## 1) Git config command

This command configures the user. The Git config command is the first and necessary command used on the Git command line. This command sets the author name and email address to be used with your commits. Git config is also used in other scenarios.

1. $ git config --global user.name "ImDwivedi1"
2. $ git config --global user.email "Himanshudubey481@gmail.com"

### 2) Git Init command

This command is used to create a local repository.

1. $ git init Demo

### 3) Git clone command

This command is used to make a copy of a repository from an existing URL. If I want a local copy of my repository from GitHub, this command allows creating a local copy of that repository on your local directory from the repository URL.

**Syntax**

1. $ git clone URL

### 4) Git add command

This command is used to add one or more files to staging (Index) area.

**Syntax**

To add one file

1. $ git add Filename

### 5) Git commit command

Commit command is used in two scenarios. They are as follows.

**Git commit -m**

This command changes the head. It records or snapshots the file permanently in the version history with a message.

**Syntax**

1. $ git commit -m " Commit Message"

### 6) Git status command

The status command is used to display the state of the working directory and the staging area. It allows you to see which changes have been staged, which haven't, and which files aren?t being tracked by Git. It does not show you any information about the committed project history. For this, you need to use the git log. It also lists the files that you've changed and those you still need to add or commit.

**Syntax**

1. $ git status

### 7) Git push Command

It is used to upload local repository content to a remote repository. Pushing is an act of transfer commits from your local repository to a remote repo. It's the complement to git fetch, but whereas fetching imports commits to local branches on comparatively pushing exports commits to remote branches. Remote branches are configured by using the git remote command. Pushing is capable of overwriting changes, and caution should be taken when pushing.

Git push command can be used as follows.

**Git push origin master**

This command sends the changes made on the master branch, to your remote repository.

**Syntax**

1. $ git push [variable name] master

variable name=origin

### 8) Git pull command

Pull command is used to receive data from GitHub. It fetches and merges changes on the remote server to your working directory.

**Syntax**

1. $ git pull URL

### 9) Git Branch Command

This command lists all the branches available in the repository.

**Syntax**

1. $ git branch

### 10) Git Merge Command

This command is used to merge the specified branch?s history into the current branch.

**Syntax**

1. $ git merge BranchName

### 11) Git log Command

This command is used to check the commit history.

**Syntax**

1. $ git log

### 12) Git Checkout Command

git checkout is also the means to navigate existing branches.

### 13) Git Reset Command

Resetting lets you clean up or completely remove changes that have not been pushed to a public repository.

### 14) Git Revert Command

When you discover a faulty commit, reverting is a safe and easy way to completely remove it from the code base.

### 14) Git Create New Branch

git checkout -**b** ＜new-branch＞

################################################################

| b [BUNDLE\_ID] | Lists information about a specific bundle including the bundle’s symbolic name, bundle ID, data root, registered (provided) and used services, imported and exported packages, and more |
| --- | --- |
| diag [BUNDLE\_ID] | Lists information about why the specified bundle is not working (e.g., unresolved dependencies, etc.) |
| dm na | Lists OSGi components that are “not available” (unresolved). Please see [Leveraging the Shell](http://felix.apache.org/documentation/subprojects/apache-felix-dependency-manager/tutorials/leveraging-the-shell.html) for more Dependency Manager information. |
| ds:unsatisfied | Lists all unsatisfied Declarative Services (DS) components |
| ds:unsatisfied [BUNDLE\_ID] | Lists the bundle’s unsatisfied DS components |
| headers [BUNDLE\_ID] | Lists metadata about the bundle from the bundle’s MANIFEST.MF file |
| equinox:refresh [BUNDLE\_ID] | Detects available optional dependencies and reactivates the bundle. |
| help | Lists all the available Gogo shell commands. Notice that each command has two parts to its name, separated by a colon. For example, the full name of the help command is felix:help. The first part is the command scope while the second part is the command function. The scope allows commands with the same name to be disambiguated. E.g., scope allows the felix:refresh command to be distinguished from the equinox:refresh command. |
| help [COMMAND\_NAME] | Lists information about a specific command including a description of the command, the scope of the command, and information about any flags or parameters that can be supplied when invoking the command. |
| inspect capability service [BUNDLE\_ID] | Lists services exposed by a bundle |
| install [PATH\_TO\_JAR\_FILE] | Installs the specified bundle into Liferay’s module framework |
| lb | Lists all of the bundles installed in Liferay’s module framework. Use the -s flag to list the bundles using the bundles’ symbolic names. |
| packages [PACKAGE\_NAME] | Lists all of the named package’s dependencies |
| scr:list | Lists all of the components registered in the module framework (*scr* stands for service component runtime) |
| scr:info [COMPONENT\_NAME] | Lists information about a specific component including the component’s description, services, properties, configuration, references, and more. |
| services | Lists all of the services that have been registered in the module framework |
| start [BUNDLE\_ID] | Starts the specified bundle |
| stop [BUNDLE\_ID] | Stops the specified bundle |
| system:check | Scans the system for anomalies, executing the ds:unsatisfied and dm na commands. |
| uninstall [BUNDLE\_ID] | uninstalls the specified bundle from the module framework. This does not remove the specified bundle from Liferay’s module framework; it’s hidden from Gogo’s lb command, but is still present. Adding a new version of the uninstalled bundle, therefore, will not reinstall it; it will update the currently hidden uninstalled version. To remove a bundle from Liferay’s module framework permanently, manually delete it from the [Liferay Home]/osgi folder. For more information on the uninstall command, see OSGi’s [uninstall](https://osgi.org/javadoc/r6/core/org/osgi/framework/Bundle.html#uninstall()) documentation. |

The following sub-commands are callable in the Blade CLI environment:

* *convert*: Converts a Plugins SDK plugin project to a Gradle Workspace project.
* *create*: Creates a new Liferay module project from available templates.
* *deploy*: Builds and deploys bundles to the Liferay module framework.
* *gw*: Executes Gradle command using the Gradle Wrapper, if detected.
* *help*: Gives help on a specific command.
* *init*: Initializes a new Liferay Workspace.
* *install*: Installs a bundle into Liferay’s module framework.
* *open*: Opens or imports a file or project in Liferay Dev Studio DXP.
* *samples*: Generates a sample project.
* *server*: Starts or stops server defined by your Liferay project.
* *sh*: Connects to Liferay DXP, executes succeeding Gogo command, and returns output.
* *update*: Updates Blade CLI to latest version.
* *upgradeProps:* Analyzes your old portal-ext.properties and your newly installed 7.x server to show you properties moved to OSGi configuration files or removed from the product.
* *version*: Displays version information about Blade CLI.