Helix4Git

Helix4Git Administrator Guide

2017.2

October 2017

PERFORCE

www.perforce.com

Contents

How to Use this Guide

This guide tells you how to use Helix4Git, which augments the functionality of the Helix Versioning Engine (also referred to as the Helix Server) to support Git clients. It services requests from mixed clients, that is, both "classic" Helix Server clients and Git clients, and stores Git data in Git repos that reside within a classic Helix Server depot.

Tip

- For help configuring Helix Server for building from mixed clients, see *P4 Command Reference* under **P4 Client**, the section "Including Graph Depots and repos in your client".
- For in-depth admin and usage information pertaining to the Helix Server, see:
  - *Helix Versioning Engine Administrator Guide: Fundamentals*
  - *Helix Versioning Engine User Guide*

Feedback

How can we improve this manual? Email us at manual@perforce.com.

Other documentation

See https://www.perforce.com/support/self-service-resources/documentation.

Syntax conventions

Helix documentation uses the following syntax conventions to describe command line syntax.

Notation	Meaning
`literal`	Must be used in the command exactly as shown.
italics	A parameter for which you must supply specific information. For example, for a *serverid* parameter, supply the ID of the server.
[`-f`]	The enclosed elements are optional. Omit the brackets when you compose the command.

Notation	Meaning
...	▪ Repeats as much as needed: • `alias-name[[$(arg1)... [$(argn)]]=transformation` ▪ Recursive for all directory levels: • `clone perforce:1666 //depot/main/p4... ~/local-repos/main` • `p4 repos -e //gra.../rep...`
element1 | *element2*	Either *element1* or *element2* is required.

What’s new in this guide for the 2017.2 release

This section provides a summary with links to topics in this reference. For a complete list, see the *Release Notes*.

New features

"Helix TeamHub configuration" on page 48

"Configuring Git Connector to poll repos from Helix4Git" on page 62

"Git Connector configuration for fail-over to another Git host" on page 56

Overview

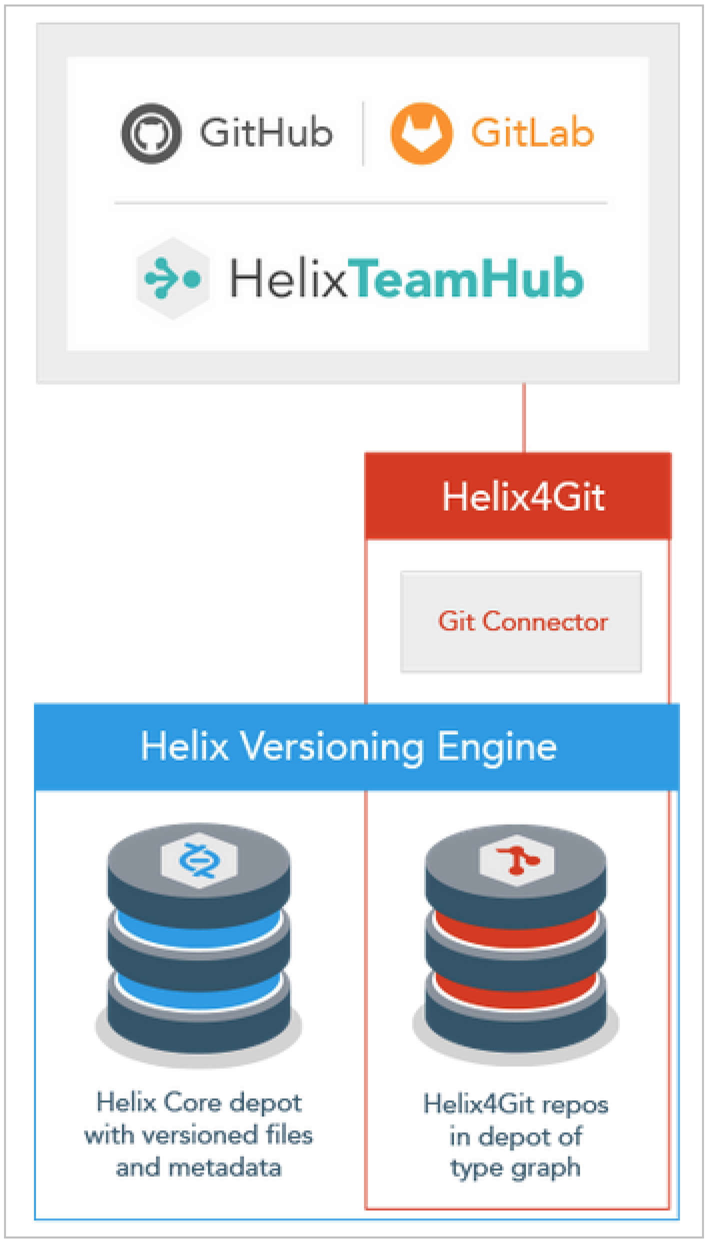
Benefits:

- Flexibility: sync any combination of repos, branches, tags, and SHA-1 hashes
- Hybrid support: you can sync data that is a mix of Git repo data and classic Helix Server depot data
- Supports "One-way mirroring from Git servers" on page 37, such as GitHub, GitLab, and Gerrit Code Review
- Automation: polling to automatically trigger a build upon updates to the workspace, and support for Jenkins
- Visibility: listing of building contents

This solution:

- stores Git repos in one or more depots of type `graph`
- services requests for the data stored in the Git repos
- supports Large File Storage (LFS) objects and service requests using HTTPS
- enforces access control on Git repos through the use of permissions granted at depot, repo, or branch level
- supports both HTTPS and SSH remote protocols
- services requests from Git clients using a combination of cached data and requests to the Helix Server
- supports clients accessing repos containing Git Large File Storage (LFS) objects (but not over SSH)

Architecture and components


GitHub
GitLab
HelixTeamHub
Helix4Git
Git Connector
Helix Versioning Engine
Helix Core depot with versioned files and metadata
Helix4Git repos in depot of type graph

Helix4Git consists of two components:

- Helix Server (or **p4d**), the traditional Helix Versioning Engine
  augmented for Git support
- The Git Connector, which acts as a Git server to Git clients,
  such as Helix TeamHub, GitLab, and GitHub.

Git users use a Git client to pull files from the graph depot to make modifications and then push the changes back into the graph depot. The Git client communicates with the Helix Server through the Git Connector.

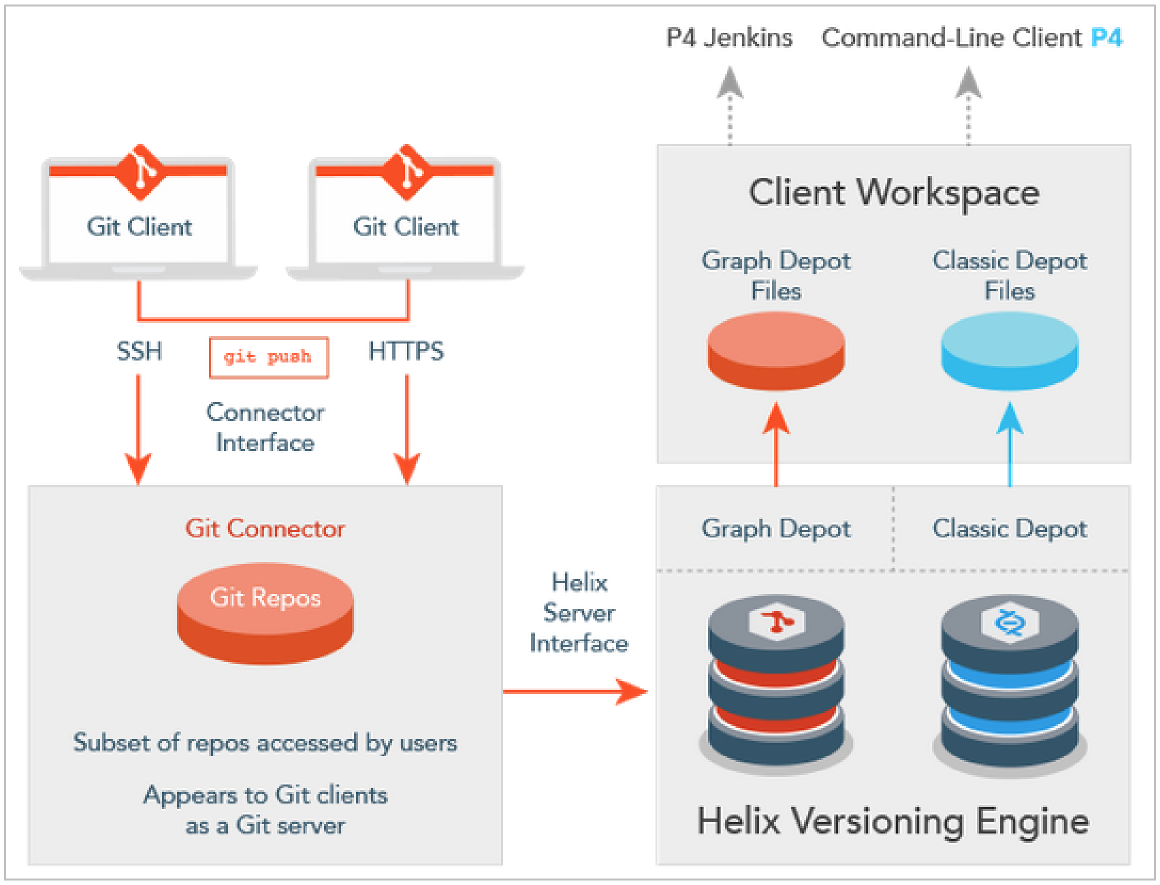
In support of advanced workflows for blended assets, such as text and large binaries in build and test automation, you can also directly **sync** and **view** graph depot content through a command line client into a single classic Helix Server workspace.

Note
To **edit** the graph depot files associated with a classic workspace, you must use a Git client.

A typical scenario:

1. A Git user pushes changes to the Git Connector.
2. The Git Connectorpushes the changes to the Helix Server.

3. A continuous integration (CI) server, such as P4Jenkins, detects changes and runs a build using *one* workspace that can include multiple Git repos and classic depot files.



P4Jenkins support

You can connect the workspace to CI tools, such as P4 Jenkins. The advantages of using the P4 Plugin for Jenkins as the continuous integration server include:

- Efficiency: being able to sync a SINGLE depot of type graph that contains MANY repos
- Hybrid support: this single depot is able to have also classic depot files
- Flexibility: sync any combination of repos, branches, tags, and SHA-1 hashes
- Automation: polling to automatically trigger a build upon updates to the workspace
- Visibility: listing of building contents

To learn how to use the P4 Plugin for Jenkins, see https://github.com/jenkinsci/p4-plugin/blob/master/GRAPH.md

Workflow

1. Install the Git Connector.
2. Configure the Git Connector, including HTTPS and SSH authentication.
3. Configure the Helix Versioning Engine to work with the Git Connector. This includes depot, repo, and permissions configuration.
4. Verify the configuration.
5. Run `p4 sync` and a subset of other p4 commands against Git repos and classic depot files.

One-time tasks

The following table summarizes one-time tasks:

Task	More information
Install the Git Connector.	"Install the Git Connector" on page 16
Configure the Git Connector. This includes configuring HTTPS and SSH authentication.	"Configure the Git Connector" on page 20
Configure the Helix Server to work with the Git Connector. This includes depot, repo, and permissions configuration.	"Perform Connector-specific Helix Server configurations" on page 22
Set up users.	"Set up Git users to work with the Git Connector" on page 25
Verify the Git Connectorconfiguration.	"Verify the Git Connector configuration" on page 28

Recurring tasks

The following table summarizes recurring tasks:

Task	More information
Create and view graph depots.	"Create graph depots" on page 30

Task	More information
Create, view, and delete Git repos.	"Create and view repos" on page 31
Manage permissions on a repo or group of repos. You can grant, revoke, and show permissions. Permissions apply at the user or group level.	"Manage access to graph depots and repos" on page 32
Set up client workspaces.	"Set up client workspaces" on page 33
Run p4 sync and a subset of other p4 commands against both Git and classic depot data.	"Sync files from graph depots" on page 34
Troubleshoot.	"Troubleshooting" on page 64

Git client tasks

Git clients must perform a couple of tasks to interact with the Git Connector:

- Obtain SSH and HTTPS URLs. See "Set up Git users to work with the Git Connector" on page 25.
- Generate SSH keys to be added to the Git Connector, if the SSH keys do not already exist.

Installation and configuration

This chapter describes how to install and configure the Git Connector. The installation requires operating system-specific packages (see "System requirements" below).

System requirements

The Git Connector requires an installation of Helix Versioning Engine 2017.1 or later.

Tip
We recommend that the Git Connector be on a machine that is separate from the machine with the Helix Server.

The Git Connector is available in two distribution package formats: Debian (`.deb`) for Ubuntu systems and RPM (`.rpm`) for CentOS and RedHat Enterprise Linux (RHEL). You can install the Git Connector on the following Linux (Intel x86_64) platforms:

- Ubuntu 14.04 LTS
- Ubuntu 16.04 LTS

- CentOs or Red Hat 6.x
  - not recommended because it requires that you manually install Git and HTTPS
  - if the operating system is CentOS 6.9, Security-Enhanced Linux (SELinux) and the iptables use-space application must allow:
    - the Git server to contact the helix/gconn service on port 443 (the HTTPS port)
    - gconn to communicate with p4d if they are both on the same machine
- CentOS or Red Hat 7.x

Note
"One-way mirroring from Git servers" on page 37 is not recommended with Centos6.

Space and memory requirements depend on the size of your Git repos and the number of concurrent Git clients.

The Git Connector works with Git version 1.8.5 or later. If the distribution package comes with an earlier release of Git, upgrade to a supported version.

Note
If your Git clients work with repos containing large file storage (LFS) objects, install Git LFS and select the files to be tracked. For details, see https://git-lfs.github.com. Git LFS requires HTTPS.

Warning

- The Helix4Git configuration process removes any SSL certificates in `/etc/apache2/ssl` before generatings new SSL certificates. Therefore, existing sites, such as Helix Swarm, might be disabled.
- Do not add custom hooks in the Git Connector because they will not work as expected. However, the Helix Versioning Engine does support triggers for depots of type graph. See https://www.perforce.com/perforce/doc.current/manuals/p4sag/#P4SAG/scripting.triggers.graph.html.

Install the Git Connector

Installing the Git Connector requires that you create a package repository file, import the package signing key, and install the package.

Before you start the installation, verify that you have root-level access to the machine that will host the Git Connector.

1. **Configure the Helix Versioning Engine package repository.**

   As `root`, perform the following steps based on your operating system:

a. **For Ubuntu 14.04:**

   Create the file `/etc/apt/sources.list.d/perforce.list` with the following content:

   ```
   deb http://package.perforce.com/apt/ubuntu trusty release
   ```

b. **For Ubuntu 16.06:**

   Create the file `/etc/apt/sources.list.d/perforce.list` with the following content:

   ```
   deb http://package.perforce.com/apt/ubuntu xenial release
   ```

c. **For CentOS/RHEL 6.x:**

   Create the file `/etc/yum.repos.d/Perforce.repo` with the following content:

   ```
   [perforce]
   name=Perforce for CentOS $releasever - $basearch
   baseurl=http://package.perforce.com/yum/rhel/6/x86_64/
   enabled=1
   gpgcheck=1
   gpgkey=http://package.perforce.com/perforce.pubkey
   ```

d. **For CentOS/RHEL 7.x:**

   Create the file `/etc/yum.repos.d/Perforce.repo` with the following content:

   ```
   [perforce]
   name=Perforce for CentOS $releasever - $basearch
   baseurl=http://package.perforce.com/yum/rhel/7/x86_64/
   enabled=1
   gpgcheck=1
   gpgkey=http://package.perforce.com/perforce.pubkey
   ```

2. **Import the Helix Versioning Engine package signing key.**

   As `root`, run the following command:

   a. **For Ubuntu 14.04 and 16.04:**

   ```
   $ wget -qO - http://package.perforce.com/perforce.pubkey |
   sudo apt-key add -
   $ sudo apt-get update
   ```

   b. **For CentOS/RHEL 6.x and 7.x:**

   ```
   $ sudo rpm --import
   http://package.perforce.com/perforce.pubkey
   ```

3. **Install the Git Connector package.**

   As `root`, run one of the following commands:

   a. **For Ubuntu 14.04 and 16.04:**

   ```
   $ sudo apt-get install helix-git-connector
   ```

   b. **For CentOS/RHEL 6.x and 7.x:**

   ```
   $ sudo yum install helix-git-connector
   ```

4. **Follow the prompts.**
5. **Configure the Git Connector**. See "Upgrading Git Connector" below and "Configure the Git Connector" on page 20.

Upgrading Git Connector

Upgrading to version 2017.2 from the 2017.1 patch

1. Verify the Git Connector server id, which, for this example, is `my-gconn-centos6`:

   `*p4 servers*`

   The response is similar to:

   ```
   my-gconn-centos6 connector git-connector ' This GitConnector
   service was configured on [Tue Aug 29 10:21:30 PDT 2017] by
   user [super]
   ```

2. Make sure that you provide the value of the `serverid` when upgrading the Git Connector:

   ```
   configure-git-connector.sh --upgrade --serverid my-gconn-
   centos6
   ```

Here is an example:

```
sudo /opt/perforce/git-connector/bin/configure-git-connector.sh --
upgrade --serverid my-gconn-centos6
hostname: Unknown host

Summary of arguments passed:
...
GitConnector SSH system user       [git]
Home directory for SSH system user [/home/git]
SSH key update interval            [10]
Server ID                          [gconn-gconn-centos6]
GitConnector hostname              [(not specified)]

For a list of other options, type Ctrl-C to exit, and then run:
$ sudo /opt/perforce/git-connector/bin/configure-git-connector.sh --help
...
Would you like to perform the upgrade of this GitConnector instance? [no]:
yes
...
Moving existing configuration file to
/opt/perforce/git-connector/gconn.conf.bak
Writing GitConnector configuration file
This GitConnector instance has been successfully upgraded.
```

Upgrading helix-git-connector that is prior to the 2017.1 patch

If you have a version of the Git Connector that is prior to the 2017.1 July patch, and you want to use Gerrit, upgrade the package and re-run the package configuration script.

1. Run as root:

Ubuntu	CentOS

`$ sudo apt-get update`
`$ sudo apt-get install helix-git-connector`
	`# yum install helix-git-connector`

2. Optionally, if your Helix Server configuration has changed, or you encounter problems, run the configuration script:

```
$ sudo /opt/perforce/git-connector/bin/configure-git-
connector.sh
```

   The configuration script:

   - warns you about the existing configuration file
   - prompts you for `P4PORT`, super user's account name, and super user password
   - updates the HTTPS and SSH authentication configurations

Configure the Git Connector

1. As `root`, run the following configuration script in interactive mode:

```
$ sudo /opt/perforce/git-connector/bin/configure-git-connector.sh
```

   In interactive mode, the configuration script displays the following summary of settings. Some settings have a default value. Other settings require that you specify a value during the configuration.

   - **Helix Server P4PORT:** The host (name or IP address) and port for the Helix Server , in the following format: `host:port`.
   - **Helix Server super-user:** The name of an existing Helix Server user with `super` level privileges. This super-user is used for all tasks related to the Helix Server, such as creating users and groups and granting permissions.
   - **Helix Server super-user password:** The password for the existing Helix Server super-user.
   - **New Graph Depot name:** The Helix Server installation automatically creates a default depot of type `graph` named `repo`. During the configuration, you can create an additional graph depot.

     A depot of type `graph` is a container for Git repos.

     A depot name must start with a letter or a number.

- **GitConnector user password:** By default, the Git Connector configuration creates a Helix Server user called `gconn-user`. This user performs the Helix Server requests. Only admins should know and set this password.

> **Note**
> If you change the `gconn-user` Helix Server password, you need to reset the password on each Git Connector by running the helper script: `/opt/perforce/git-connector/bin/login-gconn-user.sh`.

- **Configure HTTPS?:** Option to use HTTPS as authentication method. HTTPS is required if you use Git LFS.

> **Tip**
> Re-running `configure-git-connector.sh` and choosing `No` when asked if you want to configure HTTPS after having chosen `Yes` previously, does not change already configured https.

- **Configure SSH?:** Option to use SSH as authentication method.
- **GitConnector SSH system user:** The name of the SSH system user to connect to the Git Connector. By default, this is `git`.
- **Home directory for SSH system user:** The home directory for the SSH system user. By default, this is `/home/git`.
- **SSH key update interval:** How often the SSH keys are updated.

> **Tip**
> Wait 10 minutes for the keys to update. Otherwise, the Git Connector will not have the updated SSH keys in the list of authorized keys, and you will not be able to connect.

- **Server ID:** The host name of the server.

2. **Provide information to the configuration script.**

   After the summary, the configuration script prompts you for information on the Helix Server `P4PORT`, the Helix Server super-user's name and password, whether you want to create another depot of type graph, and whether you want to configure HTTPS or SSH.

   At each prompt, you can accept the proposed default value by pressing **Enter**, or you can specify your own value. If needed, you can also set values with a command line argument. For example, to specify `P4PORT` and a super-user name:

   ```
   $ sudo /opt/perforce/git-connector/bin/configure-git-
   connector.sh --p4port=ssl:IP address:1666 --super=name
   ```

   After you answer the prompts, the script creates the configuration file according to your choices. As it runs, the script displays information about the configurations taking place. The script might prompt you for more input. For example, if you opted for HTTPS support and Apache components are already present on your server.

   To see all possible configuration options, run the command:

   ```
   $ sudo /opt/perforce/git-connector/bin/configure-git-
   connector.sh --help
   ```

   This is helpful if you do not want to use the default configurations. For example, the configuration script does not prompt you for the name of the SSH user or the path to the home directory of the system user because it uses default values. If you want to overwrite these values, you need to pass in the respective parameter and argument.

3. When the configuration script has finished running, read the details to see if anything still needs to be done.

Perform Connector-specific Helix Server configurations

After installing and configuring the Git Connector, configure the Helix Server to work with the Git Connector. Tasks include:

- granting relevant permissions
- creating repos that belong to the graph depots you created during the installation
- granting users permission to push repos to the Helix Server
- configuring a client mapping to sync repos
- syncing a repo, provided the repo has already been pushed to the Helix Server

For more information on `p4` commands, see the *P4 Command Reference* or run the `p4 --help` command.

Grant permissions

The Git Connector authenticates Git users through HTTP or SSH (see "Set up Git users to work with the Git Connector" on page 25) and allows them to access resources by pull, push, and clone transactions through user or group permissions in the Helix Server.

Because the `gconn-user` performs all Helix Server requests required by the Git Connector, the `gconn-user` must have an entry in the protections table with `write` permission and have been granted `admin` permission for all graph depots manually created after the installation.

For details on Helix Server permissions, see Securing the Server in *Helix Versioning Engine Administrator Guide: Fundamentals*. For details on the `p4 protect` command, see p4 protect in the *P4 Command Reference*.

For details on access control policies related to graph depots, see "Manage access to graph depots and repos" on page 32.

Perform the following steps to grant the required permissions:

1. Add the user `gconn-user` to the protections table with `write` permission. Note that if you encounter a reference to `GConn P4 user`, this is the `gconn-user` user.

   Run the following command to open the protections table in text form:

   ```
   $ p4 protect
   ```

   Add the following line to the `Protections` field:

   ```
   $ write user gconn-user * //...
   ```

   Save the spec.

2. For any depot of type graph that you create in addition to the ones already created during the installation, grant the `gconn-user` user `admin` permission:

   ```
   $ p4 grant-permission -u gconn-user -p admin -d
   graphDepotName
   ```

3. As a superuser, grant `admin` permission to another user so that this user can manage permissions as required:

   ```
   $ p4 grant-permission -u username -p admin -d graphDepotName
   ```

4. Grant users permission to create repos for specific graph depots:

   ```
   $ p4 grant-permission -p create-repo -d graphDepotName -u
   username
   ```

5. Grant users permission to push repos to a graph depot:

   ```
   $ p4 grant-permission -p write-all -u username -d
   graphDepotName
   ```

Tip
Instead of granting permissions to single users, you can create groups, assign users to groups, and set permissions that are appropriate for that particular group. See Granting access to groups of users in *Helix Versioning Engine Administrator Guide: Fundamentals*.

Create graph depots

The Helix Server installation creates a default depot of type `graph` called `repo`. If you need to manually add additional graph depots, see "Create graph depots" on page 30.

For any additional `graph` depots that you create, grant `admin` permission to the user `gconn-user` (for details, see Granting permissions).

To view a list of existing depots, run the `p4 depots` command. See the *P4 Command Reference*.

Create repos

To create a new repo stored in an existing graph depot, run the following command:

```
$ p4 repo //graphDepotName/repo1
```

For more information on creating repos, see "Create and view repos" on page 31.

Configure a client workspace to sync repos

A client workspace is a set of directories on a user’s machine that mirrors a subset of the files in the depot. This view defines which depots you can sync to your client workspace. Classic depots are mapped by default, but to be able to sync repos from a graph depot, you need to manually edit the client workspace specification by noting the required mappings.

For more information on setting up clients, see "Set up client workspaces" on page 33.

1. Run the following command to create a depot client specification and its view:

   ```
   $ p4 client clientName
   ```

2. Edit the workspace view to meet your requirements.

   For example, to map a graph depot called `graphDepot` that includes a repo called `repo1`, the mapping could look like the following, where `workspace` is the dedicated directory on the client user's machine that contains all files located in the graph depot:

   ```
   //graphDepot/repo1/... //workspace/graphDepot/repo1/...
   ```

Sync a repo

After setting up the client workspace, you can update it to reflect the latest contents of the graph depot.

To sync a repo after the repo has been pushed to the Helix Server, run the command:

```
$ p4 sync //graphDepotName/repoName/...
```

For more information on the `p4 sync` command, see p4 sync in *P4 Command Reference*.

Set up Git users to work with the Git Connector

Depending on the network protocol you selected during the Git Connector configuration, you now need to set up either SSH or HTTPS authentication for each user and from each computer used to clone, push, and pull Git repos.

When this setup is complete, provide SSH or HTTPS URLs to Git client users. These URLs include the IP address or host name of the Git Connector and the path to the respective repo, which consists of the graph depot name and the repo name. The URLs have the following format:

- SSH:

  ```
  $ git command git@ConnectorHost:graphDepotName/repoName
  ```

- HTTPS:

  ```
  $ git command
  https://username@ConnectorHost/graphDepotName/repoName
  ```

SSH

The SSH key consists of a public/private key pair that you create for each user on each computer used as a Git client. Git users who already have an SSH key can send the public key to their administrator for further handling.

When you have the SSH key, you can share the public key with the Helix Server machine and then verify the key in the Git Connector server. By default, it takes 10 minutes for the SSH key shared with the Helix Server to be authorized in the Git Connector server, so you need to wait before you proceed to the verification step.

Note
Helix Server users who have, at a minimum, the `list` access to a filename in the protections table can add their own public SSH keys to the Helix Server. For example:

```
p4 pubkey -i -s scopeName < my_id_rsa.pub
```

A Helix Server user with the access level of `super` or `admin` can add a key for another user by specifying the (`-u`) option. For example:

```
p4 pubkey -i -s scopeName -u bruno < bruno_id_rsa.pub
```

See Prerequisites for a user to upload a key in *P4 Command Reference*.

Tip
If you have several public keys, you can define a **scope** for each key to be able to quickly distinguish between them. This is useful if you need to delete a key. To get a list of keys along with their scope, run the `p4 -ztag pubkeys` command. For examples, see https://www.perforce.com/perforce/doc.current/manuals/cmdref/p4_pubkeys.html.

1. To create the SSH key, run the following command and follow the prompts:

   ```
   $ ssh-keygen -t rsa
   ```

2. Let us assume:
   - You are a user with admin or superuser privilege on the Helix server, but you are NOT logged in to Helix server as an admin or superuser from the host running the command
   - `P4PORT` is set in your environment
   - a user named bruno, `P4USER=bruno`, has emailed his `id_rsa.pub` file to you and that file is stored in `/drive/userA/id_rsa.pub`

   To add the key to the Helix Server machine, you run the command:

   ```
   $ p4 -u admin pubkey -u bruno -s scopeName -i <
   /drive/userA/id_rsa.pub
   ```

   However, if `P4PORT` is NOT set, include the server name and port number:

   ```
   $ p4 -p helixserver:1666 -u admin pubkey -u bruno -s
   scopeName -i < /drive/userA/id_rsa.pub
   ```

   Note
   Users without `admin` permission need to run this command without the `-u` option:

   ```
   $ p4 pubkey -i -s scopeName < ~/.ssh/id_rsa.pub
   ```

   Otherwise, they receive the following error message:

   ```
   You don't have permission for this operation.
   ```

3. Wait 10 minutes for the keys to update. Otherwise, the Git Connector will not have the updated SSH keys in the list of authorized keys, and you will not be able to connect.
4. Have Git client users run the following command to verify that they can successfully connect to the Git Connector.This command is similar to the `p4 info` command in that it displays information about the installed applications.

   ```
   $ git clone git@ConnectorHost:@info
   ```

   Note
   Ignore the following message:

   ```
   fatal: Could not read from remote repository. Please make sure you
   have the correct access rights and the repository exists.
   ```

   If you see `p4 info` output, the command was successful.

   If you are prompted for the Git password, this indicates an issue with the SSH setup. See "Troubleshooting" on page 64.

HTTPS

Using HTTPS requires that you have a user account and password for the Helix Server. You need to enter these credentials when prompted, which is every time you try to connect to the Git Connector to push, pull, or clone.

- To turn off SSL verification in Git, run one of the following commands:

  ```
  $ export GIT_SSL_NO_VERIFY=true
  ```

  ```
  $ git config --global http.sslVerify false
  ```

Verify the Git Connector configuration

You already verified that the SSH key was added to the list of authorized keys in the Git Connector server as part of "Set up Git users to work with the Git Connector" on page 25. In addition, you can verify the Git Connector version installed by having Git users run the following command on the Git client machine:

When using SSH:

```
$ git clone git@ConnectorHost:@info
```

When using HTTPS:

```
$ git clone https://ConnectorHost/@info
```

Push, clone, and pull repos

After you have installed and configured the Git Connector and have verified the installation, you can start pushing repos from a Git client to a depot of type `graph` in the Helix Server. You can then clone those repos to other Git clients as needed or, if you already have the repo on your Git client, pull changes from the Helix Server .

Any Git user with `write-all` permission for the respective depots and repos in the Helix Server can push, clone, and pull through the Git Connector. For details, see Granting permissions.

SSH syntax

To push a repo into the Helix Server using SSH, run the following command:

```
$ git push git@ConnectorHost:graphDepotName/repoName
```

To clone a repo from the Helix Server using SSH, run the following command:

```
$ git clone git@ConnectorHost:graphDepotName/repoName
```

To pull a repo from the Helix Server using SSH, run the following command:

```
$ git pull git@ConnectorHost:graphDepotName/repoName
```

HTTPS syntax

To push a repo into the Helix Server using HTTPS, run the following command:

```
$ git push https://ConnectorHost/graphDepotName/repoName
```

To clone a repo from the Helix Server using HTTPS, run the following command:

```
$ git clone https://ConnectorHost/graphDepotName/repoName
```

To pull a repo from the Helix Server using HTTPS, run the following command:

```
$ git pull https://ConnectorHost/graphDepotName/repoName
```

Depots and repos

All versioned files that users work with reside in a shared repository called a *depot*. By default, a depot named `depot` of type `local` is created in the Helix Versioning Engine (the Helix Server) when the server starts up. This kind of depot is also referred to as a *classic* depot. In addition, the Helix Server installation creates a default graph depot named `repo`. A graph depot is a depot of type `graph` that serves as a container for Git repos.

Create graph depots

A graph depot can hold zero or more repositories. There is no upper limit to the number of repos that you can store in a single graph depot. You can also manually create additional graph depots at any time by running the `p4 depot` command. This command is used to create any type of depot. For details, see *P4 Command Reference* or run the `p4 help` command.

Make sure to grant `admin` permission to the `gconn-user` on any manually created graph depots. For instructions, see Granting permissions.

You can view a list of the graph depots on your server by running the `p4 depots` command. with the `--depot-type=graph` option, as follows:

```
$ p4 depots --depot-type=graph
```

or (shorter):

```
$ p4 depots -t graph
```

When you create a new depot (of any type), the resulting form that opens is called the depot spec. The depot spec for a graph depot:

- gives the graph depot a name
- establishes an owner for the depot

  The owner has certain privileges for all repos in a graph depot and automatically acquires depot-wide `admin` privileges.
- defines a storage location for the archives and Git LFS files for all repos in a graph depot

A graph depot does not use the `p4 protect` mechanism at the file level. Instead, a graph depot supports the Git model with a set of permissions for an entire repo of files. For details, see Managing access control to graph depots and repos.

1. To create a new graph depot, run the following command:

   ```
   $ p4 depot -t graph graphDepotName
   ```

2. Edit the resulting spec as needed.

   For information on the available form fields, see p4 depot in *P4 Command Reference*.

Create and view repos

Similar to the depot spec, each Git repo stored in the Helix Server is represented by a repo spec. You can create, update, and delete repo specs by running the `p4 repo` command.

Note
Helix4Git supports a maximum of 10 repos per license. To obtain more licenses, please contact your Perforce Sales representative.

Each repo has an owner (a user or a group). By default, this is the user who creates the repo. The owner automatically acquires repo-wide `admin` privileges and is responsible for managing access controls for that repo.

In addition, the repo spec includes the repo name and information on when the repo was created as well as the time and date of the last push. The spec also lets you specify:

- a description of the remote server
- a default branch to clone from

  If you do not specify a default branch here, the default branch is `refs/heads/master`. If your project uses another name, see "Specify a default branch" on the facing page.
- the upstream URL that the repo is mirrored from

  The `MirroredFrom` field is updated automatically during mirroring configuration. For details, see the chapter "One-way mirroring from Git servers" on page 37.

It is possible to enable automatic creation of a repo when you use the `git push` command to push a new repo into the Helix Server. You configure this behavior with the `p4 grant-permission` command. For details, see "Manage access to graph depots and repos" on the facing page and p4 grant-permission in *P4 Command Reference*.

You can view a list of the Git repos on your server by runnnig the `p4 repos` command. Similarly, Git users can run the following command to view a list of repos:

```
$ git clone git@ConnectorHost:@list
```

1. To create a new Git repo in an existing graph depot, run the following command:

   ```
   $ p4 repo //graphDepotName/repoName
   ```

2. Edit the resulting spec as needed.

For more information, see p4 repo in *P4 Command Reference*.

Specify a default branch

If your project uses a name other than `master` as the default branch name, make sure to specify this name in the `DefaultBranch` field of the repo spec as a full Git ref, such as `refs/heads/main`. Otherwise, if this field is left blank, the Git Connector assumes that your default branch to clone is `master`. This would mean that you need to:

- add the branch name to the Git command every time you push to, clone, or check out the branch.
- manually check out the branch *after* you clone it.

To make your work easier, specify a default branch. For example, to make `main` the default branch, you need to add the following line to the repo spec:

```
$ DefaultBranch: refs/heads/main
```

Setting the `DefaultBranch` field in the repo spec simplifies pushing and cloning branches.

In addition, you can push:

- a single branch by specifying the branch name, which creates a repo with only that branch:

  ```
  $ git push git@ConnectorHost:graphDepotName/repoName
  branchName
  ```

- all branches by passing in the `--all` option, which creates a repo with all branches:

  ```
  $ git push git@ConnectorHost:graphDepotName/repoName --all
  ```

- all branches and Git tags by passing in the `"*:*"` option, which creates a repo with all branches and Git tags.

  ```
  $ git push git@ConnectorHost:graphDepotName/repoName "*:*"
  ```

Manage access to graph depots and repos

With the `p4 grant-permission` command, you can control access rights of users and groups to graph depots and their underlying repos. This includes permissions to:

- create, delete, and view repos
- update, force-push, delete, and create branches and branch references
- write to specific files only

  This allows for scenarios where a user can clone a repo but may only push changes to a subset of the files in that repo.

- delegate the administration of authorizations to the owner of a depot or repo

  In most cases, delegating authorization management at the graph depot level should suffice because related repos typically reside in the same graph depot. However, if needed, repo owners can grant and revoke permissions for their repos.

For example, to grant user `bruno` permission to read and update files in graph depot `graphDepot`, you can run the following command:

```
$ p4 grant-permission -d graphDepot -u bruno -p write-all
```

To limit this permission to repo `repo1`, which resides in depot `graphDepot`, you can run the following command:

```
$ p4 grant-permission -n //graphDepot/repo1 -u bruno -p write-all
```

By default, the following users have permission to run the `p4 grant-permission` command:

- The owner of the graph depot or repo
- The `superuser` user for all graph depots
- `admin` users for a particular graph depot or repo

You can view access controls by running the `p4 show-permission` command. To revoke access controls, you can run the `p4 revoke-permission` command.

For initial setup instructions, see Granting permissions.

For a detailed list of permissions and their description, see p4 grant-permission in *P4 Command Reference*.

Set up client workspaces

A client workspace is a set of directories on a user's machine that mirrors a subset of the files in the depot. More precisely, it is a named mapping of depot files to workspace files. The workspace view defines which depots you can sync to your client workspace.

A view consists of mappings, one per line. The left-hand side of the mapping specifies the depot files and the right-hand side the location in the workspace where the depot files reside when they are retrieved from the depot.

When you create a client workspace, a classic depot is mapped to your workspace by default. However, a depot of type graph requires that you manually configure the mapping by editing the `View` field in the client workspace specification. You can also edit the spec to view only a portion of a depot or to change the correspondence between depot and workspace locations.

In the following example, a graph depot called `graphDepot` includes a repository called `repo1`. It is mapped to a dedicated folder called `workspace` such that all files located in the `//graphDepot/repo1` directory on the Helix Server appear in the `//workspace/graphDepot/repo1` directory on the machine where the client workspace resides.

```
//graphDepot/repo1/... //workspace/graphDepot/repo1/...
```

For advanced workflows, you could also have a mixed workspace to accommodate the mapping of both a classic depot and a graph depot. In this case, your mapping could look like this:

```
//graphDepot/repo1/... //mixed-client/graphDepot/repo1...
//depot1/moduleA/... //mixed-client/depot1/moduleA/...
```

For more information on mixed client workspaces, see Including Graph Depots and repos in your client in *P4 Command Reference*.

For more information on configuring workspace views, see Configure workspace views in *Helix Versioning Engine User Guide*.

1. To create a depot client specification and its view, run the following command:

   ```
   $ p4 client clientName
   ```

2. Edit the workspace view to meet your requirements.

Sync files from graph depots

You can sync an entire graph depot or one or more repos to a client workspace with appropriate mappings using the `p4 sync` command. When syncing information from a graph depot, this command can only take on a limited number of options.

By default, if you do not specify a branch, `p4 sync` syncs the `master` branch of the repo, unless the `DefaultBranch` field in the repo spec specifies a different branch (for more information on specifying a default branch, see "Specify a default branch" on page 32). You can also append the branch name to the command to sync a different branch, as follows:

```
$ p4 sync branchName
```

In addition, you can sync:

- a Git commit associated with a SHA-1 hashkey
- a particular reference or commit of a repo
- repos associated with a specific label
- repos/files containing a Helix Server wildcard

Note that it is not possible to sync individual files with the `p4 sync` command. You can only gain control of individual files if you specify them in the `View` field of the client workspace specification. Otherwise, the whole repo is synced, even if you specify a file in the command line.

For details and a list of examples, see Working with a depot of type graph in *P4 Command Reference*.

Sync using an automatic label

Helix Server's automatic label feature enables you to specify which repos you want to sync with which branches, tags, or commits. This enables you to sync to multiple repos, not all of which are at the same branch, tag, or commit.

This might be useful when you are building a Git project that is dependent on other projects that are at a particular release version, tag, or commit (SHA-1). In non-Helix Server Git solutions, the manifest file traditionally performs this function.

Note
To sync more narrowly than at the repo level, use the `View` field in the client (workspace) specification. See the topic p4 client in *P4 Command Reference*.

To use automatic labels with Git repos, you edit the label specification (spec) by issuing the `p4 label` command. In particular, you edit two fields: `Revision` and `View`:

- The `Revision` field must *always* be set to `"#head"` when using automatic labels with Git repo data.
- The `View` field contents vary according to what you want to sync to.

With the following label spec settings, Helix Server syncs:

- the collection of repos under depot `//android` to tag `android-7.1.1_r23`
- the collection of repos under `//android/platform/build` to branch `master`
- the repo `//android/platform/build/kati` to commit SHA-1 `341a2ceccb836ab23f92c0ba96d0a0e73142576`

```
# A Perforce Label Specification.
#
#  Label:       release1_build
#  Update:      The date this specification was last modified.
#  Access:      The date of the last 'labelsync' on this label.
#  Owner:       bruno
#  Options:     Label update options: [un]locked, [no]autoreload.
#  Revision:    "#head"
#  View:        Lines to select depot files for the label.
#
# Use 'p4 help label' to see more about label views.

Label:  release1_build

Owner:  bruno
```

```

Description:
        Created by bruno.

Options:        unlocked noautoreload

Revision:    "#head"

#  View:        Lines to select depot files for the label.
View:
        //android/…@refs/tags/android-7.1.1_r23
        //android/platform/build/…@master

//android/platform/build/kati/…@341a2ceccb836ab23f92c0ba96d0a0e73142576
```

For more information on automatic labels, see the chapter Labels in *Helix Versioning Engine User Guide*.

One-way mirroring from Git servers

Helix4Git can duplicate ("mirror") commits from a Git repo managed by one of the following Git servers:

- GitHub
- GitLab (Community Edition or Enterprise Edition)
- Gerrit Code Review
- Helix TeamHub

A typical use case for mirroring one or more external Git repos into Helix is to enable a single instance of a CI tool, such as Jenkins, to build a complex job that syncs contents from both classic Helix and Git repos.

The mirroring is one-way: from the Git server into Helix.

Tip
`graph-push-commit` triggers are supported with mirroring. See the *Helix Versioning Engine Administrator Guide: Fundamentals* chapter on "Using triggers to customize behavior".

You, the system administrator for Helix and the Git server, configure a webhook in the Git server and the Git Connector server, which enables this flow:

1. A Git user pushes a branch to the Git server.
2. The external repo in the Git server receives a commit of a Git repo or tag, which fires the webhook.
3. The Git Connector receives the webhook message and fetches the commit from the Git server repo that is the source for mirroring.
4. The Helix Server receives the update from the Git Connector.
5. Optionally, a CI tool, such as Jenkins, polls on a Helix workspace to detect changes across multiple repos and performs a build.

GitHub or GitLab configuration

"GitHub or GitLab HTTP " below

"GitHub or GitLab SSH" on page 40

GitHub or GitLab HTTP

Tip
If the repo is private or internal, consider creating an personal access token:

- For GitHub - Creating a personal access token for the command line - https://help.github.com/articles/creating-a-personal-access-token-for-the-command-line/
- For GitLab - Personal access tokens - https://docs.gitlab.com/ce/user/profile/personal_access_tokens.html

1. Log into the Git Connector server as root.
2. Set the environment variable `GCONN_CONFIG` to the absolute path to the `gconn.conf` file:

```
export GCONN_CONFIG=/opt/perforce/git-connector/gconn.conf
```

3. Configure the webhook for mirroring:

   Important
   The target repo must NOT already exist in Helix Server.

   The source repo must not be empty.

   ```
   ./bin/gconn --mirrorhooks add graphDepotName/repoName
   https://access-token:secret@GitHost.com/project/repoName.git
   ```

   where *access-token:secret* relates to your GitHub or GitLab personal access token.

   Tip
   Copy the URL from your project's HTTP drop-down box.

4. Save the webhook **secret** token that the `--mirrorhooks` command generates, which is not related to the personal access token for GitHub or GitLab.

   Tip
   The secret token is also stored in `/opt/perforce/git-connector/repos/`*`graphDepotName/repoName`*`.git/.mirror.config`

Mirror a repo over HTTP

1. Go to the hooks URL, which might resemble `https://`*`GitHost`*`/`*`project`*`/`*`repo`*`/hooks` and represents the web hook URL for your Git client:

   - For GitLab see https://docs.gitlab.com/ce/user/project/integrations/webhooks.html
   - For GitHub, see https://developer.github.com/webhooks/creating/

2. Paste the URL of the Git Connector into the **URL** text box: `https://`*`GitConnector.com`*`/mirrorhooks`
3. Paste the webhook **secret** token in the **Secret Token** text box.
4. Uncheck **Enable SSL verification**.
5. Click **Add Webhook**.
6. Click the lower right corner **Test** button to validate the web hook is correctly set up.

Troubleshooting

If there are any issues, review the following files, or send them to Perforce Technical Support:

```
/opt/perforce/git-
connector/repos/graphDepot/repoName.git/.mirror.config
```

```
/opt/perforce/git-connector/repos/graphDepot/repoName.git/push_log
```

```
/opt/perforce/git-connector/repos/graphDepot/repoName.git/fetch_log
```

```
/opt/perforce/git-connector/repos/graphDepot/repoName.git/.mirror.log
```

```
/opt/perforce/git-connector/gconn.conf
```

```
/opt/perforce/git-connector/logs/gconn.log
```

```
/opt/perforce/git-connector/logs/p4gc.log
```

GitHub or GitLab SSH

1. On the Git Connector server, log in as the `root` user and use the `su` command to become a ***`web-service-user`***.

Ubuntu	CentOS
`su -s /bin/bash - www-data`	`su -s /bin/bash - apache`

2. Create a `.ssh` directory for the ***`web-service-user`*** user:
   `mkdir /var/www/.ssh`
3. Assign the owner of the directory:
   `chown` ***`web-service-user`***`:gconn-auth /var/www/.ssh`
4. Switch to the ***`web-service-user`***:

   `su -s /bin/bash -` ***`web-service-user`***

   and generate the public and private SSH keys for the Git Connector instance:

   `ssh-keygen -t rsa -b 4096 -C` ***`web-service-user@gitConnector.com`***

   then follow the prompts.
5. Locate the public key:
   `/var/www/.ssh/id_rsa.pub`
6. Copy this public key to the GitLab or GitHub server and add `/var/www/.ssh/id_rsa.pub` to the user account (*helix-user*) that performs clone and fetch for mirroring.

7. On the Git Connector, as the `web-service-user`, set the environment variable `GCONN_CONFIG` to the absolute path to the `gconn.conf` file:

   ```
   export GCONN_CONFIG=/opt/perforce/git-connector/gconn.conf
   ```

8. Configure the mirror hooks by running the following as the `web-service-user`:

   Important
   The target repo must NOT already exist in Helix Server.

   The source repo must not be empty.

   ```
   ./bin/gconn --mirrorhooks add graphDepotName/repoName
   git@GitHost.com/project/repoName.git
   ```

   Tip
   Copy the URL from your project's SSH drop-down box.

9. Save the secret token that the `--mirrorhooks` command generates.

   Tip
   The secret token is also stored in `/opt/perforce/git-connector/repos/`*`graphDepotName/repoName`*`.git/.mirror.config`

Mirror a repo over SSH

1. Go to https://*GitHost*.com/project/*repoName*/hooks
2. Paste the URL of the Git Connector into the **URL** text box: *https://GitConnector.com/mirrorhooks*
3. Paste the webhook secret token in the **Secret Token** text box.
4. Uncheck **Enable SSL verification**.
5. Click **Add Webhook**.
6. Click the lower-right corner **Test** button to validate the web hook is correctly set up

Troubleshooting

If there are any issues, review the following files, or send them to Perforce Technical Support:

```
/opt/perforce/git-
connector/repos/graphDepot/repoName.git/.mirror.config
```

```
/opt/perforce/git-connector/repos/graphDepot/repoName.git/push_log
```

```
/opt/perforce/git-connector/repos/graphDepot/repoName.git/fetch_log
```

```
/opt/perforce/git-connector/repos/graphDepot/repoName.git/.mirror.log
```

```
/opt/perforce/git-connector/gconn.conf
```

```
/opt/perforce/git-connector/logs/gconn.log
```

```
/opt/perforce/git-connector/logs/p4gc.log
```

Gerrit configuration

Perforce provides a custom Python plug-in script named `gconn-change-merged.py`. When properly renamed, the script enables Gerrit to generate a webhook for a specific type of Git commit, either change-merged or ref-update. You might want to have two copies of the script, one for each type of action.

System requirements with Gerrit

- Helix Git Connector 2017.1 July patch
  - If your installation of the Git Connector is prior to the July 2017 patch, see "Upgrading Git Connector" on page 18.
- Gerrit version 2.13 or 2.14 installed and working on the Git server with Python version of 2.7.x. or later
- The Perforce webhook for Gerrit `gconn-change-merged.py`, which is in the `/opt/perforce/git-connector/bin` directory of the Git Connector
- A user in the Gerrit application that is limited to the minimal privileges necessary for mirroring
- A source repo in Gerrit that already exists and is not empty

Important
The target repo must NOT already exist in Helix Server.

The source repo must NOT be empty.

Next step

Installation and script renaming

Installation of the mirror hooks

On the Gerrit server

1. Transfer the `/opt/perforce/git-connector/bin/gconn-change-merged.py` file from the Git Connector into the `hooks` subdirectory of your Gerrit installation.
2. Rename the file in the `hooks` directory to `changed-merged`:
   ```
   mv gconn-change-merged.py changed-merged
   ```
   The hook `changed-merged` enables the default Gerrit behavior of a mandatory code review of a repo before merging it into a protected branch.

   > **Tip**
   > If your organization allows direct **ref** commits without a mandatory code review, make a second copy in the `hooks` subdirectory, this time with `ref-update` as the name:
   > ```
   > cp changed-merged ref-update
   > ```
   > The name `ref-update` enables direct **ref** commits.

3. Make `changed-merged` (and, optionally, `ref-update`) executable by the OS user running Gerrit.

Configure Gerrit for HTTP

On the Git Connector server

1. Log into the Git Connector server as root.
2. Set the environment variable `GCONN_CONFIG` to the absolute path to the `gconn.conf` file:
   ```
   export GCONN_CONFIG=/opt/perforce/git-connector/gconn.conf
   ```

3. Configure the webhook for mirroring:

   Important
   The target repo must NOT already exist in Helix Server.

   The source repo must not be empty.

   ```
   ./bin/gconn --mirrorhooks add graphDepotName/repoName
   https://access-
   token:secret@GerritHost.com/project/repoName.git
   ```

   Tip
   Copy the URL from your project's HTTP drop-down box.

   If the repo is private or internal, create an access token when configuring the mirror hooks, as in the example above.

4. Save the secret token that the `--mirrorhooks` command generates.

   Tip
   The secret token is also stored in `/opt/perforce/git-connector/repos/graphDepotName/repoName.git/.mirror.config`

On the Gerrit server

1. Update the configuration file for the Gerrit repository in the $GERRIT_SITE/git/*repoName*/config file,
   where $GERRIT_SITE represents the root directory of your Gerrit server.

   ```
   [gconn]

       mirror-url = https://GitConnector.com/mirrorhooks

       token = <secret_token from /opt/perforce/git-
       connector/repos/graphDepot/repoName.git/.mirror.config>

       git-http-url = <upstream_url from /opt/perforce/git-
       connector/repos/graphDepot/repoName.git/.mirror.config>

   [gconn "http"]

       sslverify = false
   ```

Next step

"Testing the mirror hook " on page 47

Configure Gerrit for SSH

Set up the SSH keys

1. On the Git Connector server, log in as the `root` user and use the `su` command to become a *`web-service-user`*.

Ubuntu	CentOS
`su -s /bin/bash - www-data`	`su -s /bin/bash - apache`

2. Create a `.ssh` directory for the *`web-service-user`* user:

   ```
   mkdir /var/www/.ssh
   ```

3. Assign the owner of the directory:

   ```
   chown web-service-user:gconn-auth /var/www/.ssh
   ```

4. Switch to the *`web-service-user`*:

   ```
   su -s /bin/bash - web-service-user
   ```

   and generate the public and private SSH keys for the Git Connector instance:

   ```
   ssh-keygen -t rsa -b 4096 -C web-service-
   user@gitConnector.com
   ```

   then follow the prompts.

5. Locate the public key:

   ```
    /var/www/.ssh/id_rsa.pub
   ```

6. Copy this public key to the Gerrit server and add `/var/www/.ssh/id_rsa.pub` to the user account (*helix-user*) that performs clone and fetch for mirroring.

7. On the Git Connector, as the `web-service-user`, set the environment variable `GCONN_CONFIG` to the absolute path to the `gconn.conf` file:

   ```
   export GCONN_CONFIG=/opt/perforce/git-connector/gconn.conf
   ```

8. Configure the mirror hooks by running the following as the `web-service-user`:

   > **Important**
   > The target repo must NOT already exist in Helix Server.
   >
   > The source repo must not be empty.

   ```
   ./bin/gconn --mirrorhooks add graphDepotName/repoName
   ssh://helix-user@GerritHost.com/repoName.git
   ```

9. Save the secret token that the `--mirrorhooks` command generates.

   > **Tip**
   > The secret token is also stored in `/opt/perforce/git-connector/repos/graphDepotName/repoName.git/.mirror.config`

On the Gerrit server

1. Update the configuration file for the Gerrit repository in the GERRIT_SITE/git/*repoName*/config file,
   where GERRIT_SITE represents the root directory of your Gerrit server.

   ```
   [gconn]
       mirror-url = https://GitConnector.com/mirrorhooks
       token = <secret_token from /opt/perforce/git-
       connector/repos/graphDepot/repoName.git/.mirror.config>
       git-ssh-url = <upstream_url from /opt/perforce/git-
       connector/repos/graphDepot/repoName.git/.mirror.config>
   [gconn "http"]
       sslverify = false
   ```

Next step

"Testing the mirror hook " on the facing page

Testing the mirror hook

On the Gerrit server

1. Set the environment variable `GIT_DIR` to the absolute path to the Gerrit repository:
   ```
   export GIT_DIR=GERRIT_SITE/git/repoName.git
   ```
   where GERRIT_SITE represents the root directory of your Gerrit server.
2. From the `GERRIT_SITE` directory, issue the command:
   ```
   ./hooks/change-merged
   ```
3. Check whether the hook displays the message that indicates successful mirroring:
   ```
   GConn Hook HTTP response: mirror from
   http://GerritHost.com/repoName.git to
   //graphDepot/repoName.git
   ```
4. If there are problems, see "Troubleshooting Gerrit one-way mirroring" below.

Troubleshooting Gerrit one-way mirroring

Note
Mirroring occurs upon commit or merge (depending on the Gerrit workflow), so pushing a Gerrit code review on a pseudo-branch, such as

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

is not sufficient to fire the webhook.

Important
To verify which repo is being mirrored, at the Git Connector command line, issue the following command:

```
bin/gconn --mirrorhooks list
```

The response might be similar to:

```
//graphDepot/repoName <<< http://GerritHost.com/repoName.git
```

which indicates that the `//graphDepot/repoName` destination repo mirrors the `http://GerritHost.com/repoName.git` source repo.

Tip
To view command-line help:

From the `GERRIT_SITE` directory, issue the command:

```
./hooks/change-merged --help
```

If there are any issues, review the following files, or send them to Perforce Technical Support:

On the Gerrit server:

`GERRIT_SITE/git/`*`repoName`*`.git/config`

On the Git Connector server:

```
/opt/perforce/git-
connector/repos/graphDepot/repoName.git/.mirror.config
```

```
/opt/perforce/git-connector/repos/graphDepot/repoName.git/push_log
```

```
/opt/perforce/git-connector/repos/graphDepot/repoName.git/fetch_log
```

```
/opt/perforce/git-connector/repos/graphDepot/repoName.git/.mirror.log
```

```
/opt/perforce/git-connector/gconn.conf
```

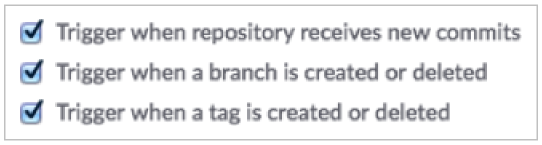
```
/opt/perforce/git-connector/logs/gconn.log
```

```
/opt/perforce/git-connector/logs/p4gc.log
```

Helix TeamHub configuration

Overview

You, the administrator of Helix TeamHub and Helix4Git, can set up mirroring a Git repository into the Helix Versioning Engine. You can choose what triggers mirroring to occur:


✓ Trigger when repository receives new commits
✓ Trigger when a branch is created or deleted
✓ Trigger when a tag is created or deleted

Sequence of events

1. An end-user does a git push from the local computer to the Helix TeamHub server.
2. The user's action fires a Repository Webhook in Helix TeamHub to notify the Helix Git Connector.
3. Helix Git Connector fetches the new changes.
4. The Helix Git Connector mirrors the Git repo into the specified Helix graph depot.
5. Optionally, an automated build occurs, using a tool such as Jenkins, which is supported by p4Jenkins.

Limitations

This use case is for Helix TeamHub **on-premise**, not the cloud version of Helix TeamHub.

Repo access is through username/password or SSH key. Helix TeamHub on-premise does not support for SSO or two-factor authentication.

For mirroring, use a repository hook, not a company hook or a project hook. For details, see https://helixteamhub.cloud/docs/user/webhooks/general/.

Authentication

Both HTTP and SSH are supported. To use SSH, the public key needs to be added to Helix TeamHub.

System requirements

- Ubuntu 14.04 LTS, Ubuntu 16.04 LTS, CentOS or Red Hat 6.x, CentOS or Red Hat 7.x
- Must be an administrator for a working Helix TeamHub, so that you can set up a Repository Webhook
- Must be an administrator on the Git Connector server, so you can run the command to add a mirror hook
- A working Git Connector with patch release string 2017.1/1572461
- A working Helix Versioning Engine server, either 17.1 patch 2017.1/1574018 or 17.2
- A Helix TeamHub repository that is not empty. This repository will be the source for mirroring into the Helix graph depot.
- A Helix TeamHub bot account you can use instead of personal credentials. The two options are:
  - A regular bot with access to relevant projects and repositories on the team view
  - A company admin bot account, which has access to every repository inside the company

  For more information, see https://helixteamhub.cloud/docs/user/bots/

Installation of Helix TeamHub On-Premise

You, the Helix4Git administrator:

1. Go to https://www.perforce.com/downloads/helix-teamhub-enterprise
2. Locate the Helix TeamHub package to download.
   See the installation instructions at https://helixteamhub.cloud/docs/admin/getting-started/ or https://helixteamhub.cloud/docs/admin/installation/combo/
3. Run the package for an on-premise installation of Helix TeamHub.

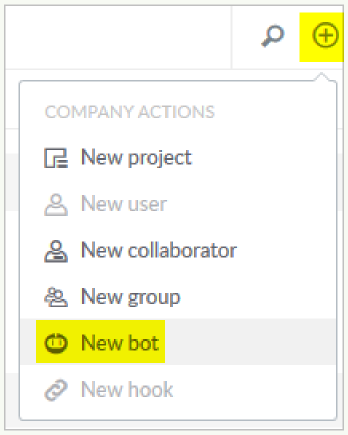
Next step

Configure for HTTP or SSH:

Helix TeamHub HTTP

Tip
Use a bot account instead of personal credentials. The two options are:

- Use a regular bot, and give it access to relevant projects and repositories on the team view
- Use a company admin bot account, which has access to every repository inside the company



For more information, see https://helixteamhub.cloud/docs/user/bots/

1. Log into the Git Connector server as root.
2. Set the environment variable `GCONN_CONFIG` to the absolute path to the `gconn.conf` file:

   ```
   export GCONN_CONFIG=/opt/perforce/git-connector/gconn.conf
   ```
3. Configure the webhook for mirroring:

   Important
   The target repo must NOT already exist in Helix Server.

   The source repo must not be empty.

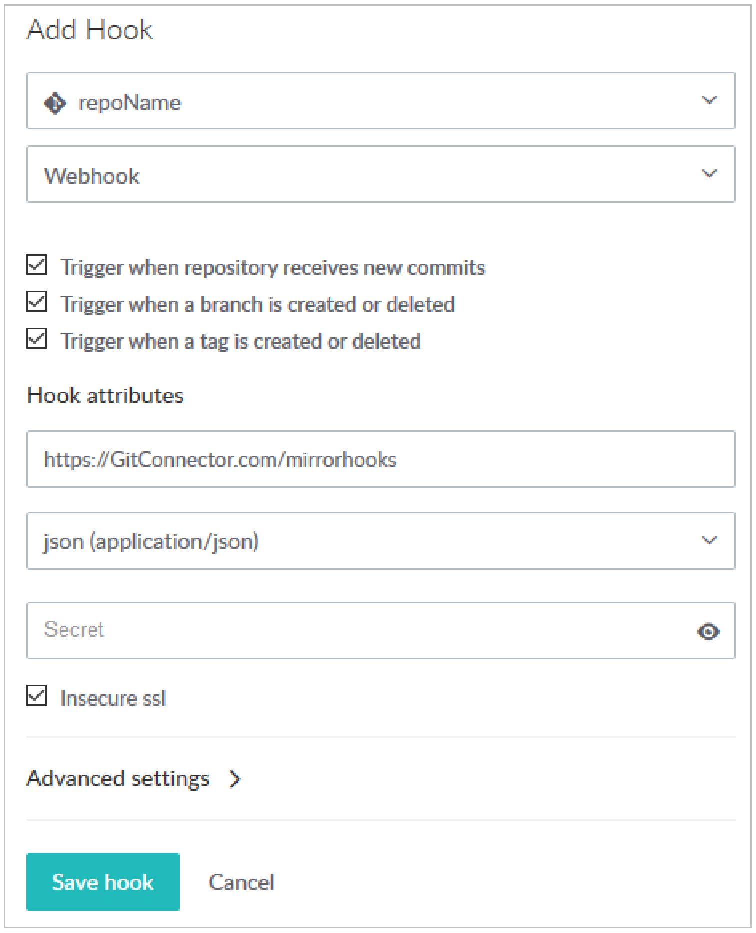
   ```
   ./bin/gconn --mirrorhooks add graphDepotName/repoName
   https://bot:password@
   HelixTeamHubServer
   /companyName/projects/projectName/repositories/git/repoName
   ```

   Tip
   Copy the URL from your project's HTTP drop-down box.

4. Save the **secret** token that the `--mirrorhooks` command generates.

> **Tip**
> The secret token is also stored in `/opt/perforce/git-connector/repos/`*`graphDepotName/repoName`*`.git/.mirror.config`

Mirror a repo over HTTP


Add Hook

repoName

Webhook

☑ Trigger when repository receives new commits
☑ Trigger when a branch is created or deleted
☑ Trigger when a tag is created or deleted

Hook attributes

https://GitConnector.com/mirrorhooks

json (application/json)

Secret

☑ Insecure ssl

Advanced settings

Save hook   Cancel

1. Select **Hooks**, **Add Hook**, and select a repository from the drop-down.
2. Select the service **WebHook** from the drop-down.
3. Check the triggers that you want to launch a mirroring action.

4. Under **Hook attributes**:
   a. Paste the URL of the Git Connector into the **URL** text box: `https://GitConnector.com/mirrorhooks`
   b. Select **content-type** of **json (application/json)** from the drop-down.
   c. Paste the mirrorhook **secret** token in the **Secret** text box.
   d. Check the **Insecure ssl** checkbox because no certificate is associated with the webhook.
5. Click **Save hook**.
6. Validate that mirroring is in place by running the following command on the Git Connector: `gconn --mirrorhooks list`

   This displays the repositories that are mirrored and the Git Host. For example:

   ```
   gconn --mirrorhooks list
   ```

   ```
   //hth/repoName <<<
   http://HelixTeamHub.com/hth/projects/projectName/repositories/git/rep
   oName.git ...
   ```

   ```
   //hth/repoName2 <<<
   http://HelixTeamHub.com/hth/projects/projectName/repositories/git/rep
   oName2.git ... Not mirrored by this Gconn instance ( no mirror config
   )
   ```

Troubleshooting

If there are any issues, review the following files, or send them to Perforce Technical Support:

Helix TeamHub log at `/var/log/hth/resque/current`

and from the Git Connector:

```
/opt/perforce/git-
connector/repos/graphDepot/repoName.git/.mirror.config
```

```
/opt/perforce/git-connector/repos/graphDepot/repoName.git/push_log
```

```
/opt/perforce/git-connector/repos/graphDepot/repoName.git/fetch_log
```

```
/opt/perforce/git-connector/repos/graphDepot/repoName.git/.mirror.log
```

```
/opt/perforce/git-connector/gconn.conf
```

```
/opt/perforce/git-connector/logs/gconn.log
```

```
/opt/perforce/git-connector/logs/p4gc.log
```

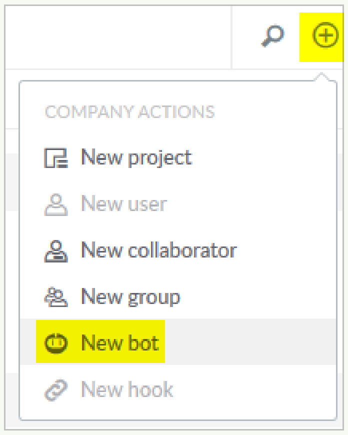
Helix TeamHub SSH

Tip

Use a bot account instead of personal credentials to store the SSH public key required for the GitConnector. (The `web-service-user` mentioned below).

The two options are:

- Use a regular bot, and give it access to relevant projects and repositories on the team view
- Use a company admin bot account, which has access to every repository inside the company



For more information, see https://helixteamhub.cloud/docs/user/bots/

1. On the Git Connector server, log in as the `root` user and use the `su` command to become a *`web-service-user`*.

Ubuntu	CentOS
`su -s /bin/bash - www-data`	`su -s /bin/bash - apache`

2. Create a `.ssh` directory for the *`web-service-user`* user:
   ```
   mkdir /var/www/.ssh
   ```
3. Assign the owner of the directory:
   ```
   chown web-service-user:gconn-auth /var/www/.ssh
   ```
4. Switch to the *`web-service-user`*:
   ```
   su -s /bin/bash - web-service-user
   ```
   and generate the public and private SSH keys for the Git Connector instance:
   ```
   ssh-keygen -t rsa -b 4096 -C web-service-
   user@gitConnector.com
   ```
   then follow the prompts.

5. Locate the public key:

   `/var/www/.ssh/id_rsa.pub`

6. Copy this public key to the Helix TeamHub server and add `/var/www/.ssh/id_rsa.pub` to the user account (*helix-user*) that performs clone and fetch for mirroring.

7. On the Git Connector, as the `web-service-user`, set the environment variable `GCONN_CONFIG` to the absolute path to the `gconn.conf` file:

   ```
   export GCONN_CONFIG=/opt/perforce/git-connector/gconn.conf
   ```

8. Configure the mirror hooks by running the following as the `web-service-user`:

   Important
   The target repo must NOT already exist in Helix Server.

   The source repo must not be empty.

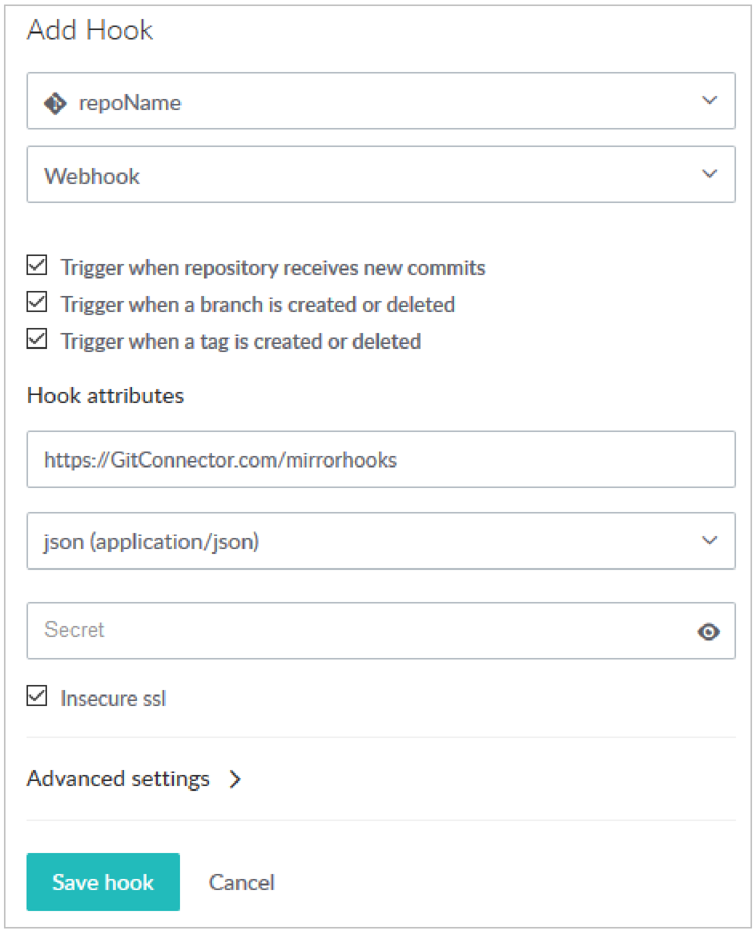
   ```
   ./bin/gconn --mirrorhooks add graphDepotName/repoName
   git@
   HelixTeamHubServer
   /companyName/projects/projectName/repositories/gitrepoName
   ```

   Tip
   Copy the URL from your project's SSH drop-down box.

9. Save the secret token that the `--mirrorhooks` command generates.

   Tip
   The secret token is also stored in `/opt/perforce/git-connector/repos/graphDepotName/repoName.git/.mirror.config`

Mirror a repo over SSH


Add Hook

repoName

Webhook

☑ Trigger when repository receives new commits
☑ Trigger when a branch is created or deleted
☑ Trigger when a tag is created or deleted

Hook attributes

https://GitConnector.com/mirrorhooks

json (application/json)

Secret

☑ Insecure ssl

Advanced settings

Save hook Cancel

1. Select **Hooks**, **Add Hook**, and select a repository from the drop-down.
2. Select service **WebHook** from the drop-down
3. Check the triggers that you want to launch a mirroring action
4. Under Hook attributes:
    a. Paste the URL of the Git Connector into the **URL** text box: `https://GitConnector.com/mirrorhooks`
    b. Select **content-type** of **json (application/json)** from the drop-down.
    c. Paste the mirrorhook **secret** token in the **Secret** text box.
    d. Check the **Insecure ssl** checkbox because no certificate is associated with the webhook.
5. Click **Save hook**.

6. Validate that mirroring is in place by running the following command on the Git Connector:
   `gconn --mirrorhooks list`

   This displays the repositories that are mirrored and the Git Host. For example:

   ```
   gconn --mirrorhooks list
   ```

   ```
   //hth/repoName <<<
   http://HelixTeamHub.com/hth/projects/projectName/repositories/git/rep
   oName.git ...
   ```

   ```
   //hth/repoName2 <<<
   http://HelixTeamHub.com/hth/projects/projectName/repositories/git/rep
   oName2.git ... Not mirrored by this Gconn instance ( no mirror config
   )
   ```

Troubleshooting

If there are any issues, review the following files, or send them to Perforce Technical Support:

Helix TeamHub log at **`/var/log/hth/resque/current`**

and from the Git Connector:

```
/opt/perforce/git-
connector/repos/graphDepot/repoName.git/.mirror.config
```

```
/opt/perforce/git-connector/repos/graphDepot/repoName.git/push_log
```

```
/opt/perforce/git-connector/repos/graphDepot/repoName.git/fetch_log
```

```
/opt/perforce/git-connector/repos/graphDepot/repoName.git/.mirror.log
```

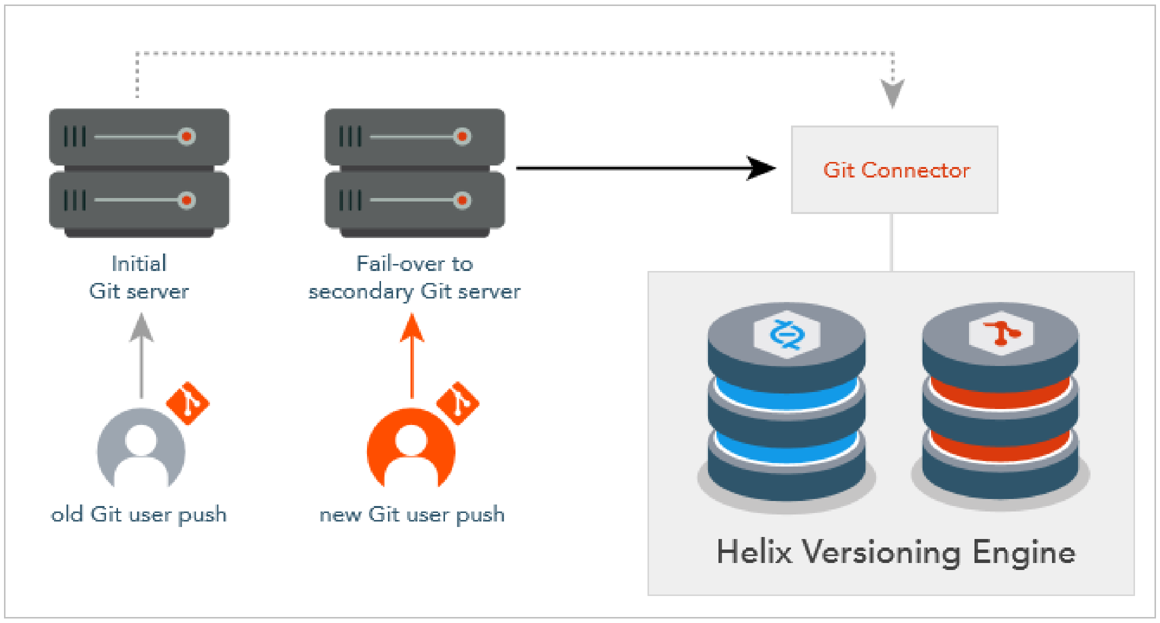
```
/opt/perforce/git-connector/gconn.conf
```

```
/opt/perforce/git-connector/logs/gconn.log
```

```
/opt/perforce/git-connector/logs/p4gc.log
```

Git Connector configuration for fail-over to another Git host

Helix4Git can mirror from a Git server, such as GitLab, GitHub, Gerrit, or Helix TeamHub. If that Git server becomes unavailable, Helix4Git supports the manual configuration of Helix4Git mirroring from the fail-over server.



Note

- The two Git servers should be replicas of each other.
- The Git Connector can run on a machine separate from the Git server and the Helix Versioning Engine (recommended), the same machine as a Git server (also recommended), or the machine with Helix Versioning Engine (not recommended).
- Perforce has tested fail-over with GitLab and Gerrit.

Procedure

To perform a fail-over of the third-party Git server that the Git Connector fetches from, use the Helix Git Connector `setremote` command.

We recommend you first use this command with the -n option:

```
gconn --mirrorhooks -n setremote oldUrl newUrl
```

where

- `oldUrl` is a pattern that matches the primary Git server URL for a set of one or more repos
- `newUrl` is replacement pattern containing the fail-over or secondary server URL, such that all mirrored repos with MirroredFrom URLs matching the `oldUrl` pattern will be modified by substitution

- `-n` displays in preview mode the names of the repos that would be affected, but does not perform the operation

To perform the operation, omit the `-n` option:

```
gconn --mirrorhooks setremote oldUrl newUrl
```

Example

1. Set the environment variable `GCONN_CONFIG` to the absolute path to the `gconn.conf` file:

   ```
   export GCONN_CONFIG=/opt/perforce/git-connector/gconn.conf
   ```

2. Run the `list` command to see the names of repos that are associated with webhooks:

   ```
   gconn --mirrorhooks list
   ```

   The output might be:

   ```
   gconn --mirrorhooks list
   //graphDepot/project1 <<<
   http://
   primaryGitHost/hth/projects/support/repositories/git/project1.git
   //graphDepot/project2 <<<
   http://primaryGitHost/hth/projects/support/repositories/git/project2.
   git  ... Not mirrored by this Gconn instance ( no mirror config )
   //graphDepot/project3 <<<
   http://primaryGitHost/hth/projects/support/repositories/git/project3.
   git ... Not mirrored by this Gconn instance ( no mirror config )
   ```

3. Include the `-n` option to see what the effect would be:

   ```
   gconn -n --mirrorhooks setremote https://primaryGitHost
   https://secondaryGitHost
   ```

   The screen output indicates this is merely a test:

   ```
   This is a report of a trial run. No MirroredFrom urls
   are changed.
   Execute without the '-n' option to update the urls.
   //graphDepot/project1: updating remote url from
   http://primaryGitHost/hth/projects/support/repositor
   ies/git/project1.git
   to
   git@http://secondaryGitHost/hth/projects/support/rep
   ositories/git/project1.git
   ```

4. To run the command that switches to the secondary server, omit the `-n` option:

   ```
   gconn --mirrorhooks setremote https://primaryGitHost
   https://secondaryGitHost
   ```

5. Run the list command again to list the repos that now need to be associated with webhooks: `gconn --mirrorhooks list`

   The output is:

```
gconn --mirrorhooks list
//graphDepot/project1 <<<
http://
secondaryGitHost/hth/projects/support/repositories/git/project1.git
//graphDepot/project2 <<<
http://primaryGitHost/hth/projects/support/repositories/git/project2.
git  ... Not mirrored by this Gconn instance ( no mirror config )
//graphDepot/project3 <<<
http://primaryGitHost/hth/projects/support/repositories/git/project3.
git ... Not mirrored by this Gconn instance ( no mirror config )
```

Effect

The `setremote` command affects both the Git Connector server and the Graph Depot server:

- On the Git Connector server, it reconfigures the `.mirror.config` file to point to the fail-over URL as the `"upstream_url"`
- On the Git Connector server, it reconfigures the upstream fetch URL of repo's cached git repository.
- On the Helix server, it reconfigures the repo spec so that the `MirroredFrom:` field points to the fail-over URL

Command-line Help

To get command-line Help on the setremote command, at your Git Connector command line, type

`gconn --help`

You will see, in addition to the explanations of the commands for `add`, `remove`, and `list`, an explanation of the `setremote` command.

Next Steps

First, configure the fail-over third-party Git server with a web hook for each repo that you want to mirror.

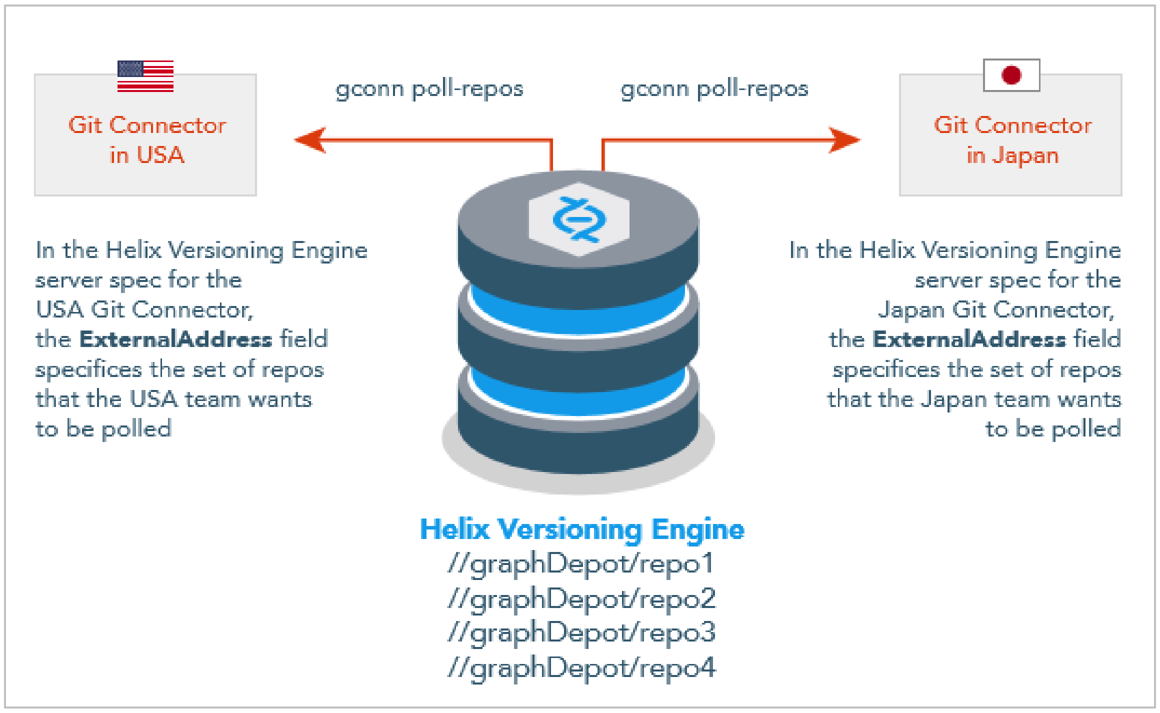
Note: you can reuse the same secret token that is in the repository's `.mirror.config` file. For detailed steps on how to set up the web hook, see the instructions that match your situation:

- "GitHub or GitLab HTTP " on page 38
- "GitHub or GitLab SSH" on page 40
- "Configure Gerrit for HTTP" on page 43
- "Configure Gerrit for SSH" on page 45
- "Helix TeamHub HTTP " on page 50
- "Helix TeamHub SSH" on page 53

Finally, push to the currently active Git server and verify that the webhook causes the Git Connector to fetch the change so that Helix4Git mirrors the change into a repo.

Configuring Git Connector to poll repos from Helix4Git

Your organization might have contributors in multiple locations that are geographically remote from one another, like Brazil and Japan. The administrator of the Git Connector at each location might want the local Git Connector to periodically get the latest version of a set of repos. This can enable the end-users for a given location to experience fast clones and fetches.


Git Connector in USA
gconn poll-repos
gconn poll-repos
Git Connector in Japan
In the Helix Versioning Engine server spec for the USA Git Connector, the **ExternalAddress** field specifices the set of repos that the USA team wants to be polled
In the Helix Versioning Engine server spec for the Japan Git Connector, the **ExternalAddress** field specifices the set of repos that the Japan team wants to be polled
Helix Versioning Engine
//graphDepot/repo1
//graphDepot/repo2
//graphDepot/repo3
//graphDepot/repo4

Procedure

1. The administrator in Brazil uses the `p4 server` command to edit the server specification that corresponds to the `ServerId` for the instance of the Git Connector in Brazil. This administrator populates the `ExernalAddress:` field in the server spec to contain a list of repos for which Brazil wants the latest updates. The list can be comma-separated or space-separated, and can be a subset of the repos:

   ```
   ExternalAddress: //graphDepotName/repo1
   //graphDepotName/repo3 //graphDepotName/repo4
   ```

2. At the Git Connector command-line, the Brazil administrator:
   a. Sets the environment variable `GCONN_CONFIG` to the absolute path to the `gconn.conf` file:

      ```
      export GCONN_CONFIG=/opt/perforce/git-connector/gconn.conf
      ```

   b. Runs the command `gconn poll-repos` and verifies that this manual test has pulled the latest for the set of repos:

Command-Line Output	Meaning
`Polling repo: //graphDepot/repo1` `From p4gc://brazilURL/graphDepoA/repo1` `* [new branch]      master     -> master`	Brazil gets a new branch for this repo
`Polling repo: //graphDepot/repo3` `From p4gc://brazilURL/graphDepoA/repo3` `784a8e8..e6a5604  master     -> master`	Brazil gets an update for this repo
`Polling repo: //graphDepot/repo4`	Brazil is already has the latest for this repo

3. The Brazil administrator configures the UNIX **cron** utility to schedule an automatic run of the `gconn poll-repos` command at a specified interval. For example, `etc/cron.d/gconn` can poll for updated repos every 10 minutes:

   ```
   */10 * * * * git /usr/bin/gconn poll-repos
   ```

Note
The administrator for Japan can edit the server spec associated with the Japan Git Connector such that this server spec for Japan contains none, some, or all of the repos as the server spec for Brazil. Similarly, the administrator for Japan might set up a different interval for polling.

Troubleshooting

The following sections indicate problems you might encounter, how to fix them, and how to facilitate troubleshooting with "Special Git commands" on page 73.

Connection problems

This section lists problems related to accessing graph depots or repos.

SSH: user prompted for git's password

Problem	Solution

`git clone git@ConnectorHost/gD1/repo8` causes the user to be prompted for git's password: Cloning into 'repo8'… `git@ConnectorHost's password:`	Try one or more of the following:
	1: Run **`p4 protect`** to open the spec form, and add the **`gconn-user`** to the protections table with the **`write`** permission:
	`write user gconn-user * //...`
	See p4 protect in *P4 Command Reference*.
	
	2: Run **`p4 show-permission`** to find out whether the **`gconn-user`** has **`admin`** permission.
	`p4 show-permission -u gconn-user -d gD1`
	If not, run **`p4 grant-permission`** to grant **`admin`** access to the **`gconn-user`**.
	`p4 grant-permission -p admin -d gD1 -u gconn-user * //...`
	See p4 grant-permission in *P4 Command Reference*.
	
	3: Add the user's SSH public key to the Git Connector:
	`p4 pubkey -i -u user < id_rsa.pub`
	and wait ten minutes for the Git Connector to update the Helix Server.
	See p4 pubkey in *P4 Command Reference*.

SSL certificate problem

Problem	Solution
`git clone https://ConnectorHost/gD1/repo8` results in `Cloning into 'gD1/repo8'... fatal: unable to access https://ConnectorHost/gD1/repo8/': SSL certificate problem: Invalid certificate chain`	Turn off SSL validation: `git config --global http.sslVerify false`

HTTPS: user does not exist

Problem	Solution
`git clone https://ConnectorHost/gD1/repo8` results in `Cloning into 'gD1/repo8'...` `Username for https://ConnectorHost: bruno` `Password for https://bruno@ConnectorHost:` `remote: User is not authenticated: User bruno doesn't exist.` `fatal: Authentication failed for https://ConnectorHost/gD1/repo8/.`	Create the missing user by running **`p4 user`**. See p4 user in *P4 Command Reference*.

Permission problems

This sections lists permission-related problems.

The gconn-user needs admin access

Problem	Solution

If
`git push origin master`
results in
`... GConn P4 user needs admin access ...`
	As a superuser, run **`p4 protect`** to open the spec form, then add the **`gconn-user`** to the protections table with the **`write`** permission:
	`write user gconn-user * //gD1/...`
	See p4 protect in *P4 Command Reference*.
	and,
	Run **`p4 show-permission`** to find out whether the **`gconn-user`** has **`admin`** permission.
	`p4 show-permission -u gconn-user -d gD1`
	If not, run **`p4 grant-permission`** to grant **`admin`** access to the **`gconn-user`** for the specified depot.
	See p4 grant-permission in *P4 Command Reference*.

Unable to clone: missing read permission

Problem	Solution

`git clone https://bruno@ConnectorHost/gD1/repo8`
results in:
`No read permission`
	Grant the **`read`** permission:
	`p4 grant-permission -u bruno -p read -d gD1`
	See p4 grant-permission in *P4 Command Reference*.

Unable to push: missing create-repo permission

Problem	Solution

`git push git@`*`ConnectorHost`*`:gD1/repo8 master`
results in
`! [remote rejected] 8cf...b4d -> master (User bruno does not have administrative privileges to create repo //gD1/repo8.)`
	Grant the permission to create a repo:
	`p4 grant-permission -u bruno -p create-repo -d gD1`
	See p4 grant-permission in *P4 Command Reference*.

Unable to push: missing write-ref permission

Problem	Solution

`git push origin master`
results in
`... User bruno does not have write-ref privilege for reference refs/heads/master.`
	Grant the `write-ref` permission:
	`p4 grant-permission -u bruno -p write-ref -d gD1`
	You can specify an entire depot or repo, or limit the user to one or more branches or tags. See p4 grant-permission in *P4 Command Reference*.
	Note
	A user with the `write-ref` permission also needs p4 protect `write` access.

Unable to push: not enabled by p4 protect

Problem	Solution

If
`git push origin master`
results in
`... Access for user 'bruno' has not been enabled by 'p4 protect'...`
	Note
	A user with the `write-ref` permission also needs p4 protect `write` access.
	
	The `write-ref` permission is the sole permission that applies the protection setting in the protections table for a file or directory. As a superuser, run `p4 protect` to open the spec form, then add the user to the protections table with the `write` permission:
	`write user bruno * //gD1/...`
	See p4 protect in *P4 Command Reference*.

Unable to push a new branch: missing create-ref permission

Problem	Solution

`git push origin dev`
results in
`! [remote rejected] 8cf...b4d -> master (User bruno does not have create-ref privilege for reference refs/heads/dev.)`
	Grant the permission to create a reference in the graph depot.
	`p4 grant-permission -u bruno -p create-ref -d gD1`
	See p4 grant-permission in *P4 Command Reference*.

Unable to delete a branch: missing delete-ref permission

Problem	Solution

`git push origin :dev`
results in
`remote:  ! [remote rejected] dev (User bruno does not have delete-ref privilege for reference refs/heads/dev.)`
	Grant the permission to delete a repo in the graph depot:
	`p4 grant-permission -u bruno -p delete-ref -d gD1`
	See p4 grant-permission in *P4 Command Reference*.

Unable to force a push: missing force-push permission

Problem	Solution

Some organizations allow one or more special users or administrators to overwrite other people's work by granting this user the `force-push` permission. The `force-push` permission implies the powers associated with the following permissions: `read`, `write-ref`, `write-all`, `create-ref` and `delete-ref`.
 	
If the user does not have the `force-push` permission,
 	
`git push --force origin master`
 	
results in
 	
`remote:  ! [remote rejected]`
`d59...2bf - master`
`(User bruno does not have force-push privilege for`
`reference`
`refs/heads/master.)`
	Grant the `force-push` permission to the special user.
	
	`p4 grant-permission -u bruno -p force-push -d gD1`
	
	See p4 grant-permission in *P4 Command Reference*.

Branch problems

This section lists problems related to branches.

Push results in message about HEAD ref not existing

Running the following command:

```
$ git push git@ConnectorHost:gD1/repo8 main
```

results in:

```
Counting objects: 3, done.
Writing objects: 100% (3/3), 226 bytes \| 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
remote: HEAD ref is "refs/heads/master", but this ref does not exist.
remote: Consider asking the admin for repo "gD1/repo8.git"
remote: to set its default branch to a valid ref so that
remote: "git clone" and "git checkout" can check out
remote: without specifying a branch name.
To git@xx.x.xx.xxx:repo/grepo1
 * [new branch]      main -> main
```

To resolve this issue, do one of the following:

- Edit the repo spec to specify `refs/heads/main` as the default branch to clone from. This is required for any project not using the `refs/heads/master` default branch. For details, see "Specify a default branch" on page 32.
- Run the following special command to set the default branch to `refs/heads/main`:

  ```
  $ git clone
  git@ConnectorHost:@defaultbranch:gD1/repo8=refs/heads/main
  ```

  This results in the following output:

  git clone git@ConnectorHost:@defaultbranch:gD1/repo8=refs/heads/main
  Cloning into 'main'...
  repo='gD1/repo8', old DefaultBranch='', new DefaultBranch='refs/heads/main'
  fatal: Could not read from remote repository.
  Please make sure you have the correct access rights and the repository exists.

  Note
  Because the special command is not standard Git syntax, Git cannot parse it and the command terminates with:

  ```
  Fatal: Could not read from remote repository.
  ```

  You can also run `@defaultbranch:gD1/repo8` to show the default branch and `@defaultbranch:gD1/repo8=` to clear the default branch.

Clone results in "remote HEAD refers to nonexistent ref"

Running the following command:

```
$ git clone git@ConnectorHost:gD1/repo8
```

results in:

```
Cloning into 'repo8'...
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.
Checking connectivity... done.
warning: remote HEAD refers to nonexistent ref, unable to checkout.
```

To resolve this issue, do one of the following:

- Edit the repo spec to specify `refs/heads/main` as the default branch to clone from. This is required for any repo not using the `refs/heads/master` default branch. For details, see "Specify a default branch" on page 32.
- Run the following special command to set the default branch to `refs/heads/main`:

  ```
  $ git clone
  git@ConnectorHost:@defaultbranch:gD1/repo8=refs/heads/main
  ```

  This results in the following output:

  ```
  git clone git@ConnectorHost:@defaultbranch:gD1/repo8=refs/heads/main
  Cloning into 'repo8=refs/heads/main'...
  repo='gD1/repo8', old DefaultBranch='', new DefaultBranch='refs/heads/main'
  fatal: Could not read from remote repository.
  ```

  Note
  The special command sets the default branch even if Git cannot parse it and the commands terminates with:

  ```
  Fatal: Could not read from remote repository.
  ```

  You can also run `@defaultbranch:gD1/repo8` to show the default branch and `@defaultbranch:gD1/repo8=` to clear the default branch.

Special Git commands

On a Git client, you can run special commands that extend Git command functionality. Each special command begins with `git clone`. Special commands work with SSH or HTTPS authentication, and here we show SSH:

- `git clone git@ConnectorHost:@help`: Shows Git Connector special command help.
- `git clone git@ConnectorHost:@info`: Shows Git Connector version information.

- **`git clone git@ConnectorHost:@list`**: Lists repositories available to you, based on permissions.
- **`git clone git@ConnectorHost:@defaultbranch:graphDepot/repo`**: Shows the default branch set for the repo.
- **`git clone git@ConnectorHost:@defaultbranch:graphDepot/repo=`**: Clears the default branch set for the repo.
- **`git clone git@ConnectorHost:@defaultbranch:graphDepot/repo=branch`**: Sets the default branch.

For example,

```
$ git clone git@ConnectorHost:@info
```

Results in the following output:

```
git clone git@connector.com:@info
Cloning into '@info'...
Perforce - The Fast Software Configuration Management System.
Copyright 1995-2016 Perforce Software.  All rights reserved.
This product includes software developed by the OpenSSL Project
for use in the OpenSSL Toolkit (http://www.openssl.org/)
See 'p4 help legal' for full OpenSSL license information
Version of OpenSSL Libraries: OpenSSL 1.0.2j  26 Sep 2016
Rev. GCONN/LINUX26X86_64/2016.2.MAIN-TEST_ONLY/1460278 (2016/11/03).
uname: Linux gconn-centos6 2.6.32-504.el6.x86_64 #1 SMP Wed Oct 15
04:27:16 UTC 2014 x86_64
P4 Info:
    caseHandling: sensitive
    clientAddress: xx.x.xx.xxx
    clientCase: sensitive
    clientCwd: /home/git
    clientHost: gconn-centos6
    clientName: unknown
    password: enabled
    peerAddress: xx.x.xx.xxx:47041
    serverAddress: xx.x.xx.xxx:16200
    serverDate: 2016/11/07 14:13:41 -0800 PST
    serverLicense: none
    serverRoot: /opt/perforce/servers/16200
    serverServices: standard
    serverUptime: 76:01:42
    serverVersion: P4D/LINUX26X86_64/2017.1.MAIN-TEST_ONLY/1460278
(2016/11/03)
    tzoffset: -28800
    userName: gconn-user
fatal: Could not read from remote repository.
```

Note
Because the special command is not standard Git syntax, Git cannot parse it, so the command terminates with:

```
Fatal: Could not read from remote repository.
```

Glossary

A

access level

A permission assigned to a user to control which commands the user can execute. See also the 'protections' entry in this glossary and the 'p4 protect' command in the P4 Command Reference.

admin access

An access level that gives the user permission to privileged commands, usually super privileges.

archive

1. For replication, versioned files (as opposed to database metadata). 2. For the 'p4 archive' command, a special depot in which to copy the server data (ersioned files and metadata).

atomic change transaction

Grouping operations affecting a number of files in a single transaction. If all operations in the transaction succeed, all the files are updated. If any operation in the transaction fails, none of the files are updated.

B

base

The file revision, in conjunction with the source revision, used to help determine what integration changes should be applied to the target revision.

binary file type

A Helix Server file type assigned to a non-text file. By default, the contents of each revision are stored in full, and file revision is stored in compressed format.

branch

(noun) A set of related files that exist at a specific location in the Perforce depot as a result of being copied to that location, as opposed to being added to that location. A group of related files is often referred to as a codeline. (verb) To create a codeline by copying another codeline with the 'p4 integrate', 'p4 copy', or 'p4 populate' command.

branch form

The form that appears when you use the 'p4 branch' command to create or modify a branch specification.

branch mapping

Specifies how a branch is to be created or integrated by defining the location, the files, and the exclusions of the original codeline and the target codeline. The branch mapping is used by the integration process to create and update branches.

branch view

A specification of the branching relationship between two codelines in the depot. Each branch view has a unique name and defines how files are mapped from the originating codeline to the target codeline. This is the same as branch mapping.

broker

Helix Broker, a server process that intercepts commands to the Helix Server and is able to run scripts on the commands before sending them to the Helix Server.

C

change review

The process of sending email to users who have registered their interest in changelists that include specified files in the depot.

changelist

A list of files, their version numbers, the changes made to the files, and a description of the changes made. A changelist is the basic unit of versioned work in Helix Server. The changes specified in the changelist are not stored in the depot until the changelist is submitted to the depot. See also atomic change transaction.

changelist form

The form that appears when you modify a changelist using the 'p4 change' command.

changelist number

The unique numeric identifier of a changelist. By default, changelists are sequential.

check in

To submit a file to the Helix Server depot.

check out

To designate one or more files for edit.

checkpoint

A backup copy of the underlying metadata at a particular moment in time. A checkpoint can recreate db.user, db.protect, and other db.* files. See also metadata.

classic depot

A repository of Helix Server files that is not streams-based. The default depot name is depot. See also default depot and stream depot.

client form

The form you use to define a client workspace, such as with the 'p4 client' or 'p4 workspace' commands.

client name

A name that uniquely identifies the current client workspace. Client workspaces, labels, and branch specifications cannot share the same name.

client root

The topmost (root) directory of a client workspace. If two or more client workspaces are located on one machine, they should not share a client root directory.

client side

The right-hand side of a mapping within a client view, specifying where the corresponding depot files are located in the client workspace.

client workspace

Directories on your machine where you work on file revisions that are managed by Helix Server. By default, this name is set to the name of the machine on which your client workspace is located, but it can be overridden. Client workspaces, labels, and branch specifications cannot share the same name.

code review

A process in Helix Swarm by which other developers can see your code, provide feedback, and approve or reject your changes.

codeline

A set of files that evolve collectively. One codeline can be branched from another, allowing each set of files to evolve separately.

comment

Feedback provided in Helix Swarm on a changelist or a file within a change.

commit server

A server that is part of an edge/commit system that processes submitted files (checkins), global workspaces, and promoted shelves.

conflict

1. A situation where two users open the same file for edit. One user submits the file, after which the other user cannot submit unless the file is resolved. 2. A resolve where the same line is changed when merging one file into another. This type of conflict occurs when the comparison of two files to a base yields different results, indicating that the files have been changed in different ways. In this case, the merge cannot be done automatically and must be resolved manually. See file conflict.

copy up

A Helix Server best practice to copy (and not merge) changes from less stable lines to more stable lines. See also merge.

counter

A numeric variable used to track variables such as changelists, checkpoints, and reviews.

D

default changelist

The changelist used by a file add, edit, or delete, unless a numbered changelist is specified. A default pending changelist is created automatically when a file is opened for edit.

deleted file

In Helix Server, a file with its head revision marked as deleted. Older revisions of the file are still available. in Helix Server, a deleted file is simply another revision of the file.

delta

The differences between two files.

depot

A file repository hosted on the server. A depot is the top-level unit of storage for versioned files (depot files or source files) within a Helix Versioning Engine. It contains all versions of all files ever submitted to the depot. There can be multiple depots on a single installation.

depot root

The topmost (root) directory for a depot.

depot side

The left side of any client view mapping, specifying the location of files in a depot.

depot syntax

Helix Server syntax for specifying the location of files in the depot. Depot syntax begins with: //depot/

diff

(noun) A set of lines that do not match when two files are compared. A conflict is a pair of unequal diffs between each of two files and a base. (verb) To compare the contents of files or file revisions. See also conflict.

donor file

The file from which changes are taken when propagating changes from one file to another.

E

edge server

A replica server that is part of an edge/commit system that is able to process most read/write commands, including 'p4 integrate', and also deliver versioned files (depot files).

exclusionary access

A permission that denies access to the specified files.

exclusionary mapping

A view mapping that excludes specific files or directories.

F

file conflict

In a three-way file merge, a situation in which two revisions of a file differ from each other and from their base file. Also, an attempt to submit a file that is not an edit of the head revision of the file in the depot, which typically occurs when another user opens the file for edit after you have opened the file for edit.

file pattern

Helix Server command line syntax that enables you to specify files using wildcards.

file repository

The master copy of all files, which is shared by all users. In Helix Server, this is called the depot.

file revision

A specific version of a file within the depot. Each revision is assigned a number, in sequence. Any revision can be accessed in the depot by its revision number, preceded by a pound sign (#), for example testfile#3.

file tree

All the subdirectories and files under a given root directory.

file type

An attribute that determines how Helix Server stores and diffs a particular file. Examples of file types are text and binary.

fix

A job that has been closed in a changelist.

form

A screen displayed by certain Helix Server commands. For example, you use the change form to enter comments about a particular changelist to verify the affected files.

forwarding replica

A replica server that can process read-only commands and deliver versioned files (depot files). One or more replicat servers can significantly improve performance by offloading some of the master server load. In many cases, a forwarding replica can become a disaster recovery server.

G

Git Fusion

A Perforce product that integrates Git with Helix, offering enterprise-ready Git repository management, and workflows that allow Git and Helix Server users to collaborate on the same projects using their preferred tools.

graph depot

A depot of type graph that is used to store Git repos in the Helix Server. See also Helix4Git.

group

A feature in Helix Server that makes it easier to manage permissions for multiple users.

H

have list

The list of file revisions currently in the client workspace.

head revision

The most recent revision of a file within the depot. Because file revisions are numbered sequentially, this revision is the highest-numbered revision of that file.

Helix Server

The Helix Server depot and metadata; also, the program that manages the depot and metadata, also called Helix Versioning Engine.

Helix TeamHub

A Perforce management platform for code and artifact repository. TeamHub offers built-in support for Git, SVN, Mercurial, Maven, and more.

Helix4Git

Perforce solution for teams using Git. Helix4Git offers both speed and scalability and supports hybrid environments consisting of Git repositories and 'classic' Helix Server depots.

I

integrate

To compare two sets of files (for example, two codeline branches) and determine which changes in one set apply to the other, determine if the changes have already been propagated, and propagate any outstanding changes from one set to another.

J

job

A user-defined unit of work tracked by Helix Server. The job template determines what information is tracked. The template can be modified by the Helix Server system administrator. A job describes work to be done, such as a bug fix. Associating a job with a changelist records which changes fixed the bug.

job specification

A form describing the fields and possible values for each job stored in the Helix Server machine.

job view

A syntax used for searching Helix Server jobs.

journal

A file containing a record of every change made to the Helix Server’s metadata since the time of the last checkpoint. This file grows as each Helix Server transaction is logged. The file should be automatically truncated and renamed intoa numbered journal when a checkpoint is taken.

journal rotation

The process of renaming the current journal to a numbered journal file.

journaling

The process of recording changes made to the Helix Server’s metadata.

L

label

A named list of user-specified file revisions.

label view

The view that specifies which filenames in the depot can be stored in a particular label.

lazy copy

A method used by Helix Server to make internal copies of files without duplicating file content in the depot. A lazy copy points to the original versioned file (depot file). Lazy copies minimize the consumption of disk space by storing references to the original file instead of copies of the file.

license file

A file that ensures that the number of Helix Server users on your site does not exceed the number for which you have paid.

list access

A protection level that enables you to run reporting commands but prevents access to the contents of files.

local depot

Any depot located on the currently specified Helix Server.

local syntax

The syntax for specifying a filename that is specific to an operating system.

lock

1. A file lock that prevents other clients from submitting the locked file. Files are unlocked with the 'p4 unlock' command or by submitting the changelist that contains the locked file. 2. A database lock that prevents another process from modifying the database db.* file.

log

Error output from the Helix Server. To specify a log file, set the P4LOG environment variable or use the p4d -L flag when starting the service.

M

mapping

A single line in a view, consisting of a left side and a right side that specify the correspondences between files in the depot and files in a client, label, or branch. See also workspace view, branch view, and label view.

MDS checksum

The method used by Helix Server to verify the integrity of versioned files (depot files).

merge

1. To create new files from existing files, preserving their ancestry (branching). 2. To propagate changes from one set of files to another. 3. The process of combining the contents of two conflicting file revisions into a single file, typically using a merge tool like P4Merge.

merge file

A file generated by the Helix Server from two conflicting file revisions.

metadata

The data stored by the Helix Server that describes the files in the depot, the current state of client workspaces, protections, users, labels, and branches. Metadata includes all the data stored in the Perforce service except for the actual contents of the files.

modification time or modtime

The time a file was last changed.

N

nonexistent revision

A completely empty revision of any file. Syncing to a nonexistent revision of a file removes it from your workspace. An empty file revision created by deleting a file and the #none revision specifier are examples of nonexistent file revisions.

numbered changelist

A pending changelist to which Helix Server has assigned a number.

O

opened file

A file that you are changing in your client workspace that is checked out. If the file is not checked out, opening it in the file system does not mean anything to the versioning engineer.

owner

The Helix Server user who created a particular client, branch, or label.

P

p4

1. The Helix Versioning Engine command line program. 2. The command you issue to execute commands from the operating system command line.

p4d

The program that runs the Helix Server; p4d manages depot files and metadata.

pending changelist

A changelist that has not been submitted.

project

In Helix Swarm, a group of Helix Server users who are working together on a specific codebase, defined by one or more branches of code, along with options for a job filter, automated test integration, and automated deployment.

protections

The permissions stored in the Helix Server's protections table.

proxy server

A Helix Server that stores versioned files. A proxy server does not perform any commands. It serves versioned files to Helix Server clients.

R

RCS format

Revision Control System format. Used for storing revisions of text files in versioned files (depot files). RCS format uses reverse delta encoding for file storage. Helix Server uses RCS format to store text files. See also reverse delta storage.

read access

A protection level that enables you to read the contents of files managed by Helix Server but not make any changes.

remote depot

A depot located on another Helix Server accessed by the current Helix Server.

replica

A Helix Server that contains a full or partial copy of metadata from a master Helix Server. Replica servers are typically updated every second to stay synchronized with the master server.

repo

A graph depot contains one or more repos, and each repo contains files from Git users.

reresolve

The process of resolving a file after the file is resolved and before it is submitted.

resolve

The process you use to manage the differences between two revisions of a file. You can choose to resolve conflicts by selecting the source or target file to be submitted, by merging the contents of conflicting files, or by making additional changes.

reverse delta storage

The method that Helix Server uses to store revisions of text files. Helix Server stores the changes between each revision and its previous revision, plus the full text of the head revision.

revert

To discard the changes you have made to a file in the client workspace before a submit.

review access

A special protections level that includes read and list accesses and grants permission to run the p4 review command.

revision number

A number indicating which revision of the file is being referred to, typically designated with a pound sign (#).

revision range

A range of revision numbers for a specified file, specified as the low and high end of the range. For example, myfile#5,7 specifies revisions 5 through 7 of myfile.

revision specification

A suffix to a filename that specifies a particular revision of that file. Revision specifiers can be revision numbers, a revision range, change numbers, label names, date/time specifications, or client names.

S

server data

The combination of server metadata (the Helix Server database) and the depot files (your organization's versioned source code and binary assets).

server root

The topmost directory in which p4d stores its metadata (db.* files) and all versioned files (depot files or source files). To specify the server root, set the P4ROOT environment variable or use the p4d -r flag.

service

In the Helix Versioning Engine, the shared versioning service that responds to requests from Helix Server client applications. The Helix Server (p4d) maintains depot files and metadata describing the files and also tracks the state of client workspaces.

shelve

The process of temporarily storing files in the Helix Server without checking in a changelist.

status

For a changelist, a value that indicates whether the changelist is new, pending, or submitted. For a job, a value that indicates whether the job is open, closed, or suspended. You can customize job statuses. For the 'p4 status' command, by default the files opened and the files that need to be reconciled.

stream

A branch with additional intelligence that determines what changes should be propagated and in what order they should be propagated.

stream depot

A depot used with streams and stream clients.

submit

To send a pending changelist into the Helix Server depot for processing.

super access

An access level that gives the user permission to run every Helix Server command, including commands that set protections, install triggers, or shut down the service for maintenance.

symlink file type

A Helix Server file type assigned to symbolic links. On platforms that do not support symbolic links, symlink files appear as small text files.

sync

To copy a file revision (or set of file revisions) from the Helix Server depot to a client workspace.

T

target file

The file that receives the changes from the donor file when you integrate changes between two codelines.

text file type

Helix Server file type assigned to a file that contains only ASCII text, including Unicode text. See also binary file type.

theirs

The revision in the depot with which the client file (your file) is merged when you resolve a file conflict. When you are working with branched files, theirs is the donor file.

three-way merge

The process of combining three file revisions. During a three-way merge, you can identify where conflicting changes have occurred and specify how you want to resolve the conflicts.

trigger

A script automatically invoked by Helix Server when various conditions are met. (See "Helix Versioning Engine Administrator Guide: Fundamentals" on "Using triggers to customize behavior")

two-way merge

The process of combining two file revisions. In a two-way merge, you can see differences between the files.

typemap

A table in Helix Server in which you assign file types to files.

U

user

The identifier that Helix Server uses to determine who is performing an operation.

V

versioned file

Source files stored in the Helix Server depot, including one or more revisions. Also known as a depot file or source file. Versioned files typically use the naming convention 'filenamev' or '1.changelist.gz'.

view

A description of the relationship between two sets of files. See workspace view, label view, branch view.

W

wildcard

A special character used to match other characters in strings. The following wildcards are available in Helix Server: * matches anything except a slash; ... matches anything including slashes; %%0 through %%9 is used for parameter substitution in views.

workspace

See client workspace.

workspace view

A set of mappings that specifies the correspondence between file locations in the depot and the client workspace.

write access

A protection level that enables you to run commands that alter the contents of files in the depot. Write access includes read and list accesses.

Y

yours

The edited version of a file in your client workspace when you resolve a file. Also, the target file when you integrate a branched file.

License Statements