Change

A Gerrit change is a Git commit object uploaded for review and associated to its comments and scores. It is stored in the project's Git repository but it is not visible/accessible from the normal Git graph of commits, even it does start from a point on the commits graph.

In order for Gerrit to access the Git commit objects associated to its changes, it keeps the list of changes commit-ids in its own relational DB which are organized on a per project basis. Each change has its own unique ID, which is different to the underlying Git commit ID. The reason for this is due to the capability for a change to be amended with a set of different commits: the change id keeps a consistent link across all different amended commits.

The change id is stored at the bottom of the Git commit message into a field named **Change-Id**.

Gerrit changes include one or more patch-sets: all of the amended Git commits uploaded for review on the same change are stored as patch-sets where the last one represents the most recent code candidate to be merged.

Patch-sets are numbered, starting from 1 and incremented whenever a change is amended with another Git commit. It makes sense only to review only the latest patch-set as the previous ones are kept only for historical reasons, for allowing Reviewers to understand what's new from the latest inspection.



A common mistake is to forget the difference between a patch-set and a change in Gerrit and, based on a negative review on an existing change, submitting a different change instead of uploading just another patch-set within the same change that received the feedback.

Label Code Review

Gerrit uses one default Code Review label with a numeric value to allow the scoring of a patch-set during a Code Review phase.

Here are the possible values and the associated meanings:

Label	Value	Meaning
Code Review	-2	Veto, code cannot be submitted in any case
	-1	Code looks good but changes are needed before submission
	+1	Code looks good but I need more people to review it
	+2	Code looks good and can be submitted

Another common and very useful label used previously in the Android open source project is the **Verified** label. Since Gerrit Version 2.6 it is no longer pre-defined but it is still configurable as an additional review label with the values provided in the following table:

Label	Value	Meaning
Verified	-1	Code does not work
	+1	Code works fine



The combinations of Code Review/0 and Verified/0 are typically used for reviews consisting of only comments without any score.

Submit change

Submit is the action of accepting a change and requesting its incorporation into the target branch. The default set of rules for enabling a change for submission requires at a minimum a Code Review/+2.



Each project may potentially have a set of different rules associated with it and expressed in terms of the project's rules.



Submitting a change does not necessarily involve the actual inclusion of it into its target branch: as Code Review happens on a forked branch it is needed to be merged or rebased to the target branch in order to be closed and incorporated.

Merge change

This is the action of merging a submitted change into its target branch. It is separated from the actual submission because of the possibilities of choosing different strategies for incorporating the change.

Bear in mind that non-fast-forward merges will generate a version of the code that could potentially be different from the one under review, depending on the merge strategy configured in Gerrit. In the case of merge conflicts, the operation will be blocked and the change will remain submitted but un-merged: this will then require the change's author or one of its Reviewers to be notified of the failure, to rebase the change and submit a new patch-set for review.

Abandon change

Whenever a change after a series of negative or inconclusive reviews does not reach the minimum requirements for being submitted, it can be flagged as abandoned so that all the team members can avoid performing any further code review on it. This typically happens because either the author gives up in trying to amend the code or just because the change has become obsolete compared to the evolution of its target branch.

Don't forget that abandoned changes are not lost or removed from the repository, so they can potentially be restored at a later stage.

Summary

We have introduced Gerrit Code Review and the main benefits that it brings to the development team. In addition to the pure code quality improvement and the reduction of broken builds, Code Review has a positive effect on the team's dynamic, encourages knowledge sharing, and allows an easier on-boarding of new members and their subsequent promotions.

The main roles of Gerrit Code Review are in addition to the developer/committer contributor, and reviewer often external members of the team without push permissions to the target branch. Allowing external members to provide feedback on the code improves the API contract feedback and promotes the positive growth of the product's functionality.

We have also introduced the glossary of Code Review terms that will be used throughout this book, with an initial overview of their meaning and implication in the Code Review process: now we are ready to move on to the detail of the workflow.

2

Setting Up and Quick-start

This chapter explains in detail the Gerrit setup and configuration of an initial sandbox environment. We will take our first step with Gerrit by creating an initial administrator account, one repository, and we will also clone and set-up our local Git repository, ready for pushing our first change for review.

We will cover the Gerrit WAR distribution, going through the basic steps of getting the binaries and setting up a default working configuration, primarily used for training and test environments.

By the end of the chapter you should have a fully working Gerrit environment setup.

Pre-requisites

Gerrit Code Review assumes a basic knowledge of the Java Runtime commands and familiarity of Git.

System pre-requisites are as follows:

- Java JDK Version 1.6 (or later)
- Git native executable Version 1.6 (or later)
- Posix utilities (for example, `bash`, `grep`, `cut`, `echo`, and `perl`)

In theory Gerrit could run on any operating system that supports Java JDK Version 1.6, but in reality there are some challenges with the non-posix compliant ones, such as Microsoft Windows. Internet connectivity is typically needed for downloading the additional strong cryptography libraries for using highly secure protocols algorithms for Git over SSH and HTTP/S.

Gerrit download

The Gerrit home page (<http://code.google.com/p/gerrit>) contains the direct link to download the WAR package file and contains the full list of downloadable packages. At the time of writing of this book, the latest upcoming version of Gerrit was Version 2.7-rc4, can be downloaded from <https://code.google.com/p/gerrit/wiki/Downloads>.

We can use `curl`, a popular command-line utility on Posix systems, to download the Gerrit package in our sandbox environment, create the Gerrit home directory (`/opt/gerrit` in this case), and download the Gerrit WAR package into it:

```
$ mkdir -p /opt/gerrit
$ cd /opt/gerrit
$ curl http://gerrit-releases.storage.googleapis.com/gerrit-2.7-rc4.war
-o gerrit.war
% Total % Received % Xferd Average Speed Time Time Time
Current
                                Dload Upload Total Spent Left
Speed
100 30.5M 100 30.5M 0 0 1527k 0 0:00:20 0:00:20 --:--:--
1816k
```

Gerrit is packaged as a WAR (Java Web application ARchive) and can be used either in a Servlet 2.x compliant container (that is Apache Tomcat Version 6.x or later) or in its own standalone Jetty container, which is the most popular setup.

Running Gerrit initial setup

Before starting the Gerrit setup we need to create a dedicated user for it (for example, Gerrit), in order to allow the service to work in its own account sandbox, and not under the system administrator (root) account. For better convenience we assign the `/opt/gerrit` as home directory:

```
$ useradd -d /opt/gerrit gerrit
$ chown -R gerrit:gerrit /opt/gerrit
$ su - gerrit
```

Gerrit WAR is packaged in such a way that it can be used as standard executable archive for Java and can be run with the following command line:

```
$ java -jar gerrit.war
```

We will set up a reduced sandbox Gerrit installation with the following characteristics:

- Default security algorithms (we will cover stronger security later on)
- Standard HTTP without SSL
- Local users registration without password validation (for sandboxes only)

These characteristics are beneficial when you want to get up and running quickly with Gerrit and limits having to spend too much time setting up complex security and user authentication settings. Due to their nature, these can potentially take a lot of time and deserve a dedicated chapter to cover them properly.

The Gerrit init wizard goes through all the common settings, requesting the value for each one of them. The text shown between squared brackets `[]` indicates the suggested default or the value already present in your current configuration. In order to accept the value displayed, simply press `RETURN`. To provide a different value, just enter the new values and press `RETURN`.

Whenever a multiple choices are available, by entering a question mark `?` and pressing `RETURN` it is possible to display the full list of options.

We will now move into the Gerrit initial directory created previously and run the `init` command as follows:

```
$ cd /opt/gerrit
$ java -jar gerrit.war init
```

```
*** Gerrit Code Review 2.6-rc3
```

```
***
```

```
Git repositories directory
```

```
*** Git Repositories
```

```
***
```

```
Location of Git repositories [git]: RETURN
```

Press `RETURN` for having the Git repositories stored under a subdirectory called `git` under the Gerrit initial directory.



It is typically recommended to specify a different absolute directory for storing the Git repositories. It is typically a good idea to have them into a separate volume in order to avoid running out of disk space on the Gerrit application volume because of growing Git repository objects and for allowing easier upgrades between different versions

- **Gerrit DB:** Here we will request the list of supported database options by inserting ? and pressing RETURN. For a production installation the recommended option is a relational database such as MySQL or PostgreSQL. You could also use any JDBC compliant database (including Oracle and Microsoft SQL Server) but it is worth bearing in mind that you may encounter some difficulties, as this combination has not been thoroughly tested. For the purpose of our sandbox environment, we will use H2, a Java embedded SQL database that does not require any extra setup step and stores all the data into the local file system.

```
*** SQL Database
***
```

```
Database server type           [h2]: RETURN
```

- **User registry:** Gerrit is designed to delegate the user authentication to a dedicated external user registry (openid, openid_sso, ldap, and ldap_bind), a front-end HTTP authentication proxy (http and http_ldap), or even a combination of the two using X.509 client certificate validation and lookup (client_ssl_cert_ldap). In a sandbox environment we want to skip user registry validation altogether: type development_become_any_account and press RETURN.

```
Authentication method          [OPENID/?]: development_become_any_
account RETURN
```

- **Email submission:** The next configuration step is for allowing Gerrit to send e-mail notifications, and you can easily skip this step by pressing RETURN for each question.

- **Gerrit Java Container:**

```
*** Container Process
***
```

```
Run as                         [gerrit]: RETURN
Java runtime                   [/usr/java/jdk1.6.0_32.jdk/jre]:
RETURN
Copy gerrit.war to /opt/gerrit/bin/gerrit.war [Y/n]? Y RETURN
Copying gerrit.war to /opt/gerrit/bin/gerrit.war
```

- **Git-over-SSH Server:** The next step is the Git SSH Daemon configuration: we accept the default values except for the Bouncy Castle Crypto library where we answer n. At this stage we do not want to request Internet connectivity to download the library. It can be easily downloaded and installed once the sandbox setup is reconfigured as a production instance. The Gerrit `init` process will automatically generate a new set of DSA/RSA SSH Server keys to be used for the Git SSH Daemon.

```
*** SSH Daemon
***
```

```
Listen on address          [*]: RETURN
Listen on port             [29418]: RETURN
```

```
Gerrit Code Review is not shipped with Bouncy Castle Crypto v144
  If available, Gerrit can take advantage of features
  in the library, but will also function without it.
Download and install it now [Y/n]? N RETURN
Generating SSH host key ... rsa(simple)... done
```

- **Gerrit/Git HTTP Server:** Gerrit includes a HTTP Server for both the web-UI and the underlying Git protocol. Make sure that the default port proposed is available on your system and traffic is allowed from the external network through the firewall. You also need to make sure that the "Canonical URL" auto-detected by Gerrit is the external hostname used to reach the machine from the outside.

```
*** HTTP Daemon
***
```

```
Behind reverse proxy      [y/N]? RETURN
Use SSL (https://)        [y/N]? RETURN
Listen on address         [*]: RETURN
Listen on port             [8080]: RETURN
Canonical URL              [http://myhost.mydomain.com:8080/]:
RETURN
```

- **Plugins and Gerrit site init:** The last step is the request for additional plugins to be installed: we do not need any of them to get started using Gerrit so we answer n and after pressing `RETURN` the initial setup is finalized by creating all the initial start-up data and the DBMS of our new installation.

Installation completed

Let's review the main directories created by the Gerrit init process:

- `/bin` with Gerrit WAR file and `gerrit.sh` startup script
- `/db` with Gerrit DB files
- `/etc` with Gerrit configuration files (`gerrit.config` and `secure.config`)
- `/git` with the initial `All-Projects.git` repository
- `/logs` that will contain Gerrit log files

Our setup is now complete and we are ready to start using Gerrit!

Log in and create user profile

Gerrit daemon is controlled by `gerrit.sh` script; in order to start up Gerrit just run `gerrit.sh` with the `start` option as follows:

```
$ /opt/gerrit/bin/gerrit.sh start
```

After a few seconds (on slower systems this can be up to a few minutes) the Gerrit startup is completed successfully.

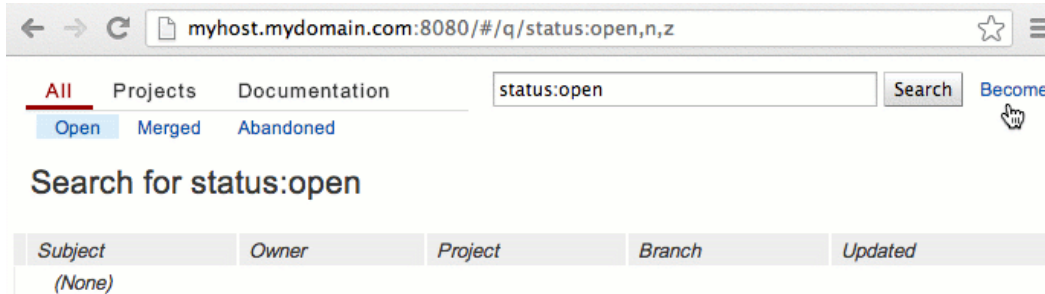
```
Starting Gerrit Code Review: OK
```

If you encounter any problems, the Gerrit error log file can be reviewed for log lines tagged with an `ERR` label. In our example everything has run smoothly and we can see only `INFO` and `WARN` messages.

```
[...]
[2013-05-26 23:36:14,946] INFO   org.eclipse.jetty.server.handler.
ContextHandler : started o.e.j.s.ServletContextHandler{/,file:/opt/
gerrit/tmp/gerrit_7224538249578427510_app/gerrit_war/}
[2013-05-26 23:36:15,390] INFO   org.eclipse.jetty.server.
AbstractConnector : Started SelectChannelConnector@0.0.0.0:8080
[2013-05-26 23:36:15,392] INFO   com.google.gerrit.pgm.Daemon : Gerrit
Code Review 2.7-rc4 ready
```

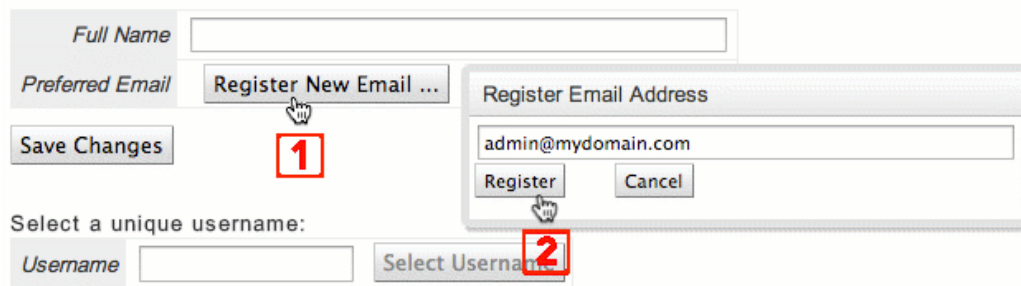
Next we can access the Gerrit web-UI by opening the browser and entering the Canonical URL previously configured during the Gerrit init.

The following screenshot shows the Gerrit home screen:



Click on the **Become** link and then click on the **New Account** button to register your first Gerrit account.

The following screenshot shows Gerrit account registration:



The Gerrit account registration screen allows you to specify the key identification information for new users. The most important identifier is their e-mail, as it is used for two main use-cases:

- Identification of the author of a Git commit
- Assignment and notification of Code Reviews

The first user that you create on the system is automatically considered to be the Gerrit system administrator. We register the user with the e-mail address `admin@mydomain.com` and we can subsequently assign a login username and associated first/last name.

The following screenshot shows **Full Name** and **Username** selection:

Full Name: Gerrit Admin

Preferred Email: admin@mydomain.com Register New Email ...

Save Changes

Select a unique username:

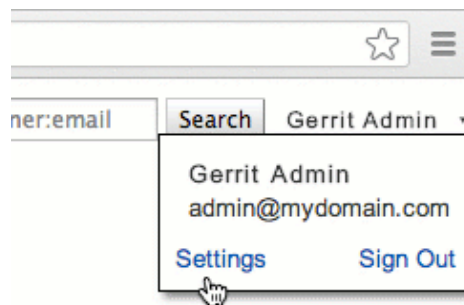
Username: admin Select Username

Generate HTTP password for Git

The first user account has now been created but in order to use it with the Git protocol we need to add a minimum level of security on HTTP: generating a random password to be used from the client. Even when used as a sandbox environment, empty passwords are not sufficient to allow a Git client to perform write operations to the repository.

When you go into the top-right drop-down menu, displaying the name of the user logged in, select the settings link.

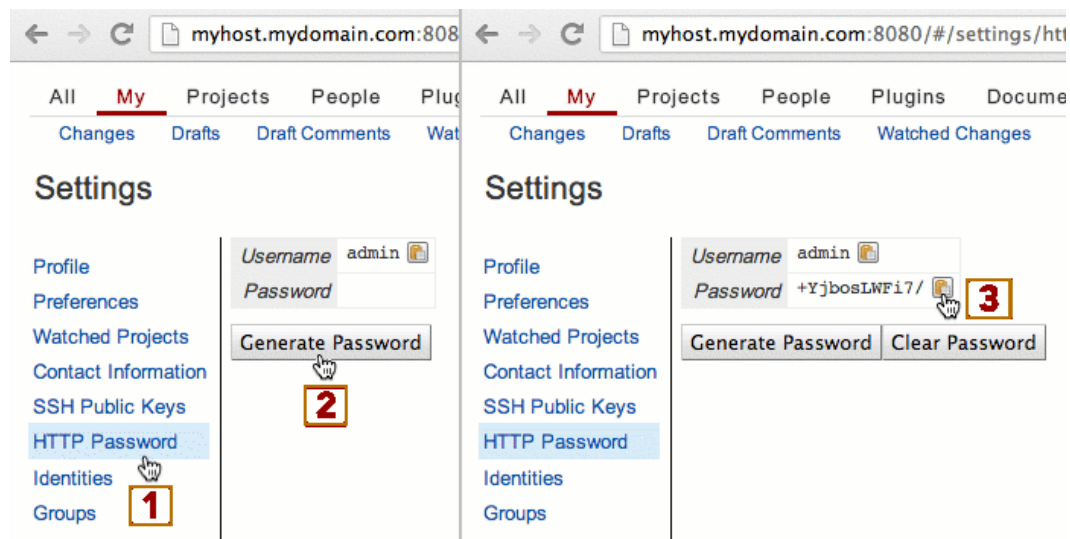
The following screenshot shows the Gerrit **Settings** link:



The user settings screen allows you to customize all information, preferences, and security settings for the current user. In this scenario we only want to specify an HTTP password for the Git protocol, so you need to select **HTTP Password** and then click on the **Generate Password** button. A new random password is generated and stored in the Gerrit DB associated to the user.

To simplify the copy-and-paste of the password, Gerrit web-UI includes a simple icon that automatically copies the password to the system clipboard so that it is ready to be pasted into any other client application. This icon will not be shown if the browser does not have Flash Player installed.

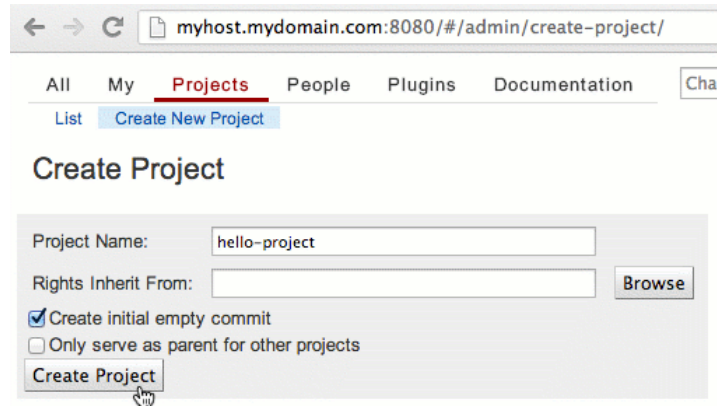
The following screenshot shows the Gerrit settings page for generating an **HTTP Password**:



Create and clone your first project

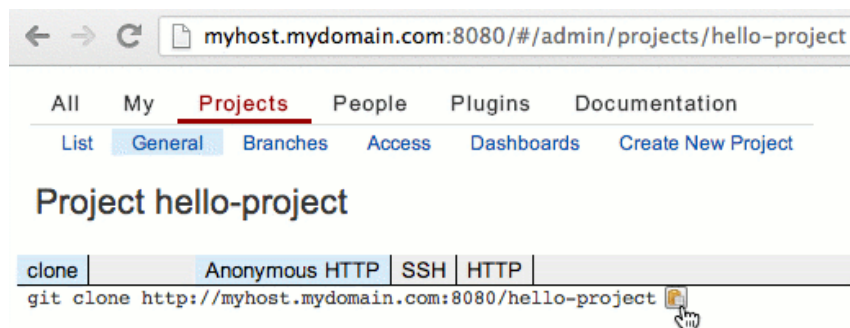
The Gerrit admin user is automatically granted the permission to create new Gerrit projects. To do this the admin user will need to click on **Projects** and then **Create New Project** from the top menu bar: the create project screen will then be displayed.

The following screenshot shows the Gerrit **Create Project** screen:



The project name should coincide with the Git repository name and for now we will leave the **Right Inherit From** field empty. We then select **Create initial empty commit** and click on the **Create Project** button.

The following screenshot shows the Gerrit project clone command:



The Gerrit project has now been created and from the settings page we can click on the icon on the top-right to project clone command: this will copy the `git` command line for cloning the project from a remote client. We can then change the URL type (anonymous HTTP, SSH or authenticated HTTP) by clicking on one of the table columns at the top of the project and then copy & paste the `git clone` command as follows:

```
$ git clone http://admin@myhost.mydomain.com:8080/hello-project
Cloning into 'hello-project'...
remote: Counting objects: 2, done
remote: Finding sources: 100% (2/2)
remote: Total 2 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (2/2), done.
```



Although we specified an HTTP password for the admin user, in this case the hello-project has not requested any username/password authentication for the clone operation, as the default ACLs for the project allowed anonymous reads.

In order to get the local repository ready to be used for Gerrit, we need to perform the following two additional important steps:

1. Configure the Git `user.email` and `user.name` aligned to the full name and e-mail previously entered in the Gerrit user registration screen.
2. Install the Gerrit `commit-msg` hook into the locally cloned Git repository.

The following example shows how to configure the Git identity on the cloned repository:

```
$ cd hello-project
$ git config user.name "Gerrit Admin"
$ git config user.email "admin@mydomain.com"
```

Additionally you will need to install the Gerrit `commit-msg` hook, used to generate the **Change-Id** field, a globally unique ID for each new change created with a Git commit.

```
$ curl -Lo .git/hooks/commit-msg http://myhost.mydomain.com:8080/tools/
hooks/commit-msg
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
  Current                                 Dload  Upload   Total   Spent    Left
Speed
100 4278  100 4278    0     0  214k      0 --:--:-- --:--:-- --:--:--
- 596k
$ chmod +x .git/hooks/commit-msg
```

The cloned repository is now ready for creating local Git commits and pushing them to Gerrit.

Summary

In this chapter, we have learned how to download Gerrit WAR and perform a default installation for a sandbox environment. Gerrit requires Java, Git, and a Posix environment as pre-requisites or can simply be installed by using a native setup.

By using the `development_become_any_account` authentication settings we have been able to create the first user, considered by Gerrit as the system administrator. We have also seen how to create a password in order to use Git over HTTP and we took our first steps with Gerrit by creating a new hello-project.

Gerrit provides easy shortcuts in its web-UI to get the clone commands for the projects: we cloned the hello-project and we created our first Git commit for review.

Change-Id is the unique identifier of a code contribution in the Code Review system: for this purpose we have downloaded and installed from Gerrit a useful `commit-msg` hook in order to generate a **Change-Id** for our changes.

3

User Authentication

This chapter explains how Gerrit integrates with existing user registries for performing authentication and completing the user profile details. We will understand how the authentication process works and what the possibilities are for configuring it to connect to LDAP, OpenID, or any other third-party system through Single-Sign-On. By the end of this chapter, our initial sandbox installation will be fully operational so that it can be used as a production system with real user validation.

How Gerrit user authentication works

Gerrit Code Review was designed to uniquely provide the user with Git Repositories and Code Reviews, and not to manage a fully featured registry of profiles and credentials.

This is why we will not find any explicit user management screen or mechanism in Gerrit: we will see instead a set of powerful configuration mechanisms for integrating with a number of different external authentication systems (for example, Active Directory or other LDAP Server), which are assumed to be already in place.

The registration screen that we used during our sandbox installation did not need any external authentication setting because it considered any unknown user as trusted by default, which is not an assumption we can make for setting up a proper production environment.

Git versus Gerrit UI authentication

Gerrit has a Code Review plus a Java-based Git Server engine and they both share the same users' domain for three specific purposes:

- **Git authentication:** It is used to authenticate users accessing through the Git protocol.
- **Gerrit interactive authentication:** This is designed to allow access to the Gerrit web-UI in order to perform either a Code Review or for Gerrit administration.
- **Gerrit batch authentication:** It is used only for performing batch operations through the Gerrit command-line interface.

Under all those scenarios, Gerrit needs to have the user registered in its own internal DB, similarly to the way it was managed in the sandbox environment setup. When it comes to user credential validation (password or other secure credentials), Gerrit relies on the queries made to the authentication system configured during the init phase. The only exception is where SSH public key authentication is used and the actual public key credentials are stored in Gerrit and cannot be retrieved externally.

The Gerrit interactive web authentication flows as follows:

1. Gerrit checks the credentials on the authentication system (for example, LDAP or OpenID).
2. If the authentication is successful, Gerrit checks that the user already exists in the DB.
3. If the User does not exist, Gerrit shows the registration page (except with LDAP).
4. Alternatively Gerrit shows the user's dashboard.



With LDAP authentication, the registration page is not displayed, as the user's profile is automatically populated from the LDAP user registry.

Gerrit batch SSH authentication flows as follows:

1. Gerrit checks if the user already exists in the DB.
2. If the user does not exist, authentication fails.
3. Alternatively SSH authentication is successful using the public key in the DB.

Git authentication:

1. Gerrit checks if the user already exists in the DB.
2. If the User does not exist, authentication fails.
3. Alternatively Gerrit checks the user's credentials (Public Key / HTTP password) in the DB.

In a nutshell, Gerrit always uses the profile stored in the local database, but in order to create a new profile it requires the user to have been previously logged in using the Gerrit UI, and therefore they have already been authenticated against the external system (for example, LDAP).

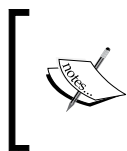
Gerrit internal accounts

In order to support unattended operations, for example build jobs, Gerrit introduces one important exception to the above rules: Gerrit internal system accounts. These are not real users but internal "robot" users intended to be used by non-interactive batch processes.

Internal accounts are typically used for the following reasons:

- Cloning Git repositories for build purposes
- Listening to Gerrit events

Internal accounts do not exist in the external authentication system, as they should never be used for repository user commits, but they technically could if allowed by access permissions. According to the authentication flows previously described, they will not succeed in getting through the Gerrit interactive web authentication but will be allowed to perform a Gerrit batch or Git SSH/HTTP operation.



Internal accounts are not verified against the official user registry and do not allow traceability of "who did what" in a reliable way: allowing those accounts to actually perform any operation on Gerrit would break the Company policies on Security and Auditing.

Gerrit internal accounts can be created by the Gerrit Administrator or by a Gerrit group of users delegated with creating internal accounts.

See the example below for how to create a "robot" batch Gerrit user with a pre-assigned SSH key and HTTP password, assuming the existence of a Gerrit administrator user admin:

```
$ cat /home/robot/.ssh/id_rsa.pub | ssh -p 29418 \  
  admin@myhost.mydomain.com gerrit \  
  create-account --fullname robot \  
  --http-password batch728p@sswd --ssh-key - robot
```

Authenticating over the Internet through OpenID

When using Gerrit over the Internet, there are a number of external authentication systems available that can be used out of the box without hardly any additional installation or management effort. New users who start using Gerrit are very likely to already have been registered to GitHub, Google+, or even just have a Facebook profile.

Even for companies that are using the Internet as a public network for their private business, the availability of "Google AppEngine" and other common service offerings allows them to use the same open authentication technologies and protocols for their Corporate Cloud Services, including e-mail and Collaboration platforms.

At the time of writing this book, Gerrit supports only the Internet authentication systems based on the OpenID 2.0 authentication protocol out of the box. (http://openid.net/specs/openid-authentication-2_0.html)

In order to enable OpenID authentication, we will need to stop Gerrit and go through the init steps again, as we did during the setup. All of the existing settings will be read from the Gerrit configuration file (`gerrit.config`) and provided as default values. In order to change them it is necessary to enter a new value for each when they are presented during the init, and to leave them unchanged just click on RETURN.

```
$ /opt/gerrit/bin/gerrit.sh stop
```

```
Stopping Gerrit Code Review: OK
```

```
$ cd /opt/gerrit
```

```
$ java -jar gerrit.war init
```

```
*** Gerrit Code Review 2.6-rc3
```

```
***
```

```
[...]
*** User Authentication
***

Authentication method [DEVELOPMENT_BECOME_ANY_ACCOUNT/?]: openid RETURN
[...]

Initialized /opt/gerrit

$ /opt/gerrit/bin/gerrit.sh start

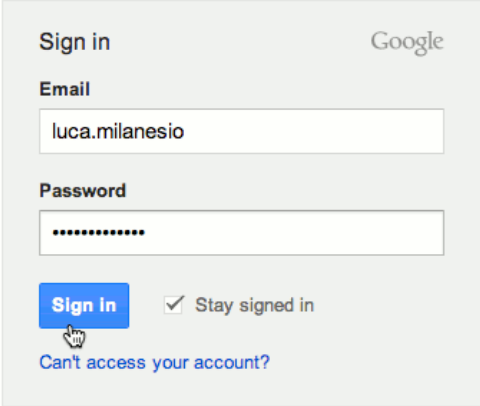
Starting Gerrit Code Review: OK
```

The Gerrit home screen has now changed and two new links will be present in the top-right corner: Register and Sign In. The difference between the two is the Gerrit target landing page displayed once authenticated. Register takes you to the user registration page, while Sign In takes you straight to the Gerrit home, providing the user is already registered. By clicking on either link we go through the OpenID authentication steps, which are as follows:

1. OpenID provider selection:
 - First we need to select the external authentication system for validating the user's credentials. The following screenshot shows OpenID provider selection:



- Gerrit provides pre-defined links to Google and Yahoo, OpenID providers: we can click on the **Google Account** link to select it as the authentication system.
 - Alternatively we can insert any other OpenID provider URL to be used as the user authentication system. (For a list of other providers, see the OpenID foundation providers page at <http://openid.net/get-an-openid/>.)
2. Redirection to provider's authentication:
- Gerrit requires the OpenID provider to return a certified user's identity and it is the provider's responsibility to confirm the user's identity by requesting and validating their credentials. Gerrit does not know the underlying authentication method used; it could be username/password, a cookie or anything else required by the OpenID provider.
 - When choosing Google OpenID, authentication happens directly between the user's browser and the Google account validation as shown in the following screenshot:

A screenshot of the Google Sign in interface. At the top left is the text "Sign in" and at the top right is the "Google" logo. Below the logo is the "Email" label followed by a text input field containing "luca.milanesio". Below that is the "Password" label followed by a password input field with masked characters. At the bottom left is a blue "Sign In" button with a mouse cursor icon pointing at it. To the right of the button is a checkbox labeled "Stay signed in". Below the button is a link that says "Can't access your account?".

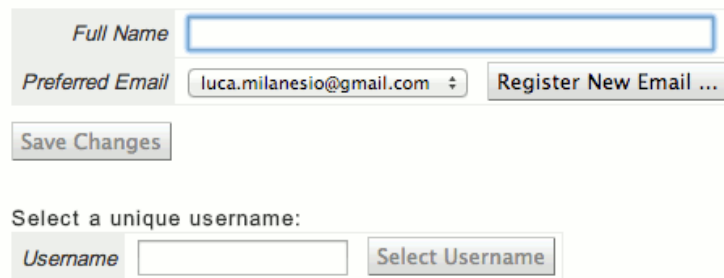
- In some situations Google may already know the user's identity because of a Browser persistent cookie. If this is the case, the login screen won't appear and the Browser will automatically be redirected to the following step.

3. OpenID user identity sent back to Gerrit for registration:
 - We have now completed the Google account login and the certified identity is passed back to Gerrit by redirecting the browser page back to the Gerrit registration page. The following screenshot shows Gerrit registration using Google user identity:

Welcome to Gerrit Code Review

Please review your contact information:

The following contact information was automatically obtained when you signed-in to the site. This information is used to display who you are to others, and to send updates to code reviews you have either started or subscribed to.



The screenshot shows the Gerrit registration interface. At the top, there's a heading "Welcome to Gerrit Code Review". Below it, a message asks the user to review their contact information. A paragraph explains that the information was automatically obtained from Google. The form contains three main sections: 1. "Full Name" with an empty text input field. 2. "Preferred Email" with a dropdown menu showing "luca.milanesio@gmail.com" and a "Register New Email ..." button. 3. A "Save Changes" button. Below these, there's a section for "Select a unique username:" with a "Username" label, an empty text input field, and a "Select Username" button.

Please note that the user's primary e-mail field has been pre-populated using the certified identity information returned by Google accounts. The remaining information will have to be manually entered as done during the sandbox user registration. The information needed is full name and a unique Gerrit username.

OpenID SSO

There is a special scenario where we do not want to show the OpenID selection/configuration to the user: everything should happen behind the scenes and the provider will be the one chosen by the company as their main trusted OpenID authentication system.

In order to modify the authentication settings first we will need to stop Gerrit as follows:

```
$ /opt/gerrit/bin/gerrit.sh stop
```

Stopping Gerrit Code Review: OK

As the configuration of the OpenID Provider URL (Google accounts URL in our example) is not offered during the init steps, we need to edit the `gerrit.config` stored under `/opt/gerrit/etc`.

```
$ vi /opt/gerrit/etc/gerrit.config
```

```
[...]
```

```
[auth]
```

```
    type = OPENID_SSO
```

```
    openIdSsoUrl = https://www.google.com/accounts/o8/id
```

```
[...]
```

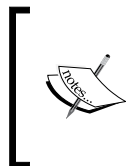
All of the configuration is now complete and we can restart Gerrit:

```
$ /opt/gerrit/bin/gerrit.sh start
```

Starting Gerrit Code Review: OK

The authentication will work exactly as before, but without the selection of the OpenID provider so the Browser will automatically redirect to Google accounts in the background.

If the user already has a Google account cookie stored in their browser, simply clicking on the **Sign In** link will automatically trigger a transparent login to Gerrit without requesting any additional information. This is why this type of authentication is referred to as a **SSO (Single-Sign-On)**.



When using OpenID authentication, the user ID returned by the provider (for example, Google) is typically related to the full canonical URL defined in Gerrit; changing this will have the effect of invalidating all existing accounts registration and all new logins will be then treated as new users.

Using in-house private Gerrit authentication (LDAP)

When using Gerrit inside a private corporate network, the most common authentication registry is Active Directory, or more generically an LDAP server.

The authentication process is radically different from the one previously described for OpenID. Following are the differences:

- There is no direct Browser to LDAP authentication. Gerrit mediates the communication between the two.
- Gerrit can pre-populate much more information on the user's profile thanks to a richer set of user attributes stored in the LDAP registry.
- Group ownership can be pre-populated using LDAP.

Gerrit LDAP support is divided into two groups as follows:

- Authentication and Lookup
- Lookup only

The first group is the most common setup and is the one discussed in this chapter.

The second group is more of a companion to a different type of authentication (HTTP or X.509 Client SSL Certificate) and the LDAP is used only to populate the user's identity that has already been validated by a different means. This is classified as third-party authentication because the LDAP plays a minor part in the authentication process.

LDAP configuration steps

For the purpose of showcasing a generic LDAP configuration, we will use an ApacheDS default installation running on the `ldap.mydomain.com` server, port 10389, with the default domain context – DC: ou=system - LDAP Admin: uid=admin, ou=system (password=secret) - Users: uid=*, ou=users, ou=system - Groups: cn=*, ou=groups, ou=system.

We need to stop Gerrit again and restart the init steps in order to review the authentication configuration settings and then set up the authentication against the ApacheDS LDAP server:

```
$ /opt/gerrit/bin/gerrit.sh stop
```

Stopping Gerrit Code Review: OK

```
$ cd /opt/gerrit
```

```
$ java -jar gerrit.war init
```

[...]

*** User Authentication

Authentication method [OPENID_SSO/?]: **ldap RETURN**

LDAP server : **ldap://ldap.mydomain.com:10389 RETURN**

LDAP username : **uid=admin,ou=system RETURN**

uid=admin,ou=system's password : **secret RETURN**

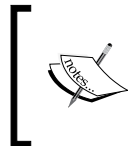
confirm password : **secret RETURN**

Account BasedDN : **ou=users,ou=system RETURN**

Group BasedDN [ou=users,ou=system] : **ou=groups,ou=system RETURN**

[...]

Initialized /opt/gerrit



All passwords are stored in separate Gerrit configuration file called `secure.config`; this allows the system administrator to differentiate and potentially enforce this separate file with additional access restrictions.

Gerrit can now be restarted.

```
$ /opt/gerrit/bin/gerrit.sh start
```

Starting Gerrit Code Review: OK

We will now run through all of the steps required for Gerrit LDAP authentication through the web-UI.

1. Gerrit LDAP username / password Sign In:
 - By clicking on the `Sign In` button, we will get a different login experience than the one seen before. The username/password login screen shows that the credential validation with LDAP is mediated by Gerrit. The following screenshot shows the Gerrit LDAP Login form:

Sign In to Gerrit Code Review at myhost.mydomain.com



2. Gerrit LDAP search:
 - Gerrit searches for the username through the entire LDAP tree, starting from the "Account BaseDN" and connects to the LDAP using the username/passwords specified during the init phase.
 - The username used for the LDAP bind when performing this search is *not* the actual user that is trying to log in but is a super user, configured during the Gerrit init, that has complete visibility of the entire tree in order to perform a complete deep lookup. If the user is not found or the LDAP bind cannot be completed, authentication fails.



This step gets executed in the background without visual responses shown in the browser.

3. LDAP bind with user's full DN:
 - If the user is found, its full DN (LDAP Distinguished Name) is retrieved and used to perform a second bind to the LDAP Server, this time using with it the password entered by the user at the Gerrit login screen.



This step gets executed in the background and only in the case of a bind failure, the user is notified in the browser.

4. LDAP user profile:

- Gerrit takes all of the information it needs for pre-populating the entire registration profile. This means that in order to start using its profile with Gerrit no further input is required by the user after their first login.

The Gerrit default LDAP fields for user profile mapping are:

Gerrit User attribute	LDAP attribute
Full name	displayName
Primary e-mail	Mail
Username	uid

Once the authentication is complete and the profile fully populated, the user is directly logged in to Gerrit.

Read-only LDAP user profile

When Gerrit fetches the user profile from the LDAP, it is automatically set to read-only and cannot be modified using the web-UI.

The value of LDAP authentication over OpenID is the standardization and alignment of the Gerrit usernames with the ones used by the Company on all of their existing systems. This generates fewer problems when troubleshooting common issues and allows more control centrally for enabling/disabling users.

Active Directory

Authentication using Microsoft Active Directory is a special type of LDAP, as there is a "magic combination" of Gerrit configuration settings that allows the management of Microsoft's implementation variants.

Following are the typical settings for `gerrit.config`:

```
[ldap]
    server = ldap://dc.ad.company.com:3268
    username = ldapuser@ad.company.com
    accountBase = DC=ad,DC=company,DC=com
    groupBase = DC=ad,DC=company,DC=com
    accountPattern = (&(objectClass=person)
(sAMAccountName=${username}))
    accountFullName = displayName
    accountEmailAddress = mail
    accountSshUserName = sAMAccountName
    groupMemberPattern = (sAMAccountName=${username})
    groupName = cn
    localUsernameToLowerCase = true
```

In this example, Gerrit connects to the federated Active Directory Global Catalog (port 3268) of a company domain named "dc.ad.company.com".



There are a number of other ways to make Active Directory work with Gerrit by using very complex mappings and translations, however the example shown represents the most widely used combination tested by corporate Gerrit users. Other implementations of Active Directory may use a different mapping of the user's profile fields to their Gerrit counterparts.

Third-party authentication options

At the time of writing this book, the support for third-party authentication systems outside the ones already mentioned is still quite limited. The next forthcoming Version 2.8 of Gerrit will introduce the concept of "pluggable authentication backend" that would provide a real integration with any external authentication system.

With this in mind, the only option is to put another HTTP frontend reverse proxy in front of Gerrit and manage the authentication externally. Gerrit will then get the user identity from a trusted HTTP header generated securely by the HTTP frontend.

Example – Apache HTTP frontend

By using Apache 2.4 as a HTTP frontend, we can use the following available authentication modules:

- SQL database (`mod_authn_dbd.c`)
- DBM files (`mod_authn_dbm.c`)
- Password file (`mod_authn_file.c`)
- Radius (`mod_auth_radius.c`)
- Kerberos (`mod_auth_kerb.so`)

In addition to these free authentication modules, there are a number of existing commercial modules that also integrate with Apache.

Setting up Gerrit behind a reverse proxy

Gerrit configuration needs to be changed whenever it receives HTTP calls through in-bound proxies. This can easily be done by adding the `proxy-` prefix to the existing Gerrit configuration settings for the HTTP listener.

It is also a good security practice to limit Gerrit so that it only listens to its IP loopback (`127.0.0.1`) so as to prevent any external system accessing it without first going through the frontend.

An example of the `gerrit.config` snippet:

```
[httpd] listenUrl = proxy-http://127.0.0.1:8080/
```

Where there is an HTTP frontend with an Apache 2.x reverse proxy, the following `httpd.conf` configuration snippet will suffice:

```
ProxyPass / http://127.0.0.1/  
ProxyPassReverse / http://127.0.0.1/  
RequestHeader set REMOTE-USER %{REMOTE_USER}
```

Enabling Gerrit HTTP authentication

Now we will need to execute Gerrit init again in order to select an HTTP-based authentication, relying on an external trusted-user SSO validation.

```
$ /opt/gerrit/bin/gerrit.sh stop
```

```
Stopping Gerrit Code Review: OK
```

```
$ cd /opt/gerrit
```

```
$ java -jar gerrit.war init
```

```
[...]
```

```
*** User Authentication
```

```
***
```

```
Authentication method           [HTTP/?]: http RETURN
```

```
Get username from custom HTTP header [y/N]? y RETURN
```

```
Username HTTP header           [SM_USER]: REMOTE_USER RETURN
```

```
SSO logout URL                  : http://sso.mydomain.com/logout RETURN
```

```
[...]
```

From this point on, Gerrit will *only* rely on the REMOTE_USER HTTP Header to get a trusted username without performing any extra authentication steps.

User profile lookup

The pure HTTP authentication method suffers from not having control/visibility of the third-party user registry. Gerrit only has the option to present the user registration screen and ask the user for that data directly when they are logging in for the very first time.

Often, the underlying user registry is still LDAP, even if the frontend authentication is performed elsewhere. In this case a combination of HTTP and LDAP lookup can help to cover the last steps.

The HTTP_LDAP authentication method is a mix of the HTTP/SSO authentication scheme plus the standard lookup, without authentication in an LDAP user registry. LDAP configuration is similar to the one previously described; during the Gerrit init steps we just need to specify `http_ldap` as authentication method.

Summary

Our Gerrit installation at the end of this chapter has evolved from an insecure sandbox to a standard setup integrated with an external authentication system.

We have also covered the three main methods of user authentication: over the Internet through an OpenID provider such as Google accounts, in a private network using a standard LDAP or Active Directory, and finally integrated with an existing SSO system using a trusted HTTP reverse proxy.

4

SSH and HTTPS Access

In this Chapter we are going to enforce the security of our Gerrit installation using security protocols, such as, SSH and HTTPS to secure the Git protocol and Gerrit Code Review actions through the web-UI.

By the end of this chapter, the Gerrit instance configuration will be ready to be used either internally or over the Internet with the appropriate level of security.

Enabling strong security on Gerrit

The default **Java Virtual Machine (JVM)** does not allow the execution of a cryptography algorithm with strong security; however, modern Security Requirements over public networks such as the Internet require the JVM to be unlocked to make it more resilient to reverse-engineering, hacking, or attacks by malicious external users. The unlock process is based on the following manual steps:

1. Stop Gerrit and check the JVM:

We may have more than one JVM installed in the system so we need to make sure we unlock the one specifically used by Gerrit:

```
$ /opt/gerrit/bin/gerrit.sh stop
[...]  
$ git config -f gerrit.config container.javaHome  
/usr/java/jdk1.6.0_21/jre
```

2. Get and install "Unlimited Strength Jurisdiction Policy Files."

Oracle, the owner of the JVM, does not allow the automatic download of the files necessary for our use. We need to manually search the internet for the location of these files, and download them manually through the browser once the Oracle Terms and Conditions for Cryptography exports have been accepted.

The unlock kit is composed of two files that need to be replaced in the Java security libraries installation.

```
$ cd /tmp
$ unzip jce_policy-6.zip
[...]
$ cp jce/*.jar /usr/java/jdk1.6.0_21/jre/lib/security/.
```

Our JVM is now fully enabled to execute strong crypto-algorithms (such as AES) with high-strength keys (over 128 bits) to prevent attacks and reverse engineering of the data that is transferred over the network.

Installing Bouncy Castle Security

During the initial sandbox setup, Gerrit asked for the download and installation of an additional security library named Bouncy Castle, which is an open source Java implementation of the Java Cryptography Extensions, alternative to the standard one provided with the JVM. This library provides a richer set of strong encryption algorithms that can be freely used to secure the Gerrit communication over the Internet.

Whenever the Gerrit Security provider is modified or upgraded, the generated server SSH keys stored under Gerrit `/etc/ssh_host_key` need to be removed and re-created again, as they were based on weak security algorithms.

```
$ rm /opt/gerrit/etc/ssh_host_key
```

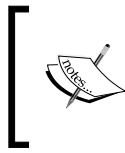
We can now run the Gerrit init steps again and answer `y` when asked to download and install the Bouncy Castle library. Once completed, the new server SSH keys will be generated using a set of higher security algorithms with a longer key length.

```
$ cd /opt/gerrit
$ java -jar gerrit.war init

*** Gerrit Code Review 2.7-rc4
***

[...]
```

```
Gerrit Code Review is not shipped with Bouncy Castle Crypto v144
  If available, Gerrit can take advantage of features
  in the library, but will also function without it.
Download and install it now [Y/n]? y RETURN
Downloading http://www.bouncycastle.org/download/bcprov-jdk16-144.jar ...
OK
Checksum bcprov-jdk16-144.jar OK
Generating SSH host key ... rsa... dsa... done
```



Gerrit tries to download the library attempting a direct connection to the Internet, without the usage of an external HTTP proxy. Should this download fail, you can download the file manually and store it under the `/opt/gerrit/lib` directory.

Once Gerrit is upgraded to use Strong Security, it can be restarted.

```
$ /opt/gerrit/bin/gerrit.sh start
Starting Gerrit Code Review: OK
```

Using SSH with Gerrit

SSH is a much more robust and secure authentication system in the tunneling of commands and it is the most widely used protocol combination used with Git over the Internet.

Git/SSH Client keys

Git secure shell protocol is built on top of OpenSSH, the most widely used open source implementation of the SSH protocol and this is why Gerrit uses OpenSSH keys by default. When using other SSH implementations (that is Putty on Microsoft Windows), keys need to be converted in OpenSSH format before being inserted in Gerrit.

In order to start using Git/SSH we need to generate a key-pair (public key + private key) using either RSA or DSA algorithms. The public key (that is `id_rsa.pub`) will be used for the user identification while the private key (that is `id_rsa`) will be kept on the local system as a secret credential and must never be sent to anyone.

The following OpenSSH command will generate a 2048 bits RSA key-pair:

```
$ ssh-keygen -t rsa -b 2048
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase): RETURN
Enter same passphrase again: RETURN
Your identification has been saved in /home/user/.ssh/id_rsa.
Your public key has been saved in /home/user/.ssh/id_rsa.pub.
The key fingerprint is:
22:67:7c:f2:10:19:b4:e6:22:ac:1a:6f:15:53:3f:e1 user@mydomain.com
```



A passphrase is a quite-long secret composed by a series of words known only to the user. It is optional and can be used for deriving an encryption key used to lock and unlock the access to the SSH private key when needed by Git. See <http://en.wikipedia.org/wiki/Passphrase> for more information on how to select a strong passphrase

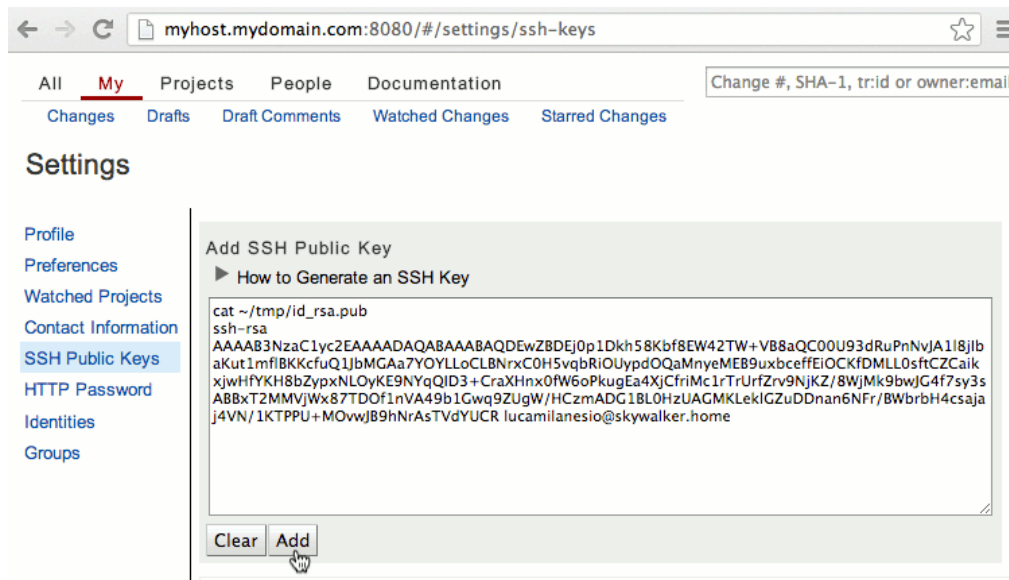
Adding SSH keys to a Gerrit user profile

The public SSH key generated under `/home/user/.ssh/id_rsa.pub` needs to be uploaded to Gerrit and associated to users's profile. During the SSH handshake user's private SSH key will be used to demonstrate the user's identity against the public key archived in the Gerrit profile.

The SSH public key is Base64 encoded and can be simply copied and pasted into the Gerrit SSH key field on the web-UI.

```
$ cat /home/user/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQDEwZBDEj0p1Dkh58Kbf8EW42TW+
VB8aQC00U93dRuPnNvJA1l8jIbaKut1mflBKKcfuQ1JbMGAA7YOYLLoCLBNrxC0H5
vqbRiOUypd0QaMnyeMEB9uxbceffeioCKfDMLL0sftCZCaikxjwHfYKH8bZypxNLO
yKE9NYqQID3+CraXHnx0fW6oPkugEa4XjCfriMc1rTrUrfZrv9NjKZ/8WjMk9bwJG4
f7sy3sABBxT2MMVjWx87TDOflnVA49b1Gwq9ZUgW/HCzmADG1BL0HzUAGMKLek
lGZuDDnan6NFr/BWbrbH4csajaj4VN/1KTPPU+MOvwJB9hNrAsTVdYUCR
user@mydomain.com
```

The SSH Public Keys section of the Gerrit user's profile allows you to add a new SSH key and paste the content previously displayed. The following screenshot shows Gerrit SSH Keys in the user profile screen:



Cloning a repo over Git/SSH

Next we need to go back to the Gerrit projects view where we can select the `hello-project` previously created and clone it using the Git/SSH command displayed.

The general format of Git/SSH URLs with Gerrit is as follows:

```
ssh://username@hostname:port/project
```

The first time that a Git clone over SSH is performed against a Gerrit server, the RSA key fingerprint of the remote key would need to be checked and confirmed.

```
$ git clone ssh://user@myhost.mydomain.com:29418/hello-project
```

```
Cloning into 'hello-project'...
```

```
The authenticity of host '[myhost.mydomain.com]:29418  
([127.0.0.1]:29418)' can't be established.
```

```
RSA key fingerprint is 82:f9:73:76:72:11:9c:97:55:d8:2d:16:01:53:2a:2b.
```

```
Are you sure you want to continue connecting (yes/no)? yes RETURN
```

```
Warning: Permanently added '[myhost.mydomain.com]:29418' (RSA) to the  
list of known hosts.
```

```
remote: Counting objects: 2, done
```

```
remote: Finding sources: 100% (2/2)
```

```
Receiving objects: 100% (2/2), 217 bytes, done.
```

```
remote: Total 2 (delta 0), reused 0 (delta 0)
```




The RSA key fingerprint shown at the time we do the clone over SSH to Gerrit, will not correspond to the Operating System SSH Keys. Git/OpenSSH client will have to ask for confirmation of the fingerprint expected from the remote system and store the corresponding public SSH Key into the user's `~/.ssh/known_hosts` file.

The expected Gerrit RSA fingerprint can be verified using the following command on the Gerrit server:

```
ssh-keygen -l -f ~/opt/gerrit/etc/ssh_host_rsa_key.pub
```

In contrast to a regular Git/SSH server implementation, the Gerrit SSH protocol stack limits the available actions to only the ones strictly needed for its own Git protocol, administrative commands, and for enforcing its own security model. In order to experiment with the SSHD layer directly, we can try to activate a remote shell using SSH to connect to Gerrit:

```
$ ssh -p 29418 user@myhost.mydomain.com
```

```
****      Welcome to Gerrit Code Review      ****
```

```
Hi User, you have successfully connected over SSH.
```

```
Unfortunately, interactive shells are disabled.
```

```
To clone a hosted Git repository, use:
```

```
git clone ssh://user@myhost.mydomain.com:29418/REPOSITORY_NAME.git
```

```
Connection to myhost.mydomain.com closed.
```

As you can see in the preceding code, we do not get any response from the underlying operating system but we do have a response from Gerrit showing that our user has been authenticated and the user profile fetched from the Gerrit DB. The first line `Hi User`, is the clue that the Gerrit authentication was successful.



Should the `ssh` command fail, it is recommended to run it again with the additional `-v -v -v` option (yes, three times) in order to display the fully verbose SSH handshake output and error description.

Enabling HTTPS

The access to the Gerrit web-UI over the Internet must also be secure as we do not want the data, cookies, and credentials of users connecting to Gerrit to be eavesdropped, captured, or modified by anyone in transit.

HTTP/S reverse-proxy to Gerrit

This configuration is typically chosen when an existing authentication/security proxy is already in place and used by the company for any incoming connections from the Internet. The Gerrit server listens on the port 8080 of the loopback devices (127.0.0.1) to avoid any external access over an insecure HTTP protocol. Apache (or any other HTTP/S reverse-proxy) sits in front of Gerrit and exposes an HTTPS protocol to the external interface, forwarding all the calls to the loopback interface.

The Gerrit configuration `/opt/gerrit/etc/gerrit.config` needs to be edited as follows:

```
[...]
[httpd]
    listenUrl = proxy-https://127.0.0.1:8080/
[...]
```

On the Apache configuration `/etc/httpd/conf/httpd.conf` we will need to define a Virtual Host with SSL enabled and the reverse proxy rules to redirect the calls to Gerrit:

```
[...]
LoadModule ssl_module modules/mod_ssl.so
LoadModule mod_proxy modules/mod_proxy.so
<VirtualHost _default_:443>
    SSLEngine on
    SSLProtocol all -SSLv2
    SSLCipherSuite ALL:!ADH:!EXPORT:!SSLv2:RC4+RSA:+HIGH:+MEDIUM:+LOW
    SSLCertificateFile /etc/pki/tls/certs/server.crt
    SSLCertificateKeyFile /etc/pki/tls/private/server.key
    SSLCertificateChainFile /etc/pki/tls/certs/server-chain.crt
```

```
ProxyPass / http://127.0.0.1:8080/
ProxyPassReverse / http://127.0.0.1:8080/
</VirtualHost>
[...]
```

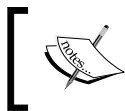
The X.509 Certificates enrollment process for the HTTP/S server key-chain is beyond the scope of this book; detailed reference information can be found at <http://www.openssl.org/docs/apps/x509.html#>.

The following example helps by creating an initial self-signed certificate to allow us to complete the installation (in the example, a 2048 bits RSA key certificate valid for 365 days):

```
$ openssl req -x509 -days 365 \
  -nodes -newkey rsa:2048 \
  -keyout /etc/pki/tls/private/server.key -keyform pem \
  -out /etc/pki/tls/certs/server.crt -outform pem
```

Generating a 2048 bit RSA private key
writing new private key to 'server.key'
[...]

```
-----
Country Name (2 letter code) [AU]:GB RETURN
State or Province Name (full name) [Some-State]: London RETURN
Locality Name (eg, city) []: London RETURN
Organization Name (eg, company) []: MyCompany RETURN
Organizational Unit Name (eg, section) []:RETURN
Common Name (eg, YOUR name) []:myhost.mycompany.com RETURN
Email Address []:admin@myhost.mycompany.com RETURN
```



Remember to always include the exact full hostname of your HTTP/S server in the Common Name line, and it needs to be the same one used by users in their browser URL.

As we do not have an official X.509 Certificate signed by a Certification Authority, we do not have a server Certificate Chain to populate. We can now simply copy the self-signed X.509 Certificate just generated into the Certificate Chain.

```
$ cd /etc/pki/tls/certs/
$ cp server.crt server-chain.crt
```

HTTP/S Support in Gerrit

If there are no requirements to use an external HTTP/S server, the easiest option is to reuse the internal one provided by Gerrit. Having previously enabled the strong security libraries on the JVM, this allows us to leverage the strong crypto algorithms on the SSL layer.

In order to enable SSL security on HTTP, we will need to stop Gerrit and restart the init process:

```
$ /opt/gerrit/bin/gerrit.sh stop
```

```
Stopping Gerrit Code Review: OK
```

```
$ cd /opt/gerrit
```

```
$ java -jar gerrit.war init
```

```
*** Gerrit Code Review 2.7-rc4
```

```
***
```

```
[...]
```

```
*** HTTP Daemon
```

```
***
```

```
Behind reverse proxy      [Y/n]? n RETURN
```

```
Use SSL (https://)        [y/N]? y RETURN
```

```
Listen on address         [*]: RETURN
```

```
Listen on port             [8080]: 8443 RETURN
```

```
Canonical URL              [http://myhost.mydomain.com:8080/]:
```

```
https://myhost.mydomain.com:8443 RETURN
```

```
Create new self-signed SSL certificate [Y/n]? y RETURN
```

```
Certificate server name    [myhost.mydomain.com]: RETURN
```

```
Certificate expires in (days) [365]: RETURN
```


```
$ /opt/gerrit/bin/gerrit.sh start
```

```
Starting Gerrit Code Review: OK
```

When we open Gerrit using the HTTPS URL (`https://myhost.mydomain.com:8443`) we will receive a browser warning on the X.509 Certificate identity; Chrome displays the following warning message: *The identity of this website has not been verified; server's certificate is not trusted*. This is expected for a self-signed X.509 Certificate. The same problem may occur when we use Git/HTTPS for cloning the repositories hosted on Gerrit.

A temporary workaround to this is to relax the X.509 Certificate validation by setting `http.sslVerify` to `false` in the Git configuration file:

```
$ git config --global http.sslVerify=false
$ git clone https://user@myhost.mydomain.com:8443/hello-project
Cloning into 'hello-project'...
remote: Counting objects: 2, done
remote: Finding sources: 100% (2/2)
remote: Total 2 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (2/2), done.
```

 Never ever use this workaround for a production installation. Relaxing the X.509 Certificate Validation introduces the risk of "man-in-the-middle" attacks (http://en.wikipedia.org/wiki/Man-in-the-middle_attack).

Summary

In this chapter we have reinforced the overall security of the Gerrit installation by unlocking the JVM and using Strong Encryption provided by the Bouncy Castle Java Library.

We started using the Git/SSH protocol to connect to Gerrit repositories and now understand the basic concepts and operations required. Also we have introduced the SSL Encryption layer to the Git/HTTP protocol support in Gerrit and have seen how to enable it with and without the adoption of an Apache HTTPS reverse-proxy.

5

Editing Your Project Permissions

In order to start organizing the projects and teams for a Code Review, we will need to explore the Gerrit permissions scheme and the association of **ACLs (Access Control Lists)** to Git references. We will also learn how to effectively use project templates and groups in order to organize role-based access for our repositories. By the end of this chapter our Gerrit installation will be well organized with the main Gerrit roles defined as groups and ready to be used for a real-life project.

Understanding the Gerrit permission scheme

Git does not enforce any permission or control over the operations performed on the repository, which could be acceptable when managing a local individual clone but it is definitely not enough when sharing server repository with multiple people. The ability of Gerrit to define and enforce access controls makes it a perfect solution, even for just serving a Git repository with associated permissions.

Although permission gives you the ability to allow or deny someone access to perform one or more actions on one or more resources, not all Gerrit permissions are about enforcing security; some of them are meant to drive and organize the Code Review flow across all different roles involved.

A permission is a pyramid composed of:

- **Subject:** Single or multiple sets of people identified by Gerrit.
- **Action:** The ability to allow or deny a specific operation.
- **Resource:** Single or multiple sets of Gerrit objects (typically Git reference) that are controlled by the permission.

Gerrit does not allow the use of individual users as a subject and always requires a group to be defined. This could be viewed as a limitation but in reality it is a powerful way to organize the way access to resource is controlled, by abstracting the subject into logical "roles" instead of managing a long list of ACLs to individuals.



For those who want to assign permissions to individual users, see the `singleusergroup` plugin that provides a surrogate group for each individual users, as described in the following external groups section.

Actions can be a positive (ALLOW) or negative (DENY or BLOCK) approval for performing some Git or Code Review operations. Gerrit is superior to many other Git server implementations because of its ability to express in detail the operations to be controlled.

Resources are either an entire system (for global capabilities permissions), a whole project (for delegation of administration control), or a specific Git reference (path-based name of a Git object namespace). All references can be specified using a simple but powerful wildcard notation or alternatively a complex regular expression.

The ability to delegate the control of the administration of a project makes Gerrit really scalable across large organizations, where the Gerrit system administration would not be able to follow the business as usual of hundreds or even thousands of access control lists.



In some cases we may want to prevent the override of permissions schemes from child projects. The BLOCK directive is designed for enforcing certain site-wide policies on sensitive operations such as creating and deleting tags or branches.

Another important concept to introduce is inheritance. Gerrit has been designed to organize development and minimize the management effort required as the number of projects and people grows over time. A child project or sub-group can inherit settings from their parent: once a default scheme has been defined it can be automatically taken and reused for existing and future projects. A project can derive its access permissions scheme from only one parent project.

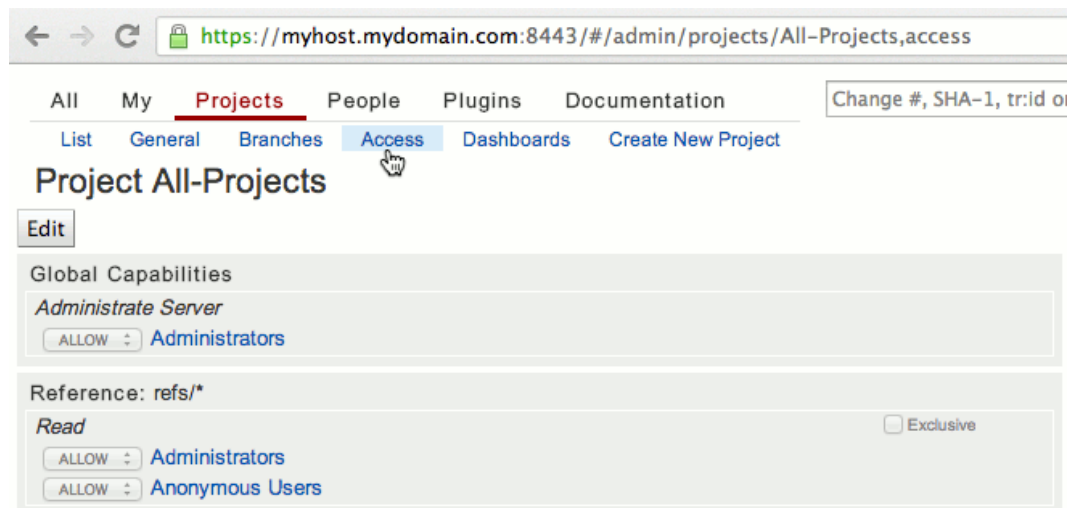
Configuring project permissions

We will cover only the main permissions currently available on Version. 2.7 and how to effectively use project permissions to organize the Gerrit installation so that it can be more flexible and manageable.

A more comprehensive description of all permissions associated to a project, which may vary from one version of Gerrit to the next, is available in the documentation pages included in your Gerrit web-UI (for example: <https://myhost.mydomain.com:8443/Documentation/access-control.html>).

All-Projects permissions

All-Projects is automatically created when Gerrit is initialized the first time; it is not a real project per se but it represents the default set of permissions applied to all of the projects that will be created in Gerrit. Permissions can be accessed selecting the **Access** link and are visible only to the Gerrit administrator, which is the first user created after the initial setup. The following screenshot shows access control screen for **All-Projects**:



All-Projects includes two sets of permissions, they are as follows:

- **Global Capabilities:** Allows you to assign and delegate some of the Gerrit administration tasks to other groups. This set of permissions are available uniquely for All-Project as it is not relevant for actual projects. (for example, Killing tasks or creating users)
- **Reference Permissions:** Allows you to grant Git and Code Review permissions on a per reference basis (see next section for more details on Git references). This type of permission can be found in every project and we will go through this in further detail.

To customize the permissions we first need to click on the **Edit** button, which will trigger a new **Add Permission...** radio button to appear at the bottom of each set of permissions. Alternatively, it is possible to extend an existing assigned permission to additional groups by clicking on the **Add Group** link or by adding an additional reference spec by name.

Git and Gerrit references

A Git reference (aka ref-spec) is a path-based syntax for identifying a name space of Git objects; Gerrit extends that namespace with additional paths to represent and store its own information about changes and reviews.

Standard Git References are:

refs/heads/*	All the branches of the Git repository, represented as a path-based structure. refs/heads/master is the main initial branch of development.
refs/tags/*	All the tags assigned inside the Git repository, represented as a path-based structure.

Main refs-specs added by Gerrit for its internal settings and Code Review are:

refs/changes/*	All the changes uploaded to Gerrit, represented in the format: <Last two digits of change>/<change Nr>/<patch set Nr>.
refs/meta/config	The Gerrit internal project configuration file, including security, groups, and submit rules.
refs/for/* refs/publish/*	The Gerrit magic branch for pushing changes for review.
refs/drafts/*	The Gerrit magic branch for pushing draft changes, visible only to the change author and the invited reviewers.

Git permissions

Gerrit includes a fully featured Git server; in addition to its Code Review functionalities it can also be used to serve a repository in a secure and controlled way.

With regards to Git access control it offers a rich and fine-grained set of permissions:

Read	Allows you to clone a branch.
Push (+Force)	Allows you to perform a fast-forward (or +forced non-fast-forward) push to an existing branch. When used with +Force it allows you to remove a branch or other reference (that is tags)
Create Reference	Allows you to create a new branch. When used on refs/tag/* it allows you to create a lightweight non-annotated tag.
Push Signed tag Push Annotated tag	When used on refs/tag/* it allows you to push a signed (or annotated) tag.
Forge Author	Allows you to impersonate the author of the change.
Forge Committer	Allows you to impersonate the person that created the commit.
Forge Server	Allows you to impersonate Gerrit server identity that created the commit. Needed when uploading commits created by Gerrit (for example, merge commits created on the server)
Owner	Allows full administration of the project when assigned to refs/*. Provides the ability to delegate the editing of access permissions when assigned to a more specific ref-set. (For example, Owner of refs/head/master means that you can change access permissions on the master branch.)

Code Review permissions

There is a rich set of additional permissions introduced by Gerrit, which allows you to control who can participate in the Code Review process.

The basic set of permissions are still Git permissions but applied to the Gerrit "magic branches" used for Code Review. We will repeat them again here to provide clarity on how they are expressed and what their effect is on the project:

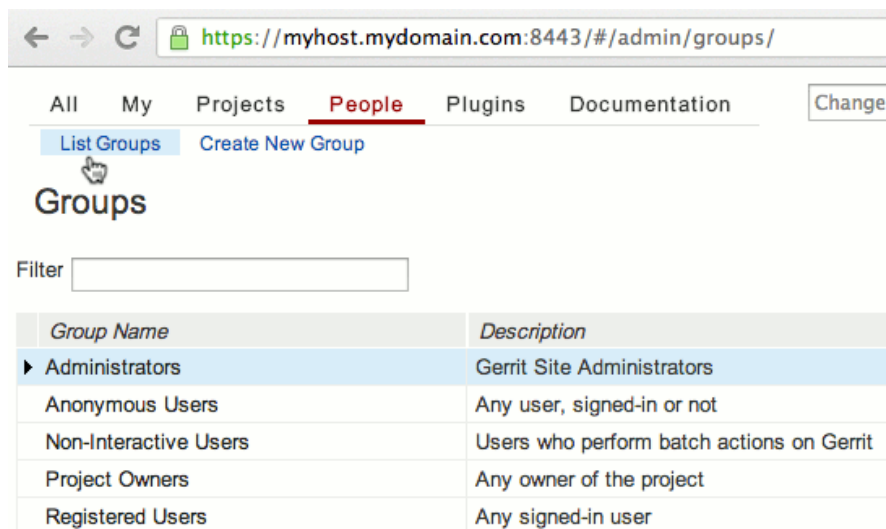
Create Reference	refs/for/refs/*	Allows you to contribute a new change for review.
Push	refs/for/refs/*	Allows you to provide an additional patch-set to an existing opened change.

This additional set of permissions is specifically related to the Code Review:

Label-<LabelType> -X..+Y	Allows the posting of a review and the provision of a score from -X to +Y to an open change. Applies to Gerrit Code Review labels such as, Code Review or any additional label configured on that project. In Gerrit 2.6 the only default pre-defined label is Code Review although others can be added, such as the historical verify used in the Android open source project.
Submit	Allows you to accept and submit a change for merge.
Abandon	Allows you to abandon a change.
Push Merge Commits	Allows you to merge an accepted change into the target branch.
Rebase	Allows you to rebase from the Gerrit web-UI an open change into its target branch.
Publish Drafts View Drafts Delete Drafts	Allows you to push (view or delete) change drafts, changes that are visible only to their authors and invited reviewers.

Managing groups

Permissions are assigned to groups in order to be effective for the team members. Groups can be listed and edited from from the **People** menu as shown in the following screenshot:



The default groups configured during the Gerrit initialization are:

- **Anonymous Users:** Identifies any possible user that could access Gerrit, either authenticated or anonymous. Granting read permission to anonymous users means that you have made the project public.
- **Registered Users:** Identifies the users that have successfully completed an authentication on the system and are registered inside the Gerrit DB. Granting read permissions to registered users while denying any access to anonymous users means the project is private and not visible from outside of the team.
- **Project Owners:** Identifies any users that have been defined as owner of a project once it has been created. Project owners are typically the users delegated to the administration of the project and activities include the management of its branches and the definition of the project ACLs.
- **Administrators:** Gets populated automatically with the first account registered in Gerrit and granted administration permissions on the entire Gerrit instance. Members of the administrators group can delegate their role to other users by inserting them as an additional member of the group.
- **Non-Interactive Users:** Gets generated automatically when the Gerrit DB is created but is typically empty. When users get created for batch administration (using the `gerrit create-account` command) they can be assigned to this group.

Custom internal groups

It is possible to create an arbitrary number of custom groups by using the **Create New Group** link: those are referred as internal groups, as they are stored and maintained using the Gerrit internal DBMS.

Each internal group is defined by the following details:

- **Name and description:** The name and description assigned to the group.
- **Owners group:** The group of users delegated to the administration of the group.
- **Visibility:** Defines whether the group should be visible to all users or just to its members.
- **Members and included groups:** The list of individual members and other groups' members that belong to the group. Usage of inclusion of other groups allows you to define group hierarchies.

External groups (group backends)

Gerrit can benefit from automatically fetching group information from other external systems (for example, LDAP). This is great functionality as companies can reuse their existing groups for use in the definition of the templates and access control of Gerrit projects.

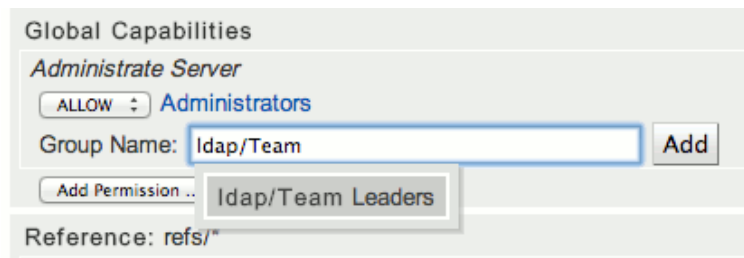
External groups are not visible from the groups list screen, as they potentially belong to a huge domain space, and showing all of them would create an unnecessarily long list. They are however, all proposed in Gerrit when a group name is requested through an auto-completion text box.

For example, when assigning access permissions to projects, we can refer to an LDAP group instead of typing the name of an internal or system Gerrit group and then the magic happens! Once an external group is referenced anywhere in Gerrit, the user-to-group membership will be automatically resolved on-demand by requesting the external group system whenever needed.

The name format for external groups is: `groupBackend/groupName`.

For example, the LDAP groups are available with the group backend `ldap`. Using the group name `ldap/Team Leaders` in Gerrit will result in searching through the configured LDAP authentication system for a group with name "Team Leaders".

The following screenshot shows adding external LDAP groups:



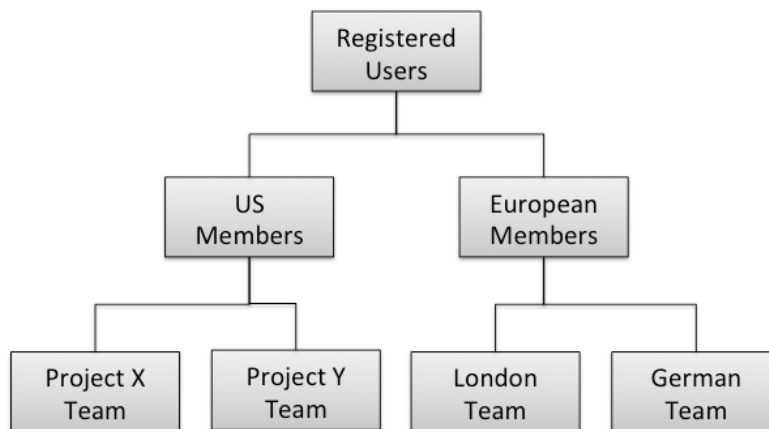


External groups can be extended through one of the group backend extensions provided by Gerrit. A very popular one is provided by the `singleusergroup` plugin (see <https://gerrit-review.googlesource.com/#/admin/projects/plugins/singleusergroup>). It exposes any individual registered Gerrit users as a "surrogate group with one user" in the form: `user/username`. Using this, the constraint of having to assign permissions only to groups can easily be overcome.

Using group hierarchy effectively

We will need to create additional groups in Gerrit in order to have a more granular partitioning of Gerrit users that are split into teams and roles.

Gerrit groups can be organized in hierarchies to better reflect the structure of the company and at the same time avoid repeating a lot of grants for a number of teams in a single division, as shown in the following screenshot:

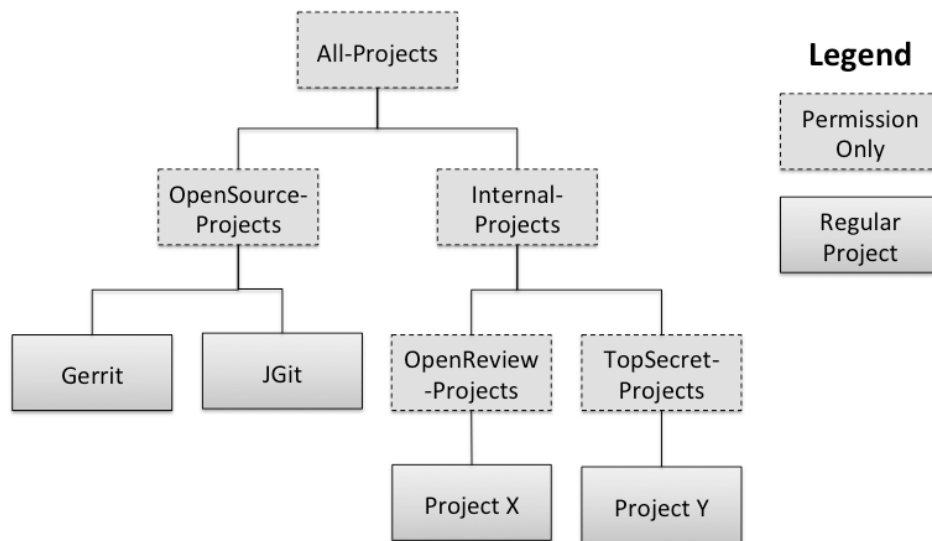


Please note that it is not always necessary to have one group per project. Depending on the policies of the company and the confidentiality of the project, groups can be more or less granular and even associated to a specific location.

Groups can also be used to indicate different roles inside the team. It is common practice to define the Code Review roles as groups and assign to them the fine-grained permissions on the projects.

Organizing project security templates

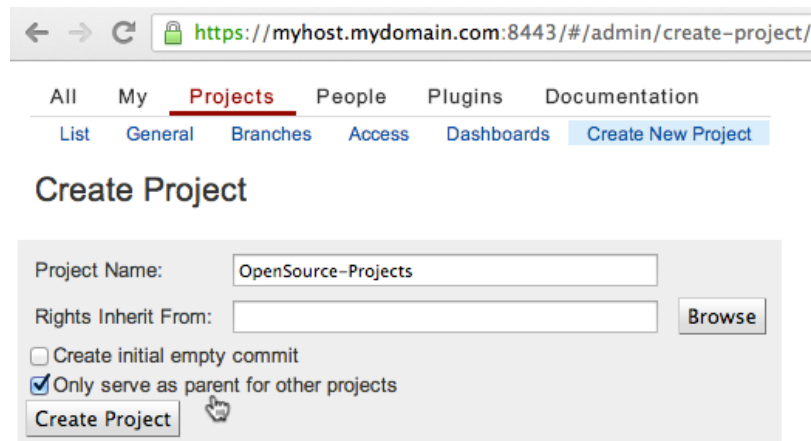
Similarly to group organization, projects can be organized in a hierarchy as well. All-Projects is already an abstract concept, representing any project that is defined in Gerrit. Whenever a new project is created, it will need to be linked to a parent in order to inherit its settings by default. In contrast to groups, projects can have only a linear tree-based hierarchy, allowing it to inherit settings from only one parent, so a matrix-based organization is not possible. The following diagram shows sample project hierarchy:



There are two types of project mentioned in the diagram, which are explained in detail as follows:

- **Permission-only:** These are used only as a template for other projects, and they are not intended to be used as end-user projects, but exclusively a set of permissions and settings to be inherited as default settings by its children.
- **Regular Projects:** These are real Gerrit projects that contain code and reviews. They typically inherit all permissions from its parent project; additional settings are defined by exception.

Permissions-only projects differs from other projects in that they only serve as parent for other projects flagged on their settings page. The following screenshot shows creating a permission-only project:

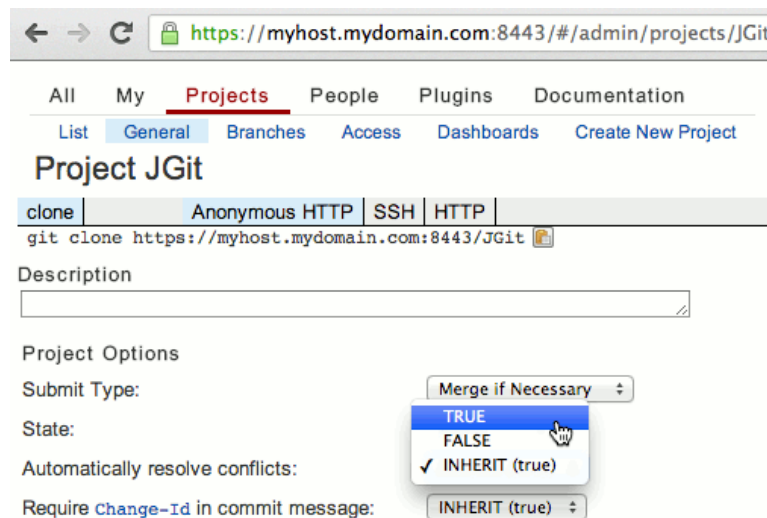


The screenshot shows a web browser window with the URL `https://myhost.mydomain.com:8443/#/admin/create-project/`. The navigation bar includes links for All, My, **Projects**, People, Plugins, and Documentation. Below this, there are sub-links: List, General, Branches, Access, Dashboards, and **Create New Project**. The main heading is 'Create Project'. The form contains the following fields and options:

- Project Name:** A text input field containing 'OpenSource-Projects'.
- Rights Inherit From:** A text input field that is currently empty, with a 'Browse' button to its right.
- ☐ Create initial empty commit
- ☒ Only serve as parent for other projects
- Create Project** button

In order to create a projects hierarchy we need to select the parent project by filling the field **Rights Inherit From**: if left blank it is assumed that All-Projects as parent.

In the project settings page, it is possible to override when necessary the default values inherited by its parent, as shown in the following screenshot:



The screenshot shows a web browser window with the URL `https://myhost.mydomain.com:8443/#/admin/projects/JGit`. The navigation bar is the same as the previous screenshot. The main heading is 'Project JGit'. Below the heading, there are tabs for clone: clone, Anonymous HTTP, SSH, and HTTP. The 'clone' tab is selected, showing the URL `git clone https://myhost.mydomain.com:8443/JGit`. Below this is a 'Description' text area. The 'Project Options' section includes the following settings:

- Submit Type:** A dropdown menu with 'Merge if Necessary' selected.
- State:** A dropdown menu with 'TRUE' selected.
- Automatically resolve conflicts:** A dropdown menu with '✓ INHERIT (true)' selected.
- Require Change-Id in commit message:** A dropdown menu with 'INHERIT (true)' selected.

Similarly all the other settings (accessible through the **Access** and **Dashboards** links) are inherited by its parent and can be overridden by the child project.

Summary

In this chapter we have explored the Gerrit project authorization system and now understand the value it brings to the organization of teams and projects.

We have seen that Gerrit can provide significant value even simply as Git server with ACLs with its ability to provide fine-grained access control to repository branches and Git commands. Additionally we have seen the additional permissions that Gerrit grants to group of users in order to control the Code Review workflow.

Gerrit groups are a powerful concept and are divided into system, internal, and external. We have analyzed and understood the use of the main system groups and how to extend them by creating new internal ones or integrating with external systems, such as LDAP. Finally we started to organize our projects into a hierarchy in order to create security templates that can be reused across the company.

We are ready now to start creating our first fully featured Code Review project and its associated review roles.

6

Changes and Code Review Workflow

In this chapter we will provide an overview of how Code Review workflow works with Gerrit, by going step-by-step through an initial change review. We will explain the jargon used by reviewers and how to interact with reviewers and contributors without creating conflicts or having changes lost in the repository.

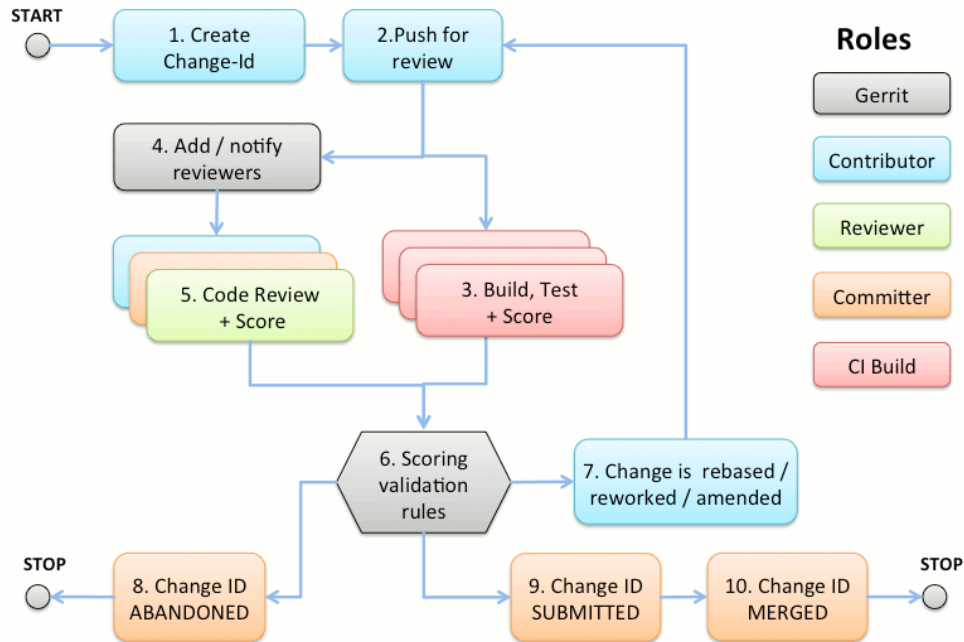
By the end of this chapter we will have completed a full review cycle with the ability to push and amend changes based on the team's feedback.

Gerrit Code Review roles and workflow

Gerrit was originally designed for developing the Android open source project and then collaboratively extended by additional contributors. The review workflow introduced (<http://source.android.com/source/life-of-a-patch.html>), has become very popular because it is the first large scale Code Review in the software industry that has been experimented. We will use a set of roles that come from our experience of collaboration on many projects including the development of Gerrit itself, bearing in mind that they are completely customizable in terms of names and workflow, and can be easily adapted to accommodate different needs.

Review workflow step-by-step

The following diagram shows a high-level view of the Gerrit review roles and how they participate in the Code Review workflow with Gerrit:



Roles overview

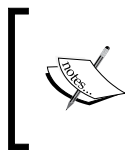
See the following table for the roles that we will use in the Code Review workflow:

Contributor	Any internal or external member of the team who uploads a commit for review.	For example, John Doe <john.doe@mycompany.com>
Reviewer	Any internal or external member of the team who is allowed to post comments on a change and provide a score.	For example, Luca Milanesio <luca@milanesio.org> John Doe <john.doe@mycompany.com>
Committer	Senior team member allowed to grant a veto or approve, submit, and merge a change.	For example, Luca Milanesio <luca@milanesio.org>
Build (optional)	Batch robot user allowed to fetch a change and provide an automated review.	For example, Jenkins Continuous Integration build

Workflow in ten steps

Following are the ten steps for creating a workflow:

1. The contributor creates a commit and assigns a new unique global **Change-Id** through the `commit-msg` hooks previously downloaded from Gerrit.
2. The contributor pushes to the Gerrit review branch, creating a new change for review. Gerrit assigns a unique URL in order to access and review the change using its web-UI.
3. This step is optional. Continuous Integration builds and verifies the change: the builder fetches and triggers a build to check if the change actually works. Feedback is reported back as a positive (code builds and works) or negative (code doesn't work or build is broken) score.



The involvement of a builder step is optional, but highly recommended because it allows other reviewers to save time by preventing reviews of potentially flawed changes and prevents instability by validating changes early in the process, thus avoiding broken builds.

4. In the Add/notify reviewers step, a contributor invites one or more reviewers to look at the code and provide their feedback. Additional people can add themselves spontaneously to review or just watch the evolution of the change under review. For existing changes with additional patch-sets, Gerrit automatically notifies all of the existing reviewers and watchers that a new review is required.
5. In the Code Review with score step, all the invited reviewers and other people allowed by project permissions (reviewers, contributors, and committers) can see and fetch the change and provide their feedback. It can be expressed using a two-way diff on the Gerrit web-UI or using a client tool such as Eclipse. Feedback is composed of inline comments on the code with an additional comment on the overall patch-set. A review can be tagged with a positive or negative score, ranging from -2 to +2.
6. Gerrit has a set of rules for getting a change approved based on the overall feedback and scoring received. Default Gerrit rules require at least one Code Review/ + 2. Rules can be tailored to each project's needs and can even have a different set of approval conditions or be based on additional custom labels (for example, verified + 1 as in the Android Open Source Project review workflow). The result of validation rules is the enablement of the submit action on the Gerrit web-UI and the subsequent permission to merge the change to its target branch.

7. When a change does not reach the minimum score to be approved, it will need to be reworked by its contributor. The typical types of reworking are **rebase** (from the time it was created, the change has become outdated) and **amend**. Comments have been made on the code that needs to be taken into consideration for getting all of the issues resolved. In the first case, a **Rebase** button is available on the Gerrit web-UI, so that a Contributor can immediately rebase the change. Should the rebase fail or if reworking is needed, the change will be rebased and pushed again for review and the workflow resumes from step 2.
8. If the review feedback is negative, the change can be abandoned so that other people do not spend further effort reviewing it. Abandoned changes are not removed from the Git repository and can be resumed at a later stage by restoring it.
9. The committer can submit a change once it has successfully satisfied the minimum conditions for approval. The submission automatically triggers the merge step.
10. Gerrit merges the change onto its target branch, as a consequence of the previous submit operation. Should the merge fail because of a conflict, the change has to be reworked again and validation has to resume from step 2.

Gerrit branch namespace for code review

We can now experiment with a real-life review cycle by putting into practice the workflow already described. The only action we will not explore at this stage is the CI Build integration, which will be covered later in a dedicated Appendix.

The first action that triggers a Code Review is the initial commit pushed to Gerrit. Gerrit introduces a new Git reference namespace completely dedicated to the Code Review prefixed by `refs/for` before the branch path (for example, for pushing a change for review on the master branch we need to use `refs/for/master`).

We can then create our first change on the `hello-project` we cloned in our initial sandbox.

```
$ git clone \
  ssh://jdoe@myhost.mydomain.com:29418/hello-project
[...]
```

Receiving objects: 100% (2/2), done.



Remember to set the Git author identity, aligned with the one present in Gerrit, otherwise, the push will be rejected:

```
$ cd hello-project
$ git config user.name "John Doe"
$ git config user.email "john.doe@mycompany.com"
```

Additionally, we need to set up the Gerrit `commit-msg` hook for generating a unique change ID. We are using `curl` in the following examples, but you can use any other different web client for downloading the `commit-msg` hook from Gerrit.

```
$ curl -Lko .git/hooks/commit-msg \
https://myhost.mydomain.com:8443/tools/hooks/commit-msg
$ chmod +x .git/hooks/commit-msg
```

We create our first change and push for review to `refs/for/master`:

```
$ echo "My first code review" > code-review-sample.txt
$ git add code-review-sample.txt
$ git commit -m "First Change for review"
```

[...]

```
create mode 100644 code-review-sample.txt
```

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

[...]

remote:

remote: New Changes:

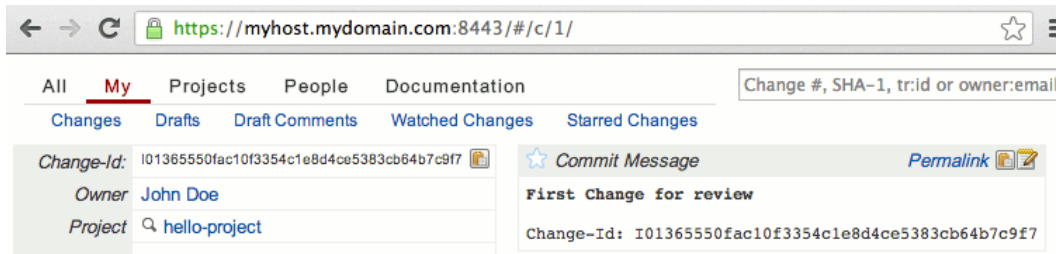
remote: https://myhost.mydomain.com:8443/1

remote:

To ssh://jdoe@myhost.mydomain.com:29418/hello-project

```
* [new branch]      HEAD -> refs/for/master
```

Our first change has been successfully created and can be displayed on the Gerrit web-UI using `https://myhost.mydomain.com:8443/1`, as shown in the following screenshot:



Setting topics

Organizing changes in topics allows you to selectively engage with the most appropriate audience of reviewers. A topic is a set of related changes that are meant to fulfill a global goal or are part of a global feature.

We can create a new change and assign a topic called `first-topic`, by leveraging the right part of the Gerrit reference after the `%` separator, we can add the topic parameter to indicate the target topic name.

```
$ git push origin HEAD:refs/for/master%topic=first-topic
[...]
```

remote: New Changes:

remote: https://myhost.mydomain.com:8443/2

remote:

To ssh://jdoe@myhost.mydomain.com:29418/hello-project

* [new branch] HEAD -> refs/for/master%topic=first-topic

Adding reviewers

There is a Gerrit reference for inviting reviewers; just add on the correct part of the reference after the `%` symbol, setting the `r` parameter to the reviewers e-mail:

```
$ git push origin \
    HEAD:refs/for/master%topic=first-topic,r=luca@milanesio.org
[...]
```

```

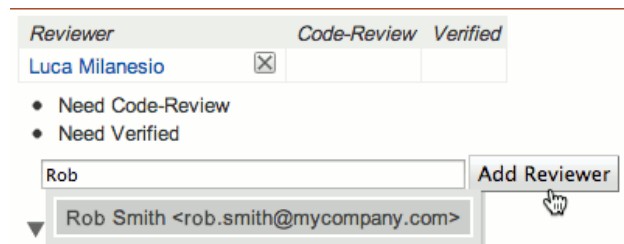
remote: Processing changes: new: 1, refs: 1, done
remote:
remote: New Changes:
remote:   https://myhost.mydomain.com:8443/5
remote:
To ssh://jdoe@myhost.mydomain.com:29418/hello-project
* [new branch]      HEAD -> refs/for/master%topic=first-topic,r=luca@
milanesio.org

```



Reviewers needs to be Gerrit registered users, that is, users who have logged in and already provided their e-mail address.

The change is now automatically assigned to the reviewer and they will receive an e-mail with the review URL. Alternatively, additional reviewers can be added using the web-UI and by typing the name or e-mail of another user with review permissions on the change screen. The following screenshot shows adding reviewers to a change:



Review labels

The Code Review activity was historically divided in the Android Open Source Project into two specific categories (or labels):


- **Review:** Dedicated to manual code inspection
- **Validate:** For functional verification through builds and tests

Today, review is the only predefined label in Gerrit, while additional arbitrary labels can be added at a later time. Refer to the documentation embedded in the Gerrit web-UI (https://myhost.mydomain.com:8443/Documentation/config-labels.html#label_custom) for detailed information on adding custom labels.

Review

The review feedback is associated to the quality of the code and is expressed by a numeric score ranging from -2 to + 2. The following list describes the meaning of each score value:

- **Review/-2 (Veto):** Typically set when a Committer of the Project has fundamental reasons for opposing the merge of this change. A Veto cannot be cancelled out by any other review, and unless it is removed it will block the change from being merged.
- **Review/-1:** Even though it is a negative score, it is not a block for merging the change. It indicates that more work is needed to fix the code and bring the change to an accepted state. Typically, the comments on the code will explain what needs to be amended for the change to be accepted. I would prefer that you didn't submit this.
- **Review/+1:** Positive score, no problems found on the code. However, the change cannot be submitted either because the Reviewer is not a Committer or because the Committer is not completely convinced of the change and requests somebody else's opinion on it. Looks good to me, but someone else must approve it.
- **Review/+2 (Approved):** Approval score, enables the submission of the change.

[ Review/0 is generally used for publishing comments without indicating any score.]

Commenting and scoring changes

Gerrit provides a very powerful web-UI for reviewing changes, allowing us to compare all differences side by side, and then directly inserting comments in line. We can access the side by side review pane by clicking on the list of files included in a change.

Gerrit side by side diff screen is very simple and effective. We can see the differences between files highlighted and then select what patch-sets we want to compare by clicking on the patch-set number (or base line) on either side of the diff screen.

Double-click on the points of the code to comment, enter the message text, and then click on **Save** to store the feedback. The following screenshot shows adding comments in side by side diff screen:



Review etiquette

There are golden rules of etiquette for inserting and replying to other's comments in the review board. It allows a more effective interaction between people and achieves a higher change submission rate. The etiquettes are as follows:

- **Review as discussion board:** Without Code Review, we would have used chats and e-mails to interact with people and discuss a code change. With Gerrit we need to keep all the discussion threads within the tool, tied to the code, and stored together with the change.
- **Review each file:** When reviewing somebody else's code, spend time doing a deep review of all changed files in order to try to understand and make the code yours. This allows maximum collective code ownership while doing an inspection.
- **It's all about the code:** Negative review has to be seen in a positive way to make things better. Comments are not meant to appoint personal blame to individuals. The objective of a review is to get better code and not assess people's skills.
- **Always answer all comments:** We need to answer every comment received, by using the **Reply** button. If the comment has been addressed, reply simply with "done" or explain why that review was ignored or was not important for us.
- **Use code in comments:** There is no better way to describe an idea than by proposing an alternative version of the same code. Gerrit understands the code fragment indenting and formats it accordingly in the web-UI rendering.

- **One change, one thing:** Reviews take a considerable amount of time. The main objective is getting a change merged. If a change becomes too big, it will require more time to get reviewed and will reduce the chance of getting it approved and merged.
- **Use topics:** Splitting big changes and grouping them into topics helps everyone to understand the general idea behind them, and at the same time getting each small change approved and merged has small benefits which accumulate.

There are additional keywords used in the Code Review jargon. The following list describes the most common terms used:

- **NIT on review message:** This indicates a minor aesthetic problem that requires a small change.
- **[Optional] on review message:** This indicates a personal preference or style and can be ignored if the author does not agree with the change.
- **[RFC] (Request For Comments) on commit message:** This indicates that the change is not necessarily intended to be merged, but has been pushed only to share an idea or an experiment in terms of change review, and trigger a discussion around it.
- **[WIP] (Work In Progress) on commit message:** This is often used for draft reviews that need to be public for getting early feedback. It does not require a proper full review, but just an overview and discussion around it.

Publish review and scoring

Comments are not visible until we click on the **Review** button on the patch-set. Draft comments will then be published and visible to all users. The following screenshot shows publishing review messages:

▼ Patch Set 3 acace11ad444334f3a798345d68f76e8ac4aa667

Author **John Doe** <john.doe@mycompany.com> Jun 2, 2013 8:37 PM

Committer **John Doe** <john.doe@mycompany.com> Jun 2, 2013 11:49 PM

Parent(s) 8b4f5117a656540fc8d79344128744865cf008fa Merge "First Change for review on topic with one reviewer"

Download [checkout](#) | [pull](#) | [cherry-pick](#) | [patch](#) | [Anonymous HTTP](#) | [SSH](#) | [HTTP](#) | `git fetch ssh://lmlanesio@myhost.mydomain.com:29418/hello-project refs/changes/02/2/3 && git`

[Review](#) [Abandon Change](#)

	File Path	Comments	Size	Diff
▶	Commit Message			Side-by-Side Unified
A	alpha-private-review.txt	1 draft	1 line	Side-by-Side Unified
			+1, -0	All Side-by-Side All Unified

The review screen allows to add a general comment and overall score to a change. Should the user be entitled to submit the change and scores enough for approval, publish and submit can be performed in one go by clicking on the **Publish and Submit** button.

Amending code under review

A review could result in a negative score that will need rework on the code. The change then needs to be amended with a new patch-set. The best approach is to restart with a fresh checkout of the change, so that you have a local environment that is not contaminated by other local edits.

To checkout the latest version of the change, we can use the browser and click on the clipboard icon next to the latest patch-set checkout command. The following screenshot shows checking out the latest change patch-set:


→ ↺ <https://myhost.mydomain.com:8443/#/c/2/>

Patch Set 1 80f4b3f035dafa43c3bb1c0a35a897680d873ef3

Author **John Doe** <john.doe@mycompany.com> Jul 21, 2013 9:39 AM

Committer **John Doe** <john.doe@mycompany.com> Jul 21, 2013 9:39 AM

Parent(s) 546fe49a82de5f467684634f7fb9172ada9718d2 First Change for review

Download [checkout](#) | [pull](#) | [cherry-pick](#) | [patch](#) | [Anonymous HTTP](#) | [SSH](#) | [HTTP](#) | `git fetch https://myhost.mydomain.com:8443/hello-project refs/changes/02/2/1 && git checkout FETCH_HEAD` 

We can then paste the command onto a local clone of the `hello-project` repository:

```
$ git fetch ssh://jdoe@myhost.mydomain.com:29418/hello-project refs/changes/02/2/1 && git checkout FETCH_HEAD
```

[...]

HEAD is now at acace11... Alpha private published Change

We can then edit the files to accommodate the feedback provided during the review. Remember to address all of the comments in your next patch-set, in order to minimize the number of review loops before the change approval. Additionally, it will allow the other reviewers to save time by checking everything again in one go:

```
$ echo "Change amended" >> alpha-private-review.txt
```

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

[...]

Final private published Change

Change-Id: I96afbedf8d3485b2aca1371ebafea2fb6aa35bec

[...]

[detached HEAD 8a42e67] Final private published Change

Author: John Doe <john.doe@mycompany.com>

1 file changed, 1 insertion(+)

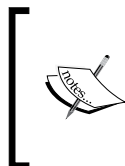
create mode 100644 alpha-private-review.txt

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

[...]

To ssh://jdoe@myhost.mydomain.com:29418/hello-project

* [new branch] HEAD -> refs/for/master



Gerrit associates a new commit to the same change using the change ID field. The most typical way to submit a new patch is by using the `--amend` option when reworking an existing commit; otherwise, a new change ID will be generated by `commit-hook` and Gerrit would then create a new change instead of adding a patch-set to the existing one.

The reworked change has now been uploaded and all reviewers have been notified to go and check it out. This time, hopefully, everything will be fine and the contributor will be able to win the change approval for being submitted and subsequently merged.

Summary

In this chapter we have completed a full cycle of Code Review, starting from a fresh change up to the multiple feedback loop and cycles through review and amendments.

We have learned how to interact with the workflow tools and with the people around it, understanding their jargon, and applying review etiquette to avoid conflicts and get more successful reviews.

Code Review takes time and cycles through the same commits again and again. We now know how to manage the reworking cycles and amend the code for addressing reviewers feedback. In the next chapter we will explore the strategies for merging changes into the main branch and working with multiple dependent changes at the end.

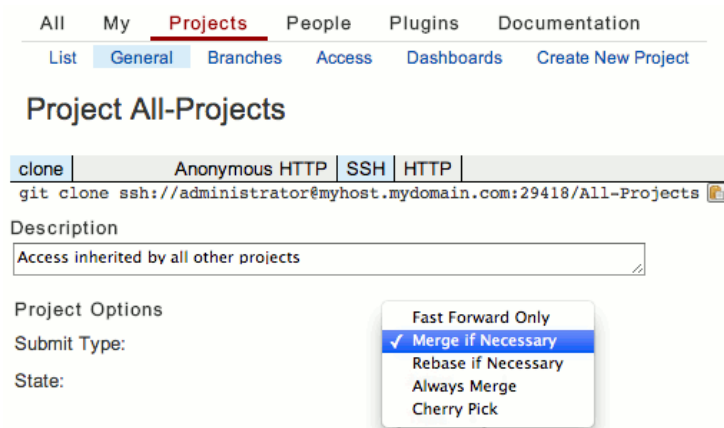
7

Submit Types and Concurrency

The most delicate moment is when concurrent and dependent changes need to be merged and executed in a specific order to prevent conflicts. In this chapter we will go through these problems and analyze the different situations in which they may occur, and how Gerrit helps you to manage them. Finally we will experiment with a set of concurrent changes under review and learn how to keep them up-to-date until their submission and merge.

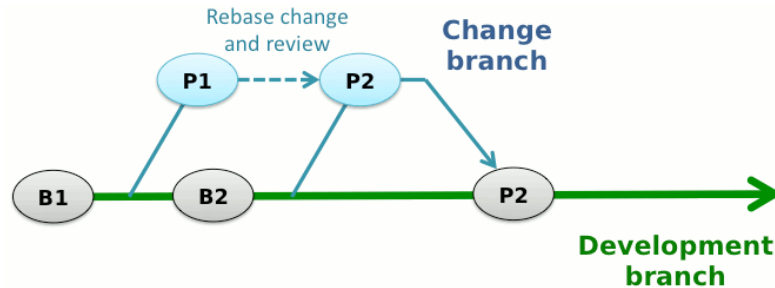
Submit types

While the process of scoring and submission in Gerrit is driven by the Code Review rules, as soon as the change is submitted to its target branch, it gets executed according to the project's submit type. The following screenshot shows project **Submit Type**: on the settings page:



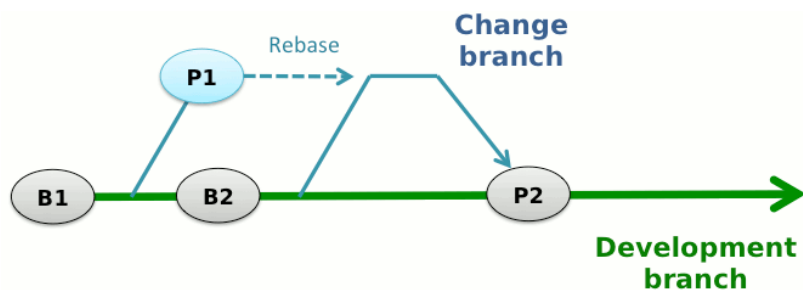
Fast forward only

The fast forward strategy accepts a change merged, only if there is a fast forward of the latest **head** of its target branch. In real terms, this forces the Contributor to continuously rebase his code as soon as other commits get pushed to the target branch; otherwise, any potential submission will not be able to merge it. The following diagram shows a fast forward submit:



Rebase if necessary

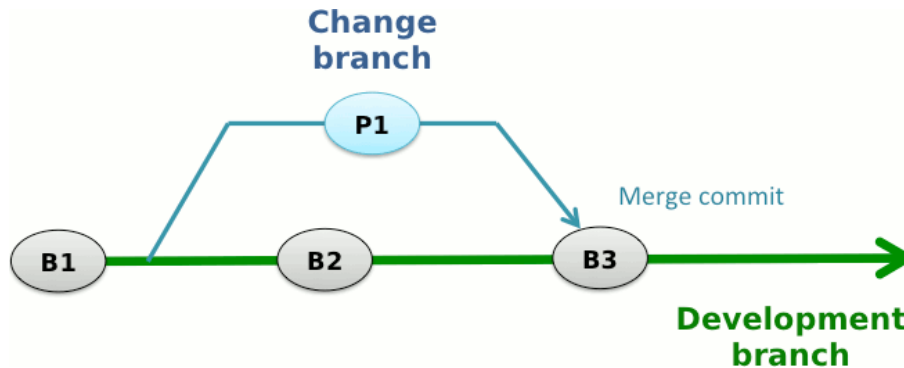
This is similar to the previous case but with one exception: whenever the change is not a direct superset of the latest commit on the target branch, Gerrit tries to rebase it. If no conflicts are generated, then the new rebased commit is merged into the target branch. The following diagram shows a rebase if necessary submit:



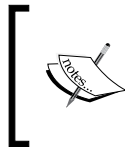
Even though this is technically similar to the previous scenario, in this situation the rebased commit (P2) was not validated through a Code Review cycle. In theory, it can have a different effect to the initial commit P1 that has passed the review.

Merge if necessary / always merge

Merge if necessary, uses the same principle of the default Git merge command: fast forward to the change if the head of the target branch was a direct predecessor. Always merge will generate a merge commit in all conditions, always leaving a track of the Code Review branch in the history of the destination branch. The following diagram shows the merge if necessary scheme:



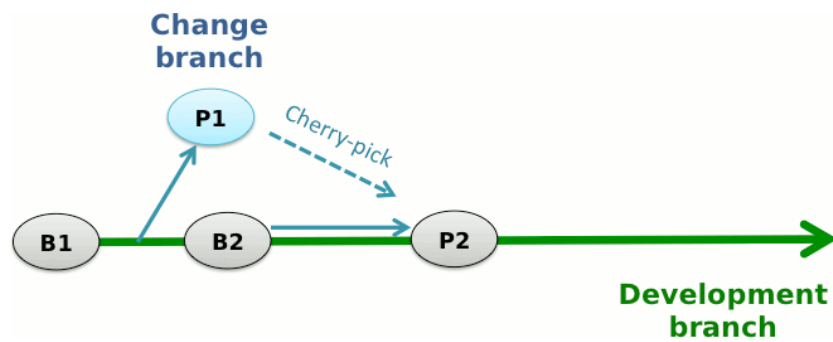
Even though this strategy seems less problematic because there are no implicit or explicit rebases needed, it may lead to unexpected results. Change in the branch context (P1) could have been legitimate and the review passed successfully, but once it is merged into its target branch it can potentially come together with other changes in different parts of the project, becoming then B3.



Because of the potential risk of having broken some other functionality during the unattended merge, we encourage the contributor to perform regular rebases of their changes, just to keep it fresh with the latest head version of the target branch.

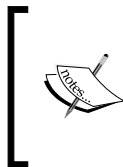
Cherry pick

The change that is actually merged is a completely new commit, built using the differences introduced in **P1** since its predecessor. If there are conflicts the cherry pick needs to be aborted. The following diagram shows the cherry pick submit type:



The new merged commit **P2** still keeps a reference in its commit message of the original change **P1** it came from, but the new cherry picked **P2** commit could require a full validation phase.

This strategy is similar to a rebase if necessary, but creates some concern in case of dependent changes: if **P1** depends on another change not yet merged, the dependency is completely ignored and **P1** gets cherry picked without its predecessor.



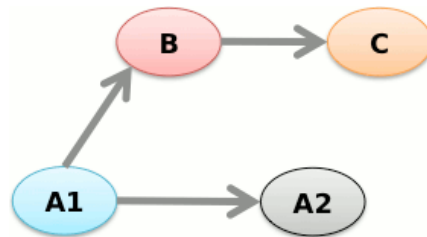
This strategy has to be used with care and could be suitable for projects under maintenance where the target branch is stable and commits don't change the overall structure of the code. It would not work at all in the situation of code refactoring, as it typically happens in an active development phase.

Concurrent code reviews

When teams start using Code Review, the number of open changes starts to increase and the chances of having more than one change running concurrently get higher. Because of the intrinsic latency introduced by the review workflow, it is likely that more changes will need to be based on other code that belongs to other non-approved changes.

Changes dependency tracking

Gerrit understands the branching structure of dependent changes and creates an acyclic dependency graph between them, where every change points to its successors and vice versa. The following diagram shows an example of Gerrit dependent changes:

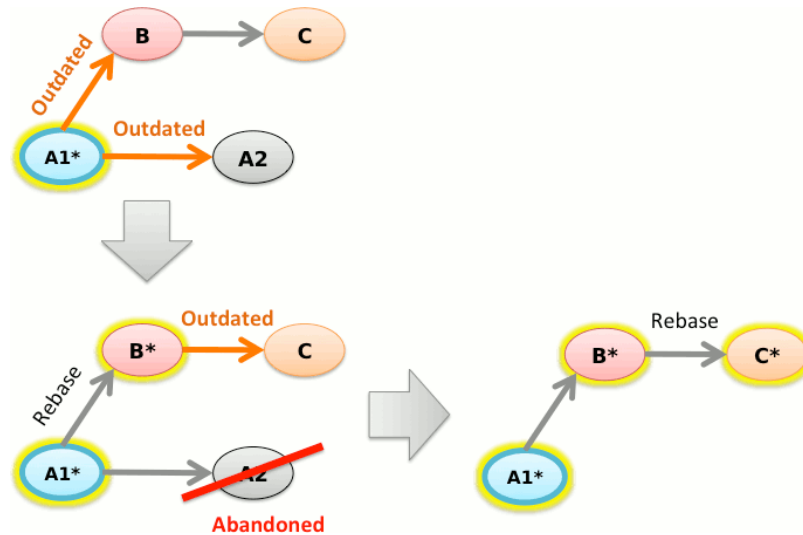


Gerrit also detects whenever a change in the graph has been amended with a new patch-set and passes this information down the chain to its direct children nodes. All the links between the amended node and its direct children are then marked as outdated and will need to be managed accordingly.

There are two ways to update an outdated link:

- **Rebase:** As its predecessor has moved, the change will also need to move accordingly with a code rebase. Should conflicts arise, the change will need to be manually reworked to resolve them.
- **Abandon:** Amendments on its predecessor may have invalidated the purpose of the change itself. If this is the case, it does not make any sense to continue as the code has become obsolete.

Once the link has been updated, either with a change rebase or reworking, its synchronization status is aligned with the new status of the dependency graph. The following diagram shows propagation and update of changes status:



Real-life change dependencies with Gerrit

Now let's experiment with the depicted scenario by creating concurrent changes under review.

First we create Change A1 related to topic A:

```
$ echo "Change A1" > change-a1.txt
$ git add change-a1.txt
$ git commit -a -m "Change A1"
[...]
$ git push origin HEAD:refs/for/master%topic=topic-A
[...]
remote: New Changes:
remote:   https://myhost.mydomain.com:8443/7
remote:
To ssh://jdoe@myhost.mydomain.com:29418/hello-project
* [new branch]      HEAD -> refs/for/master%topic=topic-A
```

Now we create Change A2 on top of Change A1, which is our head point:

```
$ echo "Change A2" >> change-a1.txt
$ git commit -a -m "Change A2"
[master 4564778] Change A2
 1 file changed, 1 insertion(+)

$ git push origin HEAD:refs/for/master%topic=topic-A
[...]
remote: New Changes:
remote:  https://myhost.mydomain.com:8443/8
remote:
To ssh://jdoe@myhost.mydomain.com:29418/hello-project
 * [new branch]      HEAD -> refs/for/master%topic=topic-A
```

Next we need to reset the current HEAD to Change A1 and create a new Change B on top:

```
$ git checkout HEAD^
[...]
HEAD is now at ada66d6... Change A1
$ echo "Change B" > change-b.txt
$ git add change-b.txt
$ git commit -m "Change B"
[...]
$ git push origin HEAD:refs/for/master
[...]
remote: New Changes:
remote:  https://myhost.mydomain.com:8443/9
remote:
To ssh://jdoe@myhost.mydomain.com:29418/hello-project
 * [new branch]      HEAD -> refs/for/master
```

Finally we create Change C on top of Change B, which is the current HEAD:

```
$ echo "Change C" > change-c.txt
$ git add change-c.txt
$ git commit -m "Change C"
[...]
$ git push origin HEAD:refs/for/master
[...]
remote: New Changes:
remote:  https://myhost.mydomain.com:8443/10
remote:
To ssh://jdoe@myhost.mydomain.com:29418/hello-project
 * [new branch]      HEAD -> refs/for/master
```

Navigating through the Gerrit change dependency graph

All four changes have now been pushed to Gerrit and we can see how they have been connected in their dependency graph.

By opening the Gerrit web-UI to **Change A1** (<https://myhost.mydomain.com:8443/7>), we can see the dependencies section of the change screen, which indicates the links with its dependent children. The following screenshot shows the change **Dependencies** panel:

Change-Id: I2cf61258d269359971d60449c49a6c74e1d1e9a0

Owner: John Doe

Project: hello-project

Branch: master

Topic: topic-A

Commit Message

Change A1

Change-Id: I2cf61258d269359971d60449c49a6c74e1d1e9a0

Dependencies

Subject	Owner	Project	Branch	Updated
Depends On				
(None)				
Needed By				
Change A2	John Doe	hello-project	master (topic-A)	4:04 PM
Change B	John Doe	hello-project	master	4:08 PM

Change A1 is linked to Change A2 and Change B as dependent nodes. All entries displayed in the dependency list can be used as hyperlinks to navigate through the graph very easily.

We can then click on **Change B** to navigate down through the dependency graph.

Managing change graph updates

Now we want to experiment with what happens when we push a new patch-set to Change A1. We expect to see an action by propagating the status update through the graph and flagging the outdated links.

We will need to make Change A1 active in our local workspace:

```
$ git checkout HEAD^^
[...]
HEAD is now at ada66d6... Change A1
```

Next we amend Change A1 and push it to Gerrit to generate a new patch-set:

```
$ echo "Change A1 amended" >> change-a1.txt
$ git commit -a --amend
[...]
Change A1 amended

Change-Id: I2cf61258d269359971d60449c49a6c74e1d1e9a0
[...]
[detached HEAD 55d9adc] Change A1 amended
1 file changed, 1 insertion(+)
create mode 100644 change-a1.txt

$ git push origin HEAD:refs/for/master%topic=topic-A
[...]
* [new branch]      HEAD -> refs/for/master%topic=topic-A
```

A new patch-set has been pushed to Change A1.

Change B is now flagged with an outdated dependency, because it is based on an older patch-set of Change A1. To re-synchronize the dependency, we can choose one of the buttons at the bottom of the Change B – patch-set 1: **Rebase Change** or **Abandon Change**.

If we click on **Rebase Change**, a new patch-set 2 is automatically created. Also the dependency pointing back to Change A1 is automatically refreshed and turns back to up-to-date.



Should the rebase result in conflicts, the operation will fail and the rebase would need to be performed manually and pushed as an additional patch-set.

The rebase of Change B has also created an outdated dependency with Change C. The operation must be repeated through the dependency chain until all of the dependencies are turned back to up-to-date.

Summary

In this chapter we have learned that Gerrit can control the way changes are going to be incorporated into their target branches. We have covered the advantages and risks of the different submit types in order to provide you with the basic elements to configure the most appropriate policy to different projects.

We have also seen how complex parallel Code Reviews can be in real life; thanks to Gerrit we have the ability to manage the topic branches paradigm effectively.

Finally, we have pushed Gerrit to the limits by using the changes dependency graph and tracking the alignment status across all the nodes. We have learned how to align the status of outdated links in the dependency chain by using Gerrit rebase functionality.

We are now fully empowered to use Gerrit Code Review in real-life projects with multiple projects and teams working together concurrently.

In Appendix A and B we will see how Gerrit can be integrated with Jenkins and used in conjunction with existing projects on GitHub.

A

Using Gerrit with GitHub

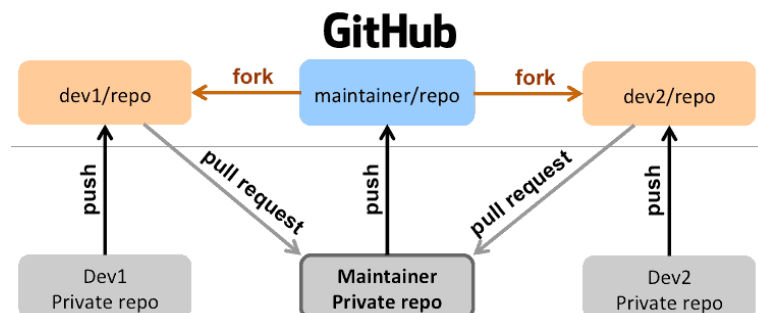
GitHub is the world's largest platform for the free hosting of Git Projects, with over 4.5 million registered developers. Most of the readers of this book probably already have a GitHub account and while reading the previous chapters you may be thinking about how to effectively use Gerrit with your existing GitHub repositories.

We will now provide a step-by-step example of how to connect Gerrit to an external GitHub server so as to share the same set of repositories. Additionally, we will provide guidance on how to use the Gerrit Code Review workflow and GitHub concurrently.

By the end of this Appendix we will have our Gerrit installation fully integrated and ready to be used for both open source public projects and private projects on GitHub.

GitHub workflow

GitHub has become the most popular website for open source projects, thanks to the migration of some major projects to Git (for example, Eclipse) and new projects adopting it, along with the introduction of the social aspect of software projects that piggybacks on the Facebook hype. The following diagram shows the GitHub collaboration model:



The key aspects of the GitHub workflow are as follows:

- Each developer pushes to their own repository and pulls from others
- Developers who want to make a change to another repository, create a fork on GitHub and work on their own clone
- When forked repositories are ready to be merged, pull requests are sent to the original repository maintainer
- The pull requests include all of the proposed changes and their associated discussion threads
- Whenever a pull request is accepted, the change is merged by the maintainer and pushed to their repository on GitHub

GitHub controversy

The preceding workflow works very effectively for most open source projects; however, when the projects gets bigger and more complex, the tools provided by GitHub are too unstructured, and a more defined review process with proper tools, additional security, and governance is needed.

In May 2012 *Linus Torvalds*, the inventor of Git version control, openly criticized GitHub as a commit editing tool directly on the pull request discussion thread: *"I consider GitHub useless for these kinds of things. It's fine for hosting, but the pull requests and the online commit editing, are just pure garbage"* and additionally, *"the way you can clone a (code repository), make changes on the web, and write total crap commit messages, without GitHub in any way making sure that the end result looks good."* See <https://github.com/torvalds/linux/pull/17#issuecomment-5654674>.

Gerrit provides the additional value that *Linus Torvalds* claimed was missing in the GitHub workflow: Gerrit and GitHub together allows the open source development community to reuse the extended hosting reach and social integration of GitHub with the power of governance of the Gerrit review engine.

GitHub authentication

The list of authentication backends supported by Gerrit does not include GitHub and it cannot be used out of the box, as it does not support OpenID authentication. However, a GitHub plugin for Gerrit has been recently released in order to fill the gaps and allow a seamless integration.

GitHub implements OAuth 2.0 for allowing external applications, such as Gerrit, to integrate using a three-step browser-based authentication. Using this scheme, a user can leverage their existing GitHub account without the need to provision and manage a separate one in Gerrit. Additionally, the Gerrit instance will be able to self-provision the SSH public keys needed for pushing changes for review.

In order for us to use GitHub OAuth authentication with Gerrit, we need to do the following:

- Build the Gerrit GitHub plugin
- Install the GitHub OAuth filter into the Gerrit libraries (`/lib` under the Gerrit site directory)
- Reconfigure Gerrit to use the HTTP authentication type

Building the GitHub plugin

The Gerrit GitHub plugin can be found under the Gerrit `plugins/github` repository on <https://gerrit-review.googlesource.com/#/admin/projects/plugins/github>. It is open source under the Apache 2.0 license and can be cloned and built using the Java 6 JDK and Maven.

Refer to the following example:

```
$ git clone https://gerrit.googlesource.com/plugins/github
$ cd github
$ mvn install
[...]
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 9.591s
[INFO] Finished at: Wed Jun 19 18:38:44 BST 2013
[INFO] Final Memory: 12M/145M
[INFO] -----
```

The Maven build should generate the following artifacts:

- `github-oauth/target/github-oauth*.jar`, the GitHub OAuth library for authenticating Gerrit users
- `github-plugin/target/github-plugin*.jar`, the Gerrit plugin for integrating with GitHub repositories and pull requests

Installing GitHub OAuth library

The GitHub OAuth JAR file needs to be copied to the Gerrit `/lib` directory; this is required to allow Gerrit to use it for filtering all HTTP requests and enforcing the GitHub three-step authentication process:

```
$ cp github-oauth/target/github-oauth-*.jar /opt/gerrit/lib/
```

Installing GitHub plugin

The GitHub plugin includes the additional support for the overall configuration, the advanced GitHub repositories replication, and the integration of pull requests into the Code Review process.

We now need to install the plugin before running the Gerrit init again so that we can benefit from the simplified automatic configuration steps:

```
$ cp github-plugin/target/github-plugin-*.jar \
/opt/gerrit/plugins/github.jar
```

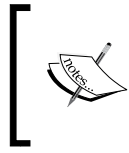
Register Gerrit as a GitHub OAuth application

Before going through the Gerrit init, we need to tell GitHub to trust Gerrit as a partner application. This is done through the generation of a `ClientId/ClientSecret` pair associated to the exact Gerrit URLs that will be used for initiating the 3-step OAuth authentication.

We can register a new application in GitHub through the URL `https://github.com/settings/applications/new`, where the following three fields are requested:

- **Application name:** It is the logical name of the application authorized to access GitHub, for example, Gerrit.
- **Main URL:** The Gerrit canonical web URL used for redirecting to GitHub OAuth authentication, for example, `https://myhost.mydomain:8443`.
- **Callback URL:** The URL that GitHub should redirect to when the OAuth authentication is successfully completed, for example, `https://myhost.mydomain:8443/oauth`.

GitHub will automatically generate a unique pair `ClientId/ClientSecret` that has to be provided to Gerrit identifying them as a trusted authentication partner.



ClientId/ClientSecret are not GitHub credentials and cannot be used by an interactive user to access any GitHub data or information. They are only used for authorizing the integration between a Gerrit instance and GitHub.

Running Gerrit init to configure GitHub OAuth

We now need to stop Gerrit and go through the init steps again in order to reconfigure the Gerrit authentication. We need to enable HTTP authentication by choosing an HTTP header to be used to verify the user's credentials, and to go through the GitHub settings wizard to configure the OAuth authentication.

```
$ /opt/gerrit/bin/gerrit.sh stop
```

```
Stopping Gerrit Code Review: OK
```

```
$ cd /opt/gerrit
```

```
$ java -jar gerrit.war init
```

```
[...]
```

```
*** User Authentication
```

```
***
```

```
Authentication method      []: HTTP RETURN
```

```
Get username from custom HTTP header [Y/n]? Y RETURN
```

```
Username HTTP header       []: GITHUB_USER RETURN
```

```
SSO logout URL              : /oauth/reset RETURN
```

```
*** GitHub Integration
```

```
***
```

```
GitHub URL [https://github.com]: RETURN
Use GitHub for Gerrit login ? [Y/n]? Y RETURN
ClientId [] : 384cbe2e8d98192f9799 RETURN
ClientSecret [] : f82c3f9b3802666f2adcc4 RETURN
```

```
Initialized /opt/gerrit
```

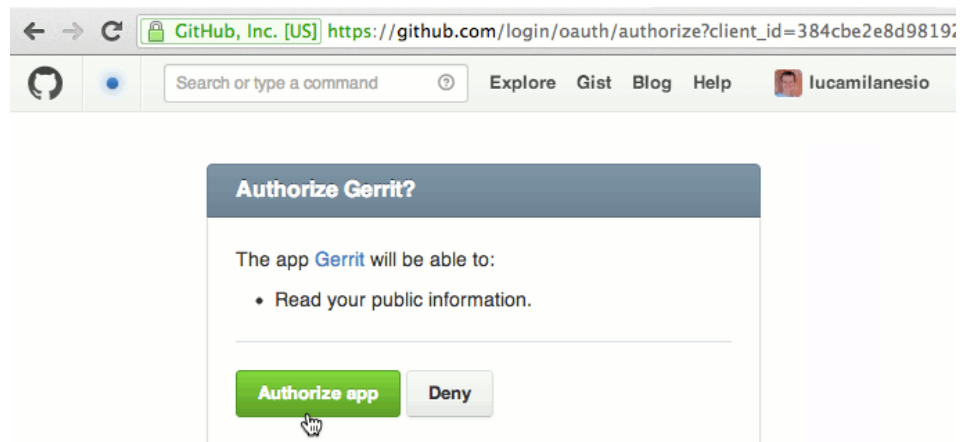
```
$ /opt/gerrit/bin/gerrit.sh start
```

```
Starting Gerrit Code Review: OK
```

Using GitHub login for Gerrit

Gerrit is now fully configured to register and authenticate users through GitHub OAuth. When opening the browser to access any Gerrit web pages, we are automatically redirected to the GitHub for login. If we have already visited and authenticated with GitHub previously, the browser cookie will be automatically recognized and used for the authentication, instead of presenting the GitHub login page. Alternatively, if we do not yet have a GitHub account, we create a new GitHub profile by clicking on the **SignUp** button.

Once the authentication process is successfully completed, GitHub requests the user's authorization to grant access to their public profile information. The following screenshot shows GitHub OAuth authorization for Gerrit:



The authorization status is then stored under the user's GitHub applications preferences on <https://github.com/settings/applications>.

Finally, GitHub redirects back to Gerrit propagating the user's profile securely using a one-time code which is used to retrieve the full data profile including username, full name, e-mail, and associated SSH public keys.

Replication to GitHub

The next steps in the Gerrit to GitHub integration is to share the same Git repositories and then keep them up-to-date; this can easily be achieved by using the Gerrit replication plugin.

The standard Gerrit replication is a master-slave, where Gerrit always plays the role of the master node and pushes to remote slaves. We will refer to this scheme as **push replication** because the actual control of the action is given to Gerrit through a `git push` operation of new commits and branches.

Configure Gerrit replication plugin

In order to configure push replication we need to enable the Gerrit replication plugin through Gerrit init:

```
$ /opt/gerrit/bin/gerrit.sh stop
```

```
Stopping Gerrit Code Review: OK
```

```
$ cd /opt/gerrit
```

```
$ java -jar gerrit.war init
```

```
[...]
```

```
*** Plugins
```

```
***
```

```
Prompt to install core plugins [y/N]? y RETURN
```

```
Install plugin reviewnotes version 2.7-rc4 [y/N]? RETURN
```

```
Install plugin commit-message-length-validator version 2.7-rc4 [y/N]?  
RETURN
```



```
Install plugin replication version 2.6-rc3 [y/N]? y RETURN
```

```
Initialized /opt/gerrit
```

```
$ /opt/gerrit/bin/gerrit.sh start
```

```
Starting Gerrit Code Review: OK
```

The Gerrit replication plugin relies on the `replication.config` file under the `/opt/gerrit/etc` directory to identify the list of target Git repositories to push to. The configuration syntax is a standard `.ini` format where each group section represents a target replica slave.

See the following simplest `replication.config` script for replicating to GitHub:

```
[remote "github"]  
  
    url = git@github.com:myorganisation/${name}.git
```

The preceding configuration enables all of the repositories in Gerrit to be replicated to GitHub under the `myorganisation` GitHub Team account.

Authorizing Gerrit to push to GitHub

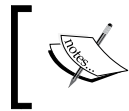
Now, that Gerrit knows where to push, we need GitHub to authorize the write operations to its repositories. To do so, we need to upload the SSH public key of the underlying OS user where Gerrit is running to one of the accounts in the GitHub `myorganisation` team, with the permissions to push to any of the GitHub repositories.

Assuming that Gerrit runs under the OS user `gerrit`, we can copy and paste the SSH public key values from the `~gerrit/.ssh/id_rsa.pub` (or `~gerrit/.ssh/id_dsa.pub`) to the **Add an SSH Key** section of the GitHub account under target URL to be set to: <https://github.com/settings/ssh>

Start working with Gerrit replication

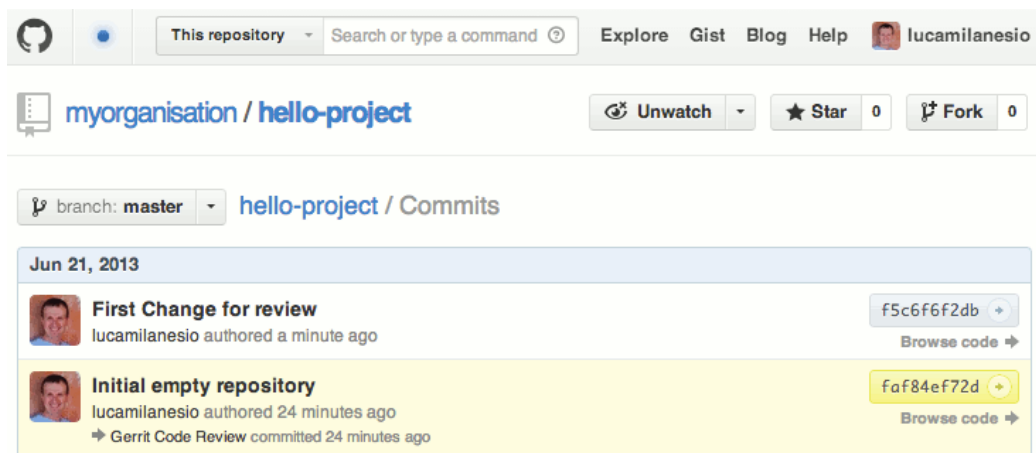
Everything is now ready to start playing with Gerrit to GitHub replication. Whenever a change to a repository is made on Gerrit, it will be automatically replicated to the corresponding GitHub repository.

In reality there is one additional operation that is needed on the GitHub side: the actual creation of the empty repositories using `https://github.com/new` associated to the ones created in Gerrit. We need to make sure that we select the organization name and repository name, consistent with the ones defined in Gerrit and in the `replication.config` file.



Never initialize the repository from GitHub with an empty commit or readme file; otherwise the first replication attempt from Gerrit will result in a conflict and will then fail.

Now GitHub and Gerrit are fully connected and whenever a repository in GitHub matches one of the repositories in Gerrit, it will be linked and synchronized with the latest set of commits pushed in Gerrit. Thanks to the Gerrit-GitHub authentication previously configured, Gerrit and GitHub share the same set of users and the commits authors will be automatically recognized and formatted by GitHub. The following screenshot shows Gerrit commits replicated to GitHub:



Reviewing and merging to GitHub branches

The final goal of the Code Review process is to agree and merge changes to their branches. The merging strategies need to be aligned with real-life scenarios that may arise when using Gerrit and GitHub concurrently.

During the Code Review process the alignment between Gerrit and GitHub was at the change level, not influenced by the evolution of their target branches. Gerrit changes and GitHub pull requests are isolated branches managed by their review lifecycle.

When a change is merged, it needs to align with the latest status of its target branch using a fast-forward, merge, rebase, or cherry-pick strategy. Using the standard Gerrit merge functionality, we can apply the configured project merge strategy to the current status of the target branch on Gerrit. The situation on GitHub may have changed as well, so even if the Gerrit merge has succeeded there is no guarantee that the actual subsequent synchronization to GitHub will do the same!

The GitHub plugin mitigates this risk by implementing a two-phase submit + merge operation for merging opened changes as follows:

- **Phase-1:** The change target branch is checked against its remote peer on GitHub and fast forwarded if needed. If two branches diverge, the submit + merge is aborted and manual merge intervention is requested.
- **Phase-2:** The change is merged on its target branch in Gerrit and an additional ad hoc replication is triggered. If the merge succeeds then the GitHub pull request is marked as completed.

At the end of Phase-2 the Gerrit and GitHub statuses will be completely aligned. The pull request author will then receive the notification that his/her commit has been merged.

Using Gerrit and GitHub on <http://gerrithub.io>

When using Gerrit and GitHub on the web with public or private repositories, all of the commits are replicated from Gerrit to GitHub, and each one of them has a complete copy of the data. If we are using a Git and collaboration server on GitHub over the Internet, why can't we do the same for its Gerrit counterpart? Can we avoid installing a standalone instance of Gerrit just for the purpose of going through a formal Code Review?

One hassle-free solution is to use the GerritHub service (<http://gerrithub.io>), which offers a free Gerrit instance on the cloud already configured and connected with GitHub through the `github-plugin` and `github-oauth` authentication library. All of the flows that we have covered in this Appendix are completely automated, including the replication and automatic pull request to change automation. As accounts are shared with GitHub, we do not need to register or create another account to use GerritHub; we can just visit <http://gerrithub.io> and start using Gerrit Code Review with our existing GitHub projects without having to teach our existing community about a new tool.

GerritHub also includes an initial setup Wizard for the configuration and automation of the Gerrit projects and the option to configure the Gerrit groups using the existing GitHub. Once Gerrit is configured, the Code Review and GitHub can be used seamlessly for achieving maximum control and social reach within your developer community.

Summary

We have now integrated our Gerrit installation with GitHub authentication for a seamless Single-Sign-On experience. Using an existing GitHub account we started using Gerrit replication to automatically mirror all the commits to GitHub repositories, allowing our projects to have an extended reach to external users, free to fork our repositories, and to contribute changes as pull requests.

Finally, we have completed our Code Review in Gerrit and managed the merge to GitHub with a two-phase change submit + merge process to ensure that the target branches on both Gerrit and GitHub have been merged and aligned accordingly. Similarly to GitHub, this Gerrit setup can be leveraged for free on the web without having to manage a separate private instance, thanks to the free set target URL to <http://gerrithub.io> service available on the cloud.

In Appendix B we will see how Gerrit can be integrated with Jenkins for providing automatic validation and feedback on the Code Review.

B

Automation with Jenkins

This chapter takes us through the steps needed to configure Jenkins for fetching code directly from Gerrit changes during the review phase. Using Gerrit in parallel with Continuous Integration enables the automation of a series of validations against the code, including the verification of coding style guidelines, build and execution of unit tests, all before the code is actually merged into its target branch.

Integration is implemented through the Gerrit trigger plugin, which allows Jenkins to listen to Gerrit stream events and consequently trigger actions. We will go through the configuration steps to get the plugin downloaded, installed, and fully configured to work seamlessly with Gerrit.

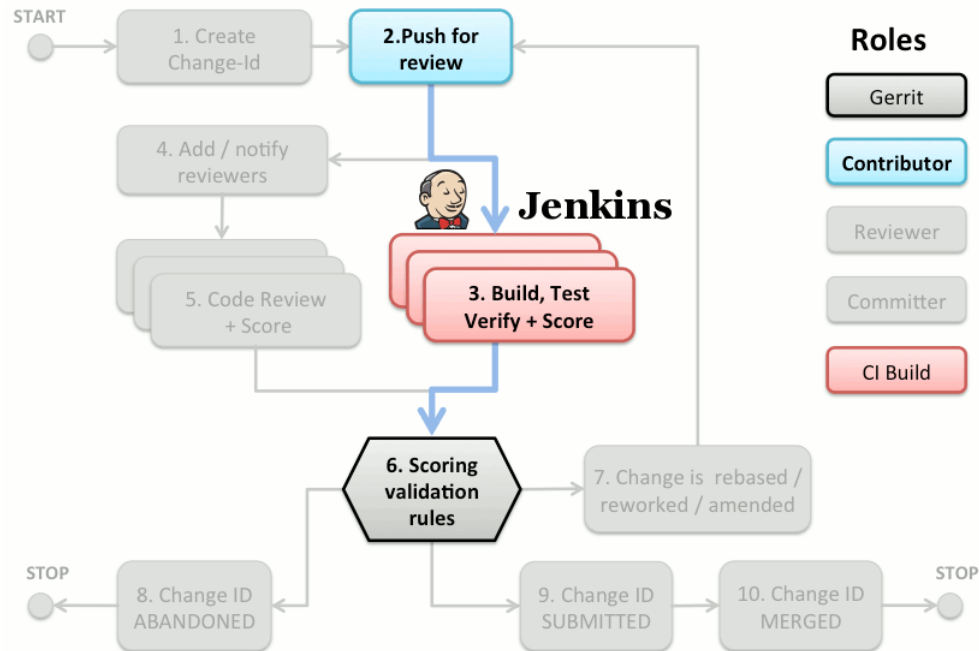
Gerrit review workflow with Jenkins

Code integration and validation plays a central role in Code Review, as it is in one of the key steps of the workflow.

As soon as a change is uploaded to Gerrit for review, it can be automatically fetched by Jenkins by going through the following steps:

1. Code is fetched from the Gerrit Code Review and checked out to the latest patch-set committed on the change.
2. Code is compiled, tested, and then packaged. This step ensures that code is complete, syntactically and semantically correct, and does not break any existing functionality.
3. Based on the results of the previous steps, a positive (+1) or negative (-1) validated score is added to the corresponding Gerrit change.

The following diagram shows Jenkins role in the Gerrit workflow:



Setting up the Jenkins Gerrit plugin

In order to integrate Jenkins with Gerrit, we need to install a series of plugins through the Jenkins plugin manager. The URLs and screenshots included in this book are referring to Jenkins version 1.519 on the host `myhost.mydomain.com`, running on port 8080, while Gerrit will still be running on the same host on port 8443 (HTTPS).

Installing Jenkins plugins

The following plugins are needed on Jenkins:

- **Jenkins Git Client plugin:** Provides support for the Git protocol using the JGit library. This is optional.
- **Jenkins Git plugin:** Provides the integration of Git as version control for fetching the code.
- **Jenkins Gerrit Trigger plugin:** Allows you to fetch the patch-sets from Gerrit changes, trigger a build, and, based on the results, submit a score with comments back to Gerrit.

Setting up the Gerrit trigger plugin

The Gerrit trigger plugin is very flexible and powerful, but rather complex to configure and get working correctly. It connects to Gerrit using two different protocol settings:

- **HTTP/S Canonical URL:** Used for building changes and patch-set URLs in Jenkins to point to Gerrit.
- **SSH Access:** Used for connecting to Gerrit and listening to the stream events over an SSH connection.

Getting Gerrit ready for Jenkins

First of all, we need to define a batch user that will be used by Jenkins to connect to Gerrit and access code along with the associated information on its repository. Do not define surrogate users on the LDAP (Lightweight Directory Access Protocol) or external authentication system. For batch usage of Jenkins automation, it is strongly recommended that you use a Gerrit internal user with limited authorization access to the system intended to be used.

Assuming that our Jenkins instance is run by the user `jenkins` with a SSH key pair under `/home/jenkins/.ssh` without a pass phrase protection (otherwise Jenkins cannot automatically use the key), we can use the Gerrit `create-account` command from an existing Gerrit admin user:

```
$ cat /home/jenkins/.ssh/id_rsa.pub | \
ssh -p 29418 admin@myhost.mydomain.com gerrit create-account \
--group "'Non-Interactive Users'" --full-name Jenkins \
--email jenkins@myhost.mydomain.com \ --ssh-key - jenkins
```

Next we need to make sure that this newly created account is granted the necessary permissions on Gerrit All-Projects on the Non-Interactive Users group:

- `Read access to refs/*` means permission to read and then clone any change from Gerrit repositories
- `Label Verified/-1..+1 to refs/heads/*` means permission to label any change as verify from -1 up to +1 score
- `Stream events` means permission to listen to Gerrit stream events remotely

Gerrit trigger plugin configuration

The Jenkins Gerrit trigger needs to have the Gerrit canonical URL and SSH connectivity details specified on its custom configuration page available at `http://myhost.mydomain.com:8080/gerrit-trigger`. Click on the **Test Connection** button to validate SSH connectivity and log in to Gerrit. The configuration can then be saved, but will not be active until the Gerrit trigger daemon inside Jenkins is restarted. Next click on the **Start/Stop** buttons at the bottom of the page or restart Jenkins and look for the following message in the log:

```
com.sonyericsson.hudson.plugins.gerrit.gerritevents.GerritHandler  
runINFO: Ready to receive data from Gerrit
```

Once the daemon has been successfully restarted, you can double-check the end-to-end functionality by opening the Gerrit trigger page on `http://myhost.mydomain.com:8080/gerrit_manual_trigger`, entering the string `status:open` in the **Query** field, and clicking on the **Search** button. You should see the same list of open changes shown in Gerrit on the page `https://myhost.mydomain.com:8443/#/q/status:open,n,z`.

Triggering a build from Gerrit

We need to define a Jenkins job to fetch the code from Gerrit and build it in the Continuous Integration environment. Additionally, we may want to disable all of the post validation phases except unit tests, in order to shorten the build execution time. Remember that this job will be triggered much more frequently than a typical Continuous Integration job.

Configuring the Gerrit trigger

We need to change the conditions for triggering a new build on the job from **SCM polling** (or other trigger policy) to **Gerrit event**. Additionally, we also need to specify the Gerrit conditions for the trigger action, available when clicking on the **Advanced** button in the Gerrit trigger specific configuration section.

The additional parameters needed are project name (specified as a plain complete text string) and branches (specified as the path regular expression `**`).

Configuring the Git plugin

Finally, we need to tell the Git plugin how to get the source code from the Gerrit trigger plugin. This inter-plugin communication happens through environment variables: Gerrit defines two variables (`$GERRIT_REFSPEC` and `$GERRIT_PATCHSET_REVISION`) populated respectively with the Gerrit ref-spec to fetch and the actual Change patch-set to be checked out.

We need then to edit the Git repository settings (by clicking on the **Advanced** button) in the following way:

- Set `$GERRIT_REFSPEC` as the Git refspec to clone
- Set `$GERRIT_PATCHSET_REVISION` as the Git branch to build
- Set the choosing strategy to **Gerrit trigger**

We suggest that you enable two additional flags:

- **Wipe out workspace:** Potentially each build could fetch a completely different branch of the code, without wiping out the workspace; we may risk relying on temporary files generated from previous builds.
- **Use shallow clone:** Changes are typically well-defined unique snapshots of code; having the entire clone history on the local file system would not help and just increases the amount of disk space used.

These flags are not necessarily needed, but are strongly recommended for preventing future headaches when fetching and building multiple changes.

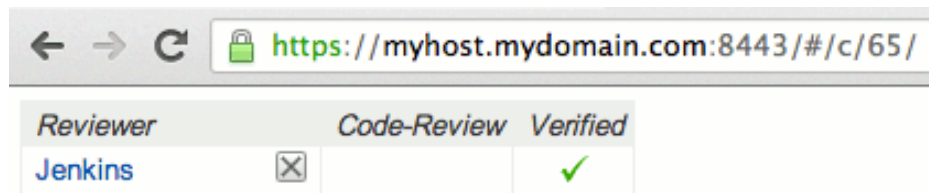
Automated code validation

The Jenkins and Gerrit setup is complete and now we can see how it plugs into the Gerrit review workflow. When the change is pushed to Gerrit for review, a new build gets automatically triggered on the Jenkins job that was previously configured.

The Jenkins job page shows the result of pushing the new change. It is similar to a normal Continuous Integration execution whereby an additional small **G** icon with two numbers indicating the change number and patch-set number represents the Gerrit trigger activation and points to the original change/patch-set built.

Additionally, the Gerrit trigger plugin adds comments to the Gerrit change in order to report any activity to the change author and reviewers. This is precious information for the other reviewers that are working on the assessment of the patch-set. A change that has been successfully validated by a build and unit test execution should be more stable and then can take a higher priority for being assessed by Gerrit reviewers.

If the build is successful, the Gerrit trigger plugin submits positive feedback on the Gerrit change with a Verified/+1 score. The following screenshot shows the Gerrit trigger plugin submit a Verified + 1:



Summary

With the integration between Jenkins and Gerrit now complete, we have come to the end of our journey for setting up and learning how to effectively use the Code Review workflow.

Jenkins has provided the additional automation that validates the code sitting in the review inbound queue.

The automatic validation saves team members' time, which would otherwise be spent fixing broken branches. This means more effort can then be spent reading somebody else's code and increasing quality, project, and code knowledge across the team.

By validating changes through Code Review, the product backlog gets naturally prioritized. Adding more flexibility to the product roadmap, the most stable features will get completed first with the remaining features following a longer and much-needed stabilization process before getting released.

C

Git Basics

This appendix provides the reader with a basic knowledge of the concepts and common terms of the Git version-control system. We will introduce the minimum set of commands needed for working with the central and local Git repository. A more complete description of the Git commands can be found via other sources; however, we will provide a useful guide that allows beginners to quickly get set up and use Git in conjunction with Gerrit Code Review.

Peer-to-peer distributed version-control system

Git is a distributed version control which means that there is no difference between a client and a server; they are seen simply as two different peer nodes that contain the full history of all archived files. There is not necessarily a Git server or a Git client, but simply Git repositories that are able to synchronize and exchange files between each other.

We can even use Git to archive and version the files without the need of a server at all. The editing of this book was archived and versioned using a Git repository for tracking its progress and changes. Once downloaded, Git is ready to be used immediately so it is the epitome of Plug and Play!

Git installation

Git can be downloaded from <http://git-scm.com>, and is made up of a collection of console-based commands. It is available as an installable package for all major platforms, including Windows and Mac OS, but on Linux it is embedded in the distribution packages and can be installed using the standard `yum`, `rpm`, or `apt-get` system tools.

On a Red Hat or CentoOS Linux distribution, we can install Git from the `root` user using:

```
$ yum install git
```

On Ubuntu or other Debian-based distribution, we can install Git using:

```
$ apt-get install git
```

To obtain the list of the most commonly used commands, you can run `git help` and for help on a specific command, just add the command name as an additional parameter:

```
$ git help
```

```
usage: git [--version] [--exec-path[=<path>]] [--html-path]
        [--man-path] [--info-path] [-p|--paginate|--no-pager]
        [--no-replace-objects] [--bare] [--git-dir=<path>]
        [--work-tree=<path>] [--namespace=<name>]
        [-c name=value] [--help]
        <command> [<args>]
```

[...]

```
$ git help config
```

NAME

git-config - Get and set repository or global options

SYNOPSIS

```
git config [<file-option>] [type] [-z|--null] name
        [value [value_regex]]
```

[...]

Git needs to be aware of the identity of the user who is actually performing the commands. Git cannot rely on the username logged in to the system, as it needs to allow portability of its files across multiple sites and it needs a global site-independent identity based on the real name (first + last name) and the associated e-mail address (`user@somedomain`).

The commands that need to be performed once per username are (assuming the John Doe user with `john.doe@mydomain.com` as e-mail address):

```
$ git config --global user.name "John Doe"
```

```
$ git config --global user.email "john.doe@mydomain.com"
```



Git does not verify the identity, as its focus is mainly on versioning files. However, Gerrit Code Review, which sits on top of the Git protocol, enforces identity authentication and validation on the repositories sitting on the server side using a proper external user registry such as LDAP or OpenID.

Creating the first hello world repository

In order to create a Git repository, we need to use the following command (assuming the repository name `hello-repo`):

```
$ git init hello-repo
```

Initialized empty Git repository in `/home/user/hello-repo/.git/`

The command has created two separate directories:

- The working directory (for example, `hello-repo`) contains all of the files that need to be tracked and versioned with Git
- The Git repository (for example, `hello-repo/.git`) contains all of the Git internal files and configuration settings for tracking and storing the different versions of the files in the working directory, and their associated version meta-data

Hello world file archived in Git

We can now create our first `hello.txt` file under the working directory, and in order to add it to the Git repository, we need to first use the `git add` and `git commit` commands:

```
$ cd hello-repo
```

```
$ echo "hello world" > hello.txt
```

```
$ git add hello.txt
```

```
$ git commit -m "My first file archived in Git"
```

```
1 file changed, 1 insertion(+)
```

```
create mode 100644 hello.txt
```

Displaying repository history

We can verify that our file has been effectively archived in Git, by using the `git log` command, which shows the full log of changes applied to the entire repository:

```
$ git log
commit 4febf3cf27c3a1599023dafb4abd5ee6f1cbb6c
Author: John Doe <john.doe@mydomain.com>
Date:   Thu Jul 18 18:51:07 2013 +0100
    My first file archived in Git
```



Git tracks groups of content changes and not individual files. This enables there to be a consistent audit log of the directory, without having to query all of the individual files under version control.

Editing and archiving a new file change

If we modify the file `hello.txt` again, Git detects the changes and displays the status of each file in the `hello-repo` directory using the `git status` command:

```
$ echo "hello git world" > hello.txt
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
directory)
#
#   modified:   hello.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

In order to archive a new version of `hello.txt`, we need to use the `git add` and `git commit` commands again:

```
$ git add hello.txt
$ git commit -m "Second change on hello.txt"
[master f08cfe1] Second change on hello.txt
1 file changed, 1 insertion(+), 1 deletion(-)
```



Git does not make distinctions between a file being added or edited. Git detects and tracks content changes: each change first has to be added and then committed to Git. The `git commit -a` command is a shortcut to add and commit for modified files.

Git basic concepts

The commands we have used so far are based on the following concepts:

- **Staging area or Index:** The command `git add` causes the file involved to be put into an intermediate bucket of changes called the **Git staging area**. This area contains all of the changes (addition, deletion, and modification) that are ready to be stored in the Git repository as an atomic version.
- **Commit or commit object:** The command `git commit -m "..."` creates an atomic unit of work (similar to a database transaction) which contains all of the changes in the staging area. Once the commit is created, the staging area is emptied and a commit object is created in the Git repository, containing a full snapshot of the files at that specific moment in time.
- **Commit ID or object ID:** The `git log` command shows an alphanumeric string next to the commit (for example, `4febfb3cf27c3a1599023dafb4abd5ee6f1cbb6c`) which represents the unique ID of the commit in the Git history. Commit IDs are 160 bit (40 alphanumeric chars) SHA-1 hashing codes, but can be abbreviated with their initial characters (for example, `4febfb3`) as long as they are not ambiguous within the same Git repository.
- **Author and Committer:** This is the Git user (configured through the `git config user.name` and `git config user.email`) that initially created a commit into the Git repository. Even if the commit is amended at later time by another person, the original author will remain unchanged while the additional contributor will be tracked as Committer.

Working with Git branches

Every commit is linked to its predecessor through its parent commit ID. The named head points to a series of linked commits in a path is called a **branch**. The default branch in Git is called the **master**. When a branch is labeled, the commits can be referenced using their branch name instead of the commit ID. The command `git branch branch-name branch-start` assigns a new branch label to an existing starting point (commit ID or branch name). In the following example, we will create the branch `oldmaster` that points to our first commit:

```
$ git branch oldmaster 4febfb3
```


The branch label is automatically updated with the latest commit added to it.

The command `git branch` without arguments displays the list of named branches, the current one prefixed by `*`:

```
$ git branch
* master
  oldmaster
```

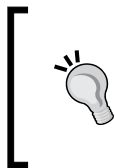
The command `git checkout branch` (commit ID or branch name) extracts from the Git repository all of the files associated to the commit pointed to `branch`:

```
$ git checkout oldmaster
Switched to branch 'oldmaster'
$ cat hello.txt
hello world
```

```
$ git checkout master
Switched to branch 'master'
$ cat hello.txt
hello git world
```

It can be very useful to display the current branch point on the working directory. When using bash, it is possible to set up a custom prompt which includes the current branch as a prefix under brackets:

```
$ source \
  /usr/local/git/contrib/completion/git-completion.bash
$ export PS1='${__git_ps1 " (%s)"} \ $ '
(master) $ git checkout oldmaster
(oldmaster) $
```



Git branches can be created from any commit in the history; this creates an arbitrary set of branches that connects all of the Git commits, which is called a Git graph. This can often be very complex, so it can be graphically displayed using the `gitk` GUI utility included in the Git distribution.

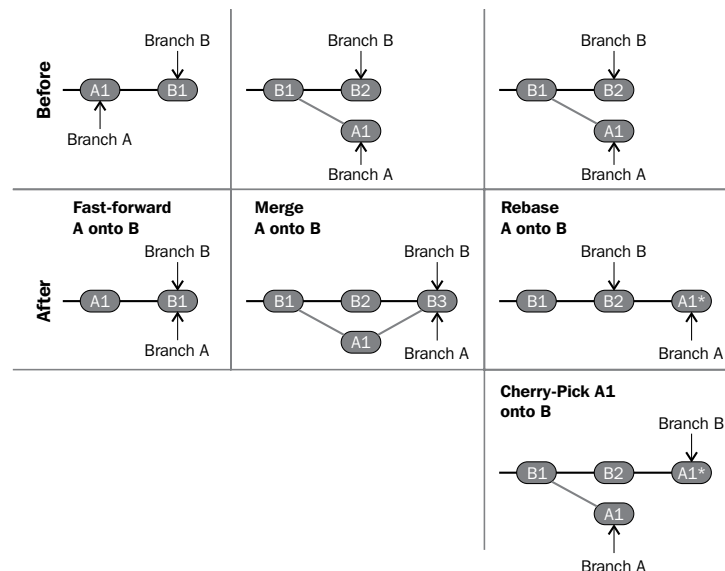


Fast forward, merge, rebase, and cherry pick

Different branches can be reunited into a single line of history using different mechanisms as follows:

- **Fast forward** is when branch A and branch B are on the same line of history. When branch A is fast forwarded to branch B, A is updated with the value of B.
- **Merge** creates a new commit containing the merged content of the files contained in the two branches. When branch A is merged onto branch B, a new merged commit gets created on top of branch B, containing all of the content changes provided by branch A.
- **Rebase** of branch A onto branch B takes all of the commits from branch A that are not yet contained in branch B and re-applies them on top of the latest commit of B. The original history of branch A is not available anymore, as it has been overwritten.
- **Cherry pick** applies the differences introduced by one commit on top of another branch. It is similar to the rebase scenario, but arbitrary commits of the original branch can be applied one by one. The original branch A history is preserved.

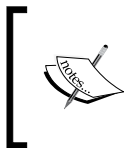
Please refer to the Git documentation for the syntax of the `git merge`, `git rebase`, and `git cherry-pick` commands. The following diagram shows the differences between all of the mechanisms described previously and also the Git merging schemes:



Working with remote Git repositories

Git allows different repositories to be connected together using the concept of **remotes**. The following example shows how to add a remote called `origin`, pointing to a repository hosted on GitHub.com:

```
(master) $ git remote add origin \
    https://github.com/gerritforge/hello-repo.git
(master) $ git remote -v
origin  https://github.com/gerritforge/hello-repo.git (fetch)
origin  https://github.com/gerritforge/hello-repo.git (push)
```



In order to proceed with the samples on remotes, you need to create your own account on <http://github.com> and also create your own `hello-repo` with an initial commit. You then need to replace `gerritforge` with your GitHub user name in all the following examples.

The most common way to work with remotes is through the `git clone` command, which automatically performs the following operations:

1. Creates a full clone of the remote Git repository into the local directory.
2. Adds a remote called `origin` pointing to the remote original repository.
3. Performs a checkout of the default branch (typically `master`) on the working directory.

The following example creates a new clone of `hello-repo` in the local directory `hello-repo-cloned` and displays its remotes' pointers and all local and remote tracked branches:

```
(master) $ cd ..

$ git clone \
    https://github.com/gerritforge/hello-repo.git \
    hello-repo-cloned

$ cd hello-repo-cloned
(master) $ git remote -v
origin  https://github.com/gerritforge/hello-repo.git (fetch)
origin  https://github.com/gerritforge/hello-repo.git (push)
```

```
$ git branch -a
* master
  oldmaster
  remotes/origin/master
  remotes/origin/oldmaster
```

Cloned repositories are fully featured local Git repositories, ready to be used for adding commits, branching, or merging as described previously. There are also additional functionalities provided by the remote repositories:

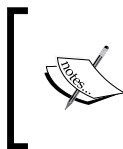
- A local branch is tracked against the original remote branch from where it was cloned
- All remote branches are available locally under the `origin` namespace
- `git fetch` automatically retrieves and updates the reference to all remote branches into the local Git repository
- `git push` sends the commits on the tracked branches to the remote repository

Pushing branches to the remote repository

The following example creates an extra local commit on the master branch and pushes it to the remote repository:

```
(master) $ echo "Hi remote repo" >> hello.txt
(master) $ git commit -a -m "Sample for push"
[master 34f4325] Sample for push
1 file changed, 1 insertion(+)

(master) $ git push
Username for 'https://github.com': gerritforge
Password for 'https://gerritforge@github.com': *****
To https://github.com/gerritforge/hello-repo.git
2162df7..34f4325 master -> master
```



Git does not mandate any user authentication or other security mechanism when communicating to remote repositories; however, in this case the authentication is implemented by GitHub by verifying username and password over the HTTP/S protocol.

Fetching from remote repositories

The fetch operation receives all updates from the branches of the remote repositories. In the following example, we fetch the remote branches from GitHub in our original local repository `hello-repo`:

```
$ cd ../hello-repo
```

```
$ git fetch
```

```
remote: Counting objects: 9, done.
```

```
remote: Compressing objects: 100% (3/3), done.
```

```
remote: Total 9 (delta 0), reused 9 (delta 0)
```

```
Unpacking objects: 100% (9/9), done.
```

```
From https://github.com/gerritforge/hello-repo
```

```
* [new branch]      master    -> origin/master
```

The fetch does not have any effect on the target local branch `master` because the commits have been stored in the locally tracked remote branches.

In order to get fetched commits into the local branches, we need to perform a merge operation with one of the strategies supported by Git (merge, rebase, or cherry pick). The recommended strategy when pulling changes from a remote repository is the rebase strategy:

```
$ git rebase origin/master
```

```
First, rewinding head to replay your work on top of it...
```

```
$ git log
```

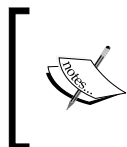
```
commit 34f432552db777d59204332c70a169a50cd1c9a2
```

```
Author: John Doe <john.doe@mydomain.com>
```

```
Date:   Fri Jul 19 08:49:53 2013 +0100
```

```
Sample for push
```

```
[...]
```



Should this operation identify any conflict, it would need to be resolved by manually editing the files and once modified, adding them again and completing the rebase with `git rebase --continue`.

Fetch and merge can be performed in one go using the `git pull` or `git pull --rebase` commands, allowing respectively to apply automatically a merge or rebase of the commits retrieved from the remote repository.

Summary

In this appendix we have introduced the basics of Git version control and gained confidence in using the most common commands for archiving versioned files locally and on a remote repository.

We have explored only a small subset of the commands needed to be able to start using Git in conjunction with Gerrit Code Review. A deeper understanding of the commands and concepts is strongly recommended, and can be gained by reading the free e-book provided on the Git website at <http://git-scm.com/book>.

Index

A

- abandon change action** 18
- Access Control Lists (ACLs)** 15, 59
- actions** 60
- Active Directory**
 - used, for authentication 44, 45
- administrators** 65
- All-Projects permissions**
 - about 61
 - global capabilities 61
 - reference permissions 61
- always merge submit type** 87, 119
- anonymous user** 65
- Apache 2.4**
 - using, as HTTP frontend 46
- Apache HTTP frontend** 46
- apt-get system tool** 113
- automated code validation** 111

B

- basic concepts, Git**
 - author and commiter 117
 - commit ID or object ID 117
 - commit or commit object 117
 - staging area or index 117
- benefits, Code Review**
 - about 7
 - build stability 8, 9
 - external early feedback 10
 - knowledge sharing 9, 10
 - qualitative code selection 12
 - shared code style 11
 - team engagement 11, 12

- BLOCK directive** 60
- Bouncy Castle Security**
 - installing 50, 51
- branches**
 - about 117
 - pushing, to remote repository 121
- branch namespace**
 - for Code Review 74, 75
- build**
 - triggering, from Gerrit 110

C

- change graph updates**
 - managing 93, 94
- cherry pick submit type** 88, 119
- code**
 - amending, under review 81, 82
- Code Review**
 - about 71
 - benefits 7
 - branch namespace 74, 75
 - versus Pair Programming 10
- Code Review jargon**
 - keywords 80
- Code Review permissions** 63, 64
- Code Review roles**
 - about 13
 - committer 14
 - contributor 13
 - maintainer 14
 - reviewer 13
- Code Review workflow**
 - roles 72
- committers** 14

concurrent code reviews

- about 89
- change graph updates, managing 93, 94
- dependent changes, tracking 89
- Gerrit change dependency graph 92, 93
- real life change dependencies,
with Gerrit 90-92

configuration, Gerrit replication plugin 101

configuration, Gerrit Trigger 110

configuration, Gerrit Trigger plugin 110

configuration, Git plugin 111

configuration, projects permissions 60

configuration steps, LDAP 41-44

Continuous Integration

- with Code Review 9
- without Code Review 8

contributor 13

curl command 22

D

dependent changes

- tracking 89

E

elements, Gerrit project

- Access Control Lists (ACLs) 15
- additional metadata 15
- changes references under review 15
- Git repository 15
- prolog rules 15

external groups 66

F

fast forward submit type 86, 119

fetch operation 122

G

Gerrit

- authorizing, for push operations 102
- build, triggering from 110
- changes, commenting 78
- changes, scoring 78

downloading 22

getting ready, for Jenkins 109

GitHub login, using for 100

home screen 27

initial setup, running 22-25

installation, completing 26

OpenID authentication 36-39

pre-requisites 21

project, cloning 30, 31

project, creating 30

project submit type 85

registering, as GitHub OAuth application 98

security, enabling on 49, 50

setting up, behind reverse proxy 46

SSH, using with 51

user profile, creating 26-28

using, on <http://gerrithub.io> 104

Gerrit batch authentication

about 34

work flow 34

Gerrit change

abandoning 18

about 16

merging 18

submitting 17

Gerrit change dependency graph

navigating through 92, 93

Gerrit configuration

HTTPS, enabling on 55

Gerrit DB 24

Gerrit/Git HTTP Server 25

Gerrit HTTP authentication

enabling 47

Gerrit init

running, for GitHub OAuth

configuration 99

Gerrit interactive authentication

about 34

versus Git authentication 34

work flow 34

Gerrit Java Container 25

Gerrit permission scheme 59, 60

Gerrit project

elements 15

Gerrit reference 62

- Gerrit replication**
 - code, approving 103
 - code, merging to GitHub branches 103
 - working with 102
- Gerrit replication plugin**
 - configuring 101
- Gerrit review workflow**
 - with Jenkins 107
- Gerrit site init 25**
- Gerrit Trigger**
 - configuring 110
- Gerrit Trigger plugin**
 - configuring 110
 - setting up 109
- Gerrit user profile**
 - SSH keys, adding to 52
- Git**
 - about 51, 113
 - hello world file, archiving 115
 - HTTP password, generating for 28, 29
 - installing 113, 114
 - URL, for downloading 113
- git add command 115**
- Git authentication**
 - about 34
 - versus Gerrit interactive authentication 34
 - work flow 35
- Git branches**
 - working with 117, 118
- git clone command 30, 120**
- git commit command 115**
- git fetch command 121**
- GitHub**
 - about 95
 - controversy 96
 - replication 101
 - using, on <http://gerrithub.io> 104
- GitHub authentication**
 - about 96
 - Gerrit, registering as GitHub OAuth application 98
 - GitHub login, using for Gerrit 100
 - GitHub OAuth library, installing 98
 - GitHub plugin, building 97
 - GitHub plugin, installing 98
- GitHub collaboration model 96**

- GitHub login**
 - using, for Gerrit 100
- GitHub OAuth application**
 - Gerrit, registering as 98
- GitHub OAuth library**
 - installing 98
- GitHub plugin**
 - building 97
 - installing 98
- GitHub workflow**
 - about 95
 - key aspects 96
- git log command 116**
- Git-over-SSH Server 25**
- Git permissions 62**
- Git plugin**
 - configuring 111
- git push command 121**
- Git reference 62**
- Git repository 15**
- Git/SSH**
 - repo, cloning over 53, 54
- Git/SSH Client keys 51**
- group hierarchy**
 - using, effectively 67
- groups**
 - external groups 66
 - internal groups 65
 - managing 64, 65

H

- hello world file**
 - archiving, in Git 115
- hello world repository**
 - creating 115
- HTTP frontend**
 - Apache 2.4, using as 46
- HTTP_LDAP authentication method 47**
- HTTP password**
 - generating, for Git 28, 29
- HTTPS**
 - about 49
 - enabling, on Gerrit configuration 55
- HTTP/S Canonical URL 109**
- HTTP/S reverse-proxy, Gerrit 55, 56**
- HTTP/S Support, Gerrit 57, 58**

I

- inheritance** 60
- in-house private Gerrit authentication**
 - using 41
- installation, Bouncy Castle Security** 50, 51
- installation, Gerrit**
 - completing 26
- installation, Git**
 - about 113, 114
 - file change, archiving 116
 - hello world file, archiving 115
 - hello world repository, creating 115
 - repository history, displaying 116
- installation, GitHub OAuth library** 98
- installation, GitHub plugin** 98
- installation, Jenkins plugins** 108
- integration** 107
- internal accounts**
 - about 35
 - need for 35
- internal groups** 65

J

- Java Cryptography Extensions** 50
- Java Virtual Machine (JVM)** 49
- Jenkins**
 - Gerrit review workflow 107
- Jenkins Gerrit plugin**
 - setting up 108
- Jenkins Gerrit Trigger plugin** 108
- Jenkins Git Client plugin** 108
- Jenkins Git plugin** 108
- Jenkins plugins**
 - installing 108
- Jenkins role**
 - in Gerrit workflow 108

K

- Kanban** 12
- knowledge sharing** 9

L

- Label Code Review** 17
- labeling** 15

LDAP

- about 109, 115
- configuration steps 41-44

M

- maintainer** 14
- master** 117
- merge change action** 18
- merge if necessary submit type** 87, 119
- methods, outdated link updation**
 - abandon 89
 - rebase 89
- MySQL** 24

N

- non-interactive users** 65

O

- OpenID** 115
- OpenID authentication, Gerrit** 36-39
- outdated link**
 - updating, ways 89

P

- Pair Programming**
 - about 9
 - versus Code Review 10
- passphrase** 52
- peer-to-peer distributed version control system** 113
- permission composition**
 - action 59, 60
 - resource 59
 - subject 59
- permissions-only projects** 68
- PostgreSQL** 24
- project owners** 65
- project security templates**
 - organizing 68, 69
 - permission-only projects 68
 - regular projects 68
- projects permissions**
 - All-Projects permissions 61, 62
 - Code Review permissions 63, 64

- configuring 60
- Gerrit reference 62
- Git permissions 62
- Git reference 62
- push replication 101**

Q

- qualitative code selection 12**

R

- RAG (red-amber-green) status 8**
- Read-only LDAP user profile 44**
- real life change dependencies**
 - with Gerrit 90, 92
- rebase if necessary submit type 86, 119**
- registered users 65**
- remote Git repositories**
 - working with 120, 121
- remote repositories**
 - branches, pushing to 121
 - fetching from 122
- remotes 120**
- replication.config file 103**
- repo**
 - cloning, over Git/SSH 53, 54
- repository history**
 - displaying 116
- Request For Comments (RFC) 13**
- resources 60**
- Review board**
 - versus Sprint Burndown chart 12
- reviewers**
 - about 13
 - adding 76
- Review etiquette 79**
- review feedback 78**
- Review labels 77**
- review messages**
 - publishing 80
- review workflow**
 - about 71
 - step-by-step 72
- reworking 74**
- rpm tool 113**

S

- score value, review feedback**
 - Review/-1 78
 - Review/+1 78
 - Review/+2 (Approved) 78
 - Review/-2 (Veto) 78
- security**
 - enabling, on Gerrit 49, 50
- shallow clone 111**
- shared code style 11**
- Single-Sign-On. *See* SSO**
- singleusergroup plugin**
 - about 60
 - URL 67
- Sprint Burndown chart**
 - versus Review board 12
- SSH**
 - about 49, 51
 - using, with Gerrit 51
- SSH Access 109**
- SSH keys**
 - adding, to Gerrit user profile 52
- SSO 39, 40**
- subject 60**
- submit change action 17**
- submit types**
 - about 85
 - always merge 87, 119
 - cherry pick 88, 119
 - fast forward only 86, 119
 - merge if necessary 87, 119
 - rebase if necessary 86, 119

T

- third-party authentication options**
 - about 45
 - Apache HTTP frontend 46
 - Gerrit HTTP authentication, enabling 47
 - Gerrit, setting up behind reverse proxy 46
 - user profile lookup 47
- topics**
 - setting 76

U

user authentication

working 33

user profile lookup 47

user registry 24

W

workflow

creating, steps 73, 74

Y

yum tool 113



Thank you for buying Getting started with Gerrit

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

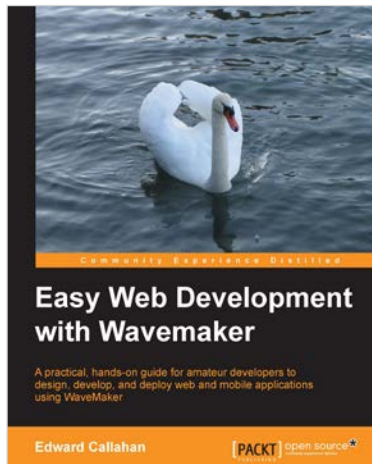
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



Easy Web Development with WaveMaker

ISBN: 978-1-782161-78-3

Paperback: 306 pages

A practical, hands-on guide for amateur developers to design, develop, and deploy web and mobile applications using WaveMaker

1. Develop and deploy custom, data-driven, and rich AJAX web and mobile applications with minimal coding using the drag-and-drop WaveMaker Studio
2. Use the graphical WaveMaker Studio IDE to quickly assemble web applications and learn to understand the project's artefacts
3. Customize the generated application and enhance it further with custom services and classes using Java and JavaScript



Software Development on the SAP HANA Platform

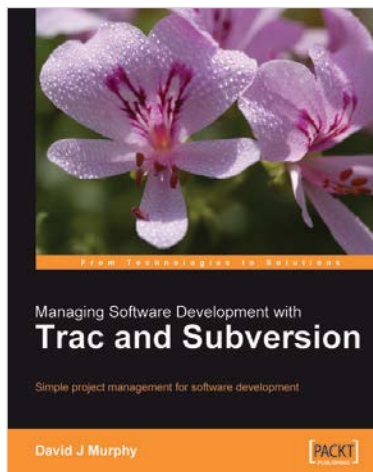
ISBN: 978-1-849689-40-3

Paperback: 328 pages

Unlock the true potential of the SAP HANA platform

1. Learn SAP HANA from an expert
2. Go from installation and setup to running your own processes in a matter of hours
3. Cover all the advanced implementations of SAP HANA to help you truly become a HANA master

Please check www.PacktPub.com for information on our titles

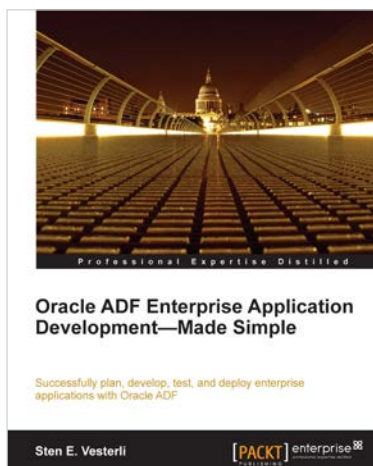


Managing Software Development with Trac and Subversion

ISBN: 9781-847191-66-3 Paperback: 120 pages

Simple project management for software development

1. Managing software development projects simply
2. Configuring a project management server
3. Installing, configuring, and using Trac
4. Installing and using Subversion



Oracle ADF Enterprise Application Development—Made Simple

ISBN: 978-1-849681-88-9 Paperback: 396 pages

Successfully plan, develop, test, and deploy enterprise applications with Oracle ADF

1. Best practices for real-life enterprise application development
2. Proven project methodology to ensure success with your ADF project from an Oracle ACE Director
3. Understand the effort involved in building an ADF application from scratch, or converting an existing application

Please check www.PacktPub.com for information on our titles