

Git Hooks

|  |  |
| --- | --- |
| Document name | Git Hooks |
| Version no. | 1.0 |
| Release date |  |

|  |
| --- |
|  |

# Table of Contents

[Table of Contents 2](#_Toc440445525)

[1. Git Hooks 3](#_Toc440445526)

[1.1 Execution of Hook 3](#_Toc440445527)

[1.2 Types of Hooks 3](#_Toc440445528)

[1.3 Pre-requiste for Hooks 3](#_Toc440445529)

[1.4 Sample Hook File 4](#_Toc440445530)

[2. Client Side Hooks 5](#_Toc440445531)

[2.1 Commit Message HOOK 5](#_Toc440445532)

[2.2 Pre-Commit HOOK 7](#_Toc440445533)

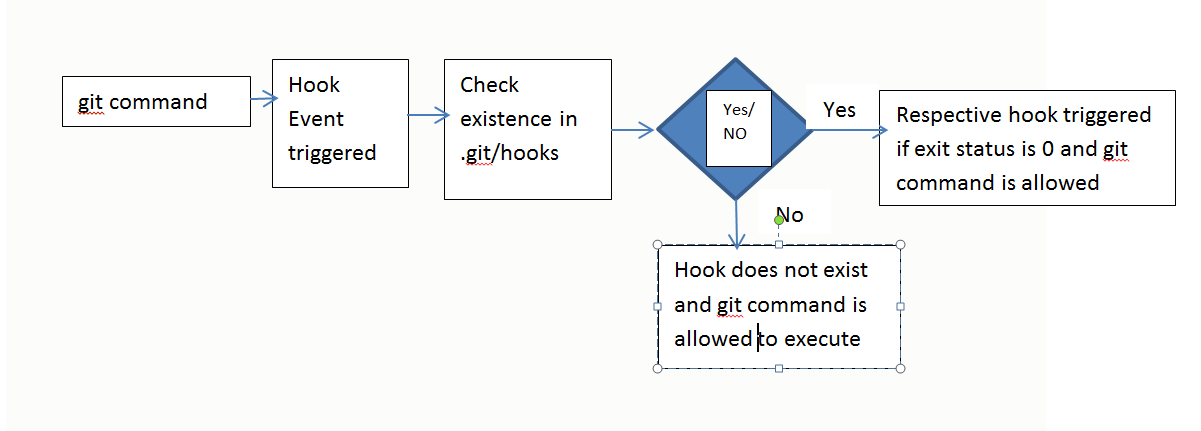
[3. Server Side Hooks 8](#_Toc440445534)

[3.1 Creating Hooks for a Specific Project 9](#_Toc440445535)

# Git Hooks

Like many other Version Control Systems, Git has a way to fire off custom scripts when certain important actions occur. There are two groups of these hooks: client-side and server-side. Client-side hooks are triggered by operations such as committing and merging, while server-side hooks run on network operations such as receiving pushed commits. You can use these hooks for all sorts of reasons.

## Execution of Hook



1. Execution of Git Hook

## Types of Hooks

1. Client side hook
2. Server side hook

## Pre-requiste for Hooks

For a script file to be a valid hook, there are 4 pre-requisites:-

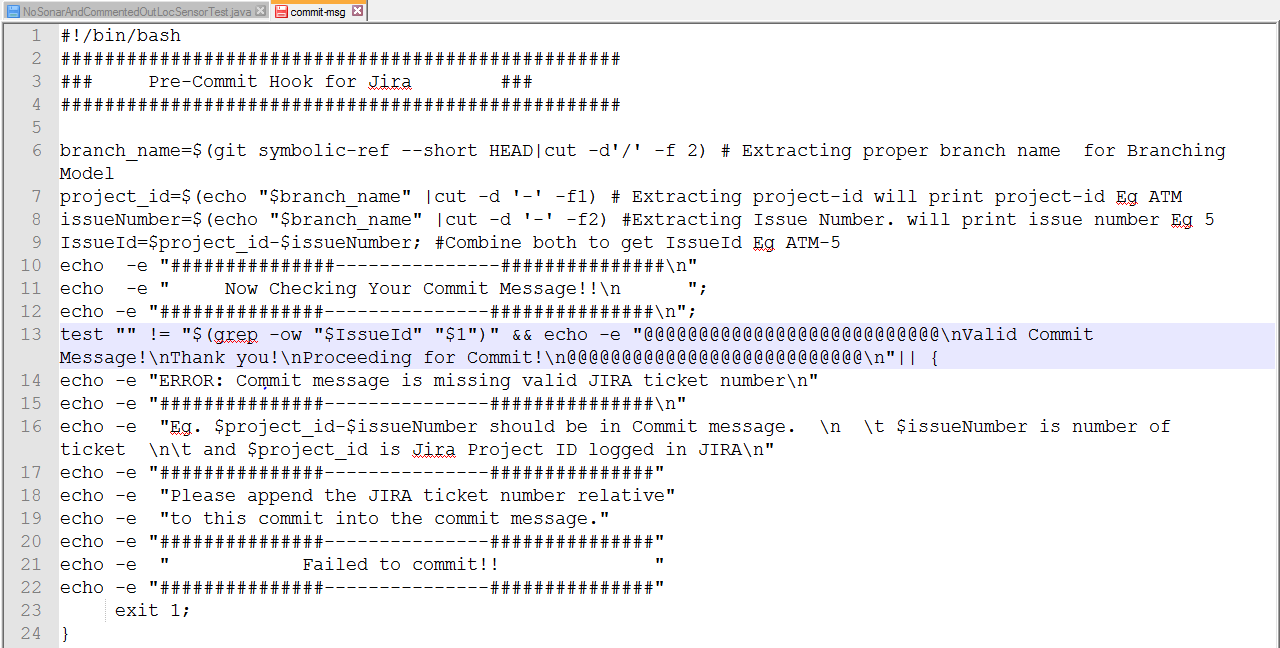
1. The file should be a script. So, it should be written in some scripting language like bash, sh, js, python, etc.

2. It must have executable permissions for all users.

3. The script must start with shebang line indicating the scripting language used. (E.g #!/bin/sh for sh script)

4. Script must refer to some refs.

## Sample Hook File



1. Sample File for Hook

# Client Side Hooks

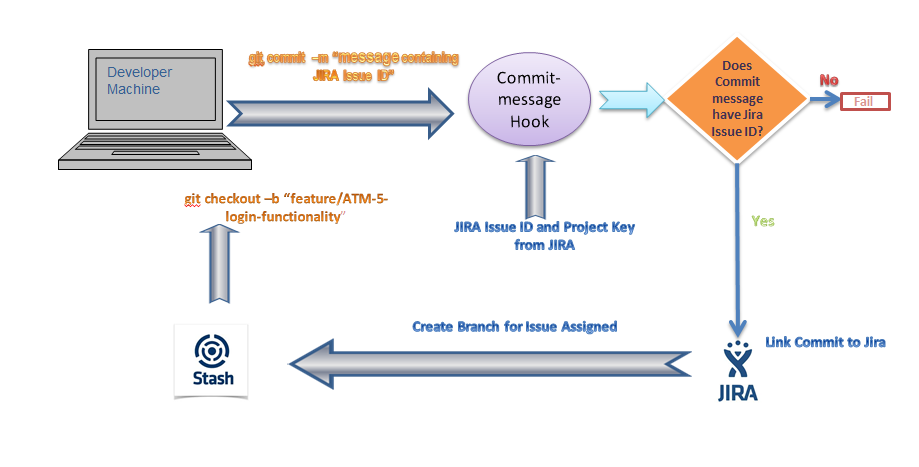
Followings are the client side hooks:

* Apply-patch,
* pre-applypatch
* pre-commit
* post-commit
* commit-msg
* pre-rebase
* etc.

## Commit Message HOOk

It got triggered after git commit “<message>” is fired. The purpose of this hook is to check whether Commit message contains proper JIRA Issue ID or not.

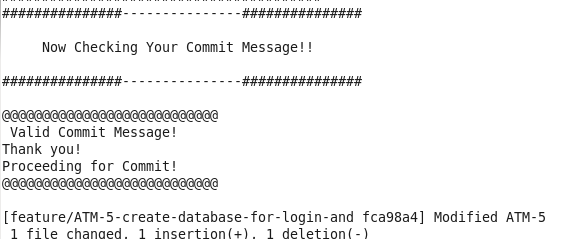
Workflow is explained as shown in following diagram:



1. Workflow of Commit-message hook



1. Execution of Commit message hook (Fail)

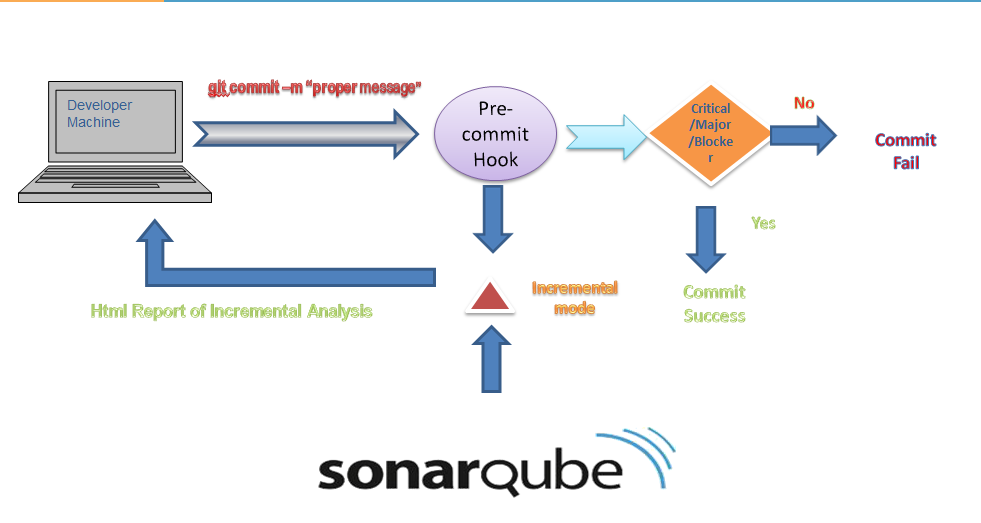


1. Execution of Commit message Hook (success)

## Pre-Commit HOOK

Pre-commit hook is triggered before git commit operation is performed. If script returns 0, then commit operation is allowed and commit object is created, but if script’s exit status is non zero, then commit operation is aborted

Following Diagram is for Pre-commit hook configured for sonar analysis:



1. Workflow of pre-commit hook

# Server Side Hooks

Server-side hooks work just like local ones, except they reside in server-side repositories (e.g., a central repository, or a developer’s public repository). When attached to the official repository, some of these can serve as a way to enforce policy by rejecting certain commits.

There are 3 server-side hooks that we’ll be discussing in the rest of this article:

* pre-receive
* update
* post-receive

All of these hooks let you react to different stages of the git push process.

The output from server-side hooks are piped to the client’s console, so it’s very easy to send messages back to the developer. But, you should also keep in mind that these scripts don’t return control of the terminal until they finish executing, so you should be careful about performing long-running operations.

#### Pre-Recieve

The first script to run when handling a push from a client is pre-receive. It takes a list of references that are being pushed from stdin; if it exits non-zero, none of them are accepted. You can use this hook to do things like make sure none of the updated references are non-fast-forwards, or to do access control for all the refs and files they’re modifying with the push.

#### Update

The update script is very similar to the pre-receive script, except that it’s run once for each branch the pusher is trying to update. If the pusher is trying to push to multiple branches, pre-receive runs only once, whereas update runs once per branch they’re pushing to. Instead of reading from stdin, this script takes three arguments: the name of the reference (branch), the SHA-1 that reference pointed to before the push and the SHA-1 the user is trying to push. If the update script exits non-zero, only that reference is rejected; other references can still be updated.

#### Post-Receive

The post-receive hook runs after the entire process is completed and can be used to update other services or notify users. It takes the same stdin data as the pre-receive hook. Examples include e-mailing a list, notifying a continuous integration server, or updating a ticket-tracking system – you can even parse the commit messages to see if any tickets need to be opened, modified, or closed. This script can’t stop the push process, but the client doesn’t disconnect until it has completed, so be careful if you try to do anything that may take a long time.

#### How to Write Server Side Hooks

The hooks are stored in “/opt/gitlab/embedded/service/gitlab-shell/hooks” folder. These hooks are common for all projects. It contains all 3 hooks:- pre-receive, post-receive and update hook. So, if any of the change is made to these hooks, then all the repositories will be affected by these hooks.

**Server side hooks need the same 4 pre-requisites as mentioned in the client-side hooks.**

## Creating Hooks for a Specific Project

#### Setup

Normally, git hooks are placed in the repository or project's hooks directory. GitLab creates a symlink from each project's hooks directory to the gitlab-shell hooksdirectory for ease of maintenance between gitlab-shell upgrades. As such, custom hooks are implemented a little differently. Behavior is exactly the same once the hook is created, though. Follow these steps to set up a custom hook.

1. Pick a project that needs a custom git hook.
2. On the GitLab server, navigate to the project's repository directory. For an installation from source the path is usually/home/git/repositories/<group>/<project>.git. For Omnibus installs the path is usually/var/opt/gitlab/git-data/repositories/<group>/<project>.git.
3. (Note: There is a hooks folder in the above mentioned project directory which is a symlink to the global hooks directory.)
4. Create a new directory in this location called custom\_hooks.
5. Inside the new custom\_hooks directory, create a file with a name matching the hook type. For a pre-receive hook the file name should be pre-receive with no extension.
6. Make the hook file executable and make sure it's owned by git.
7. Write the code to make the git hook function as expected. Hooks can be in any language. Ensure the 'shebang' at the top properly reflects the language type. For example, if the script is in Ruby the shebang will probably be #!/usr/bin/env ruby.

That's it! Assuming the hook code is properly implemented the hook will fire as appropriate.

For stash/bitbucket server also, you can write server-side hooks in same way as described above for Gitlab server. Just paste your hooks in ***/var/atlassian/application-data/bitbucket/shared/data/repositories*** folder.

For more information, refer atlassian Git hooks tutorial:- <https://www.atlassian.com/git/tutorials/git-hooks/>