Helix4Git

Helix4Git Administration

2017.1 patch

July 2017

PERFORCE

www.perforce.com

Contents

About this Manual

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

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 depot data
- Supports "One-way mirroring from third-party Git servers" on page 30, 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

Helix4Git consists of two components:

- Helix server (also known as p4d), the traditional Helix Versioning Engine augmented for Git support
- The Git Connector, which acts as a Git server to Git clients

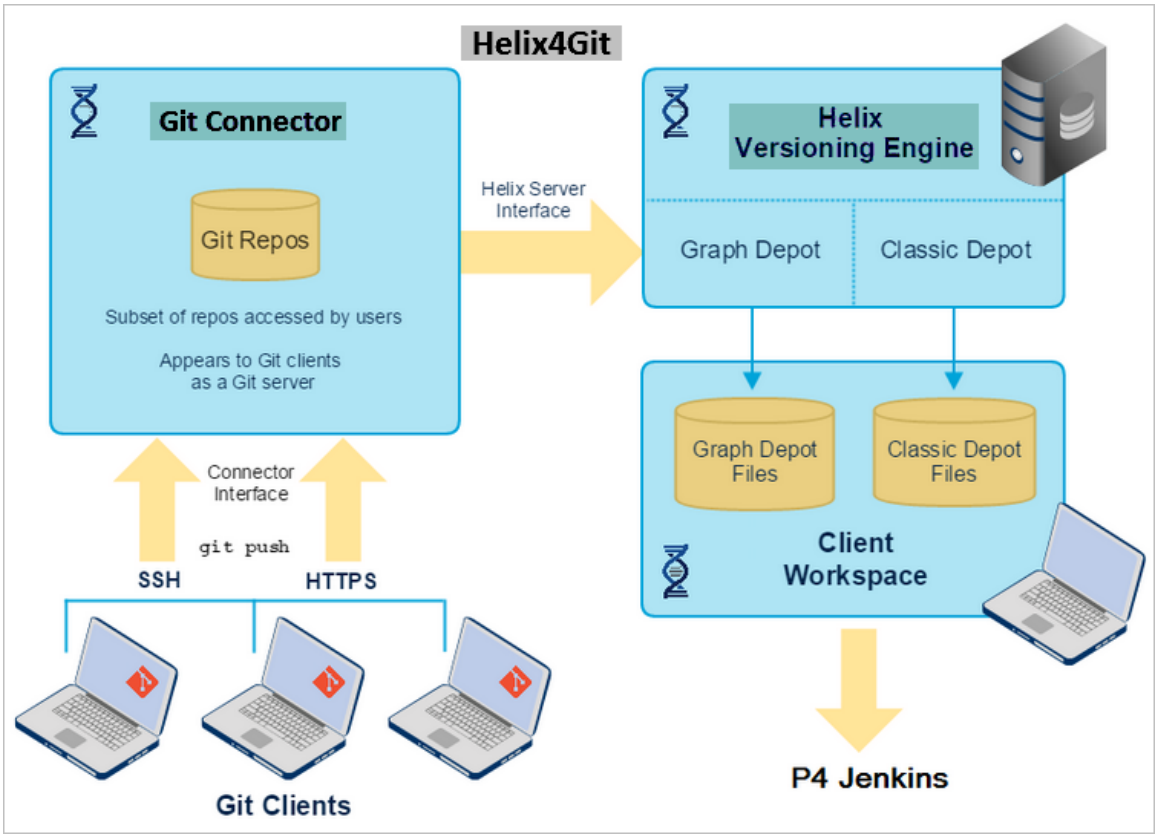
Git users use a Git client to pull files from the graph depot to make modifications and then push the changes back into the graph depot. The Git client communicates with the Helix server through the Git Connector.

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 Helix command line client into a single classic Helix workspace.

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

The following steps describe 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 have classic Helix 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 Helix files.

One-time tasks

The following table summarizes one-time tasks:

Task	More information
Install the Git Connector.	"Install the Git Connector" on page 12
Configure the Git Connector. This includes configuring HTTPS and SSH authentication.	"Configure the Git Connector" on page 14
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 16
Set up users.	"Set up Git users to work with the Git Connector" on page 19
Verify the Git Connectorconfiguration.	"Verify the Git Connector configuration" on page 21

Recurring tasks

The following table summarizes recurring tasks:

Task	More information
Create and view graph depots.	"Create graph depots" on page 23
Create, view, and delete Git repos.	"Create and view repos" on page 24
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 25
Set up client workspaces.	"Set up client workspaces" on page 26
Run p4 sync and a subset of other p4 commands against both Git and classic Helix data.	"Sync files from graph depots" on page 27
Troubleshoot.	"Troubleshooting" on page 41

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 19.
- 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 Connectorrequires an installation of the 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 Connectoris 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 third-party 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 third-party Git servers" on page 30is 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.

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 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 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 helix-git-connector" below and "Configure the Git Connector" on the next page.

Upgrading helix-git-connector

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 Perforce configuration has changed, or your 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 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 super-user password:** The password for the existing Helix 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 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 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.
- **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 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 19) 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 25.

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 23.

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 24.

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 26.

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
Any Helix server user can add an SSH key to the Helix server as long as the user has been added to the protection table. For more information, 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. To add the key to the Helix server machine, run the following command:

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

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

   ```
   $ 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 41.

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 19). 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 (hthe 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 third-party Git servers" on page 30.

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, as follows. This creates a repo with only that branch.

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

- all branches by passing in the `--all` option, as follows. This creates a repo with all branches.

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

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 25). 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 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 graph depot in *P4 Command Reference*.

Sync using an automatic label

Helix'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). In non-Helix 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 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 `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 third-party Git servers

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

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

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 third-party 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 third-party Git server, configure a webhook in the third-party Git server and the Git Connector server, which enables this flow:

1. A Git user pushes a branch to a third-party Git server.
2. The external repo in the third-party 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 third-party 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 " on the next page

"GitHub or GitLab SSH" on page 32

GitHub or GitLab HTTP

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.
   >
   > The source repo must not be empty.

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

Mirror a repo over HTTP

1. Go to the `https://GitHost.com/project/repo/hooks` URL.
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 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.

   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 helix-git-connector" on page 13.
- Gerrit version 2.13 or 2.14 installed and working on the third-party 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.

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.

   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 39

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.

   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 next 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
```

Troubleshooting

The following sections indicate problems you might encounter and how to fix them.

Connection problems:

- "SSH: user prompted for git's password" on the next page
- "SSL certificate problem" on page 43
- "HTTPS: user does not exist" on page 43

Permissions problems:

- "The gconn-user needs admin access" on page 44
- "Unable to clone: missing read permission" on page 44
- "Unable to push: missing create-repo permission" on page 45
- "Unable to push: missing write-ref permission" on page 45
- "Unable to push: not enabled by p4 protect" on page 46
- "Unable to push a new branch: missing create-ref permission" on page 46
- "Unable to delete a branch: missing delete-ref permission" on page 47
- "Unable to force a push: missing force-push permission" on page 48

Branch problems:

- "Push results in message about HEAD ref not existing" on page 48
- "Clone results in "remote HEAD refers to nonexistent ref"" on page 49

To facilitate troubleshooting: "Special Git commands" on page 50.

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 25.
- 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 25.
- 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.
```

License Statements

Perforce software includes software developed by the University of California, Berkeley and its contributors. This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/).

Perforce software includes software from the Apache ZooKeeper project, developed by the Apache Software Foundation and its contributors. (http://zookeeper.apache.org/)

Perforce software includes software developed by the OpenLDAP Foundation (http://www.openldap.org/).

Perforce software includes software developed Computing Services at Carnegie Mellon University: Cyrus SASL (http://www.cmu.edu/computing/).