PERFORCE

Helix4Git

2017.1

April 2017

Helix4Git
2017.1

April 2017

Table of Contents

Preface

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 both traditional Helix clients and Git clients, and stores Git data in Git repos that reside within a Helix depot.

Helix documentation

The following table lists and describes key documents for Helix users, developers, and administrators. For complete information see the following:

http://www.perforce.com/documentation

For specific information about…	See this documentation…
Overview of Helix version control concepts and workflows; Helix architecture, and related products.	*Helix Versioning: An Overview*
Administering the Helix versioning engine to support git repos and LFS files.	*Helix4Git*
Using the command-line interface to perform software version management and codeline management; working with Helix streams; jobs, reporting, scripting, and more.	*Helix Versioning Engine User Guide*
Basic workflows using P4V, the cross-platform Helix desktop client.	*P4V User Guide*
Working with personal and shared servers and understanding the distributed versioning features of the Helix Versioning engine.	*Using Helix for Distributed Versioning*
`p4` command line (reference).	*P4 Command Reference*, **`p4 help`**
Installing and administering the Helix versioning engine, including user management, security settings.	*Helix Versioning Engine Administrator Guide: Fundamentals*
Installing and configuring Helix servers (proxies, replicas, and edge servers) in a distributed environment.	*Helix Versioning Engine Administrator Guide: Multi-site Deployment*
Helix plug-ins and integrations.	IDEs: *Using IDE Plug-ins* Defect trackers: *Defect Tracking Gateway Guide* Others: online help from the Helix menu or web site

For specific information about…	See this documentation…
Developing custom Helix applications using the Helix C/C++ API.	*C/C++ API User Guide*
Working with Helix in Ruby, Perl, Python, and PHP.	*APIs for Scripting*

Syntax conventions

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

Notation	Meaning
`literal`	Monospace font indicates a word or other notation that must be used in the command exactly as shown.
italics	Italics indicate a parameter for which you must supply specific information. For example, for a *serverid* parameter, you must supply the id of the server.
`[-f]`	Square brackets indicate that the enclosed elements are optional. Omit the brackets when you compose the command. Elements that are not bracketed are required.
…	Ellipses (…) indicate that the preceding element can be repeated as often as needed.
element1 | *element2*	A vertical bar (|) indicates that either *element1* or *element2* is required.

Please give us feedback

We are interested in receiving opinions on this manual from our users. In particular, we'd like to hear from users who have never used Perforce before. Does this guide teach the topic well? Please let us know what you think; we can be reached at manual@perforce.com.

If you need assistance, or wish to provide feedback about any of our products, contact support@perforce.com.

Overview

This chapter provides an overview of the Helix Connector for Git.

About Helix Connector for Git

Helix Connector for Git (hereafter referred to as the Connector) augments the functionality of the Helix Versioning Engine (hereafter referred to as Helix server) to support Git clients. It services requests from both traditional Helix server clients and Git clients, and stores Git data in Git repos that reside within a depot of type **`graph`** in the Helix server. Requests are supported over both HTTP(S) and SSH connections.

The Connector improves significantly on the performance of its predecessor — Git Fusion — by eliminating the need for expensive history translation. In addition, the Connector inter-operates seamlessly with existing Git client GUIs such as GitKraken and Tower.

Clients can also access repos containing Git Large File Storage (LFS) objects. However, Git LFS is *not* supported over SSH connections.

In interaction with the Connector, Git clients can perform typical Git tasks:

- create repos implicitly, via the **`git push`** command
- create and view branches
- add and commit files
- switch branches
- push, pull, and clone repos

You can also sync data that is a heterogeneous mix of Git repo data and traditional depot data.

The Connector also supports an architecture for achieving one-way mirroring with Gitlab and GitHub. See "One-way mirroring" on page 27 for details.

Architecture and components

Helix Connector for Git consists of two main components:

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

Helix server:

- stores Git repos
- 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

Helix Connector for Git:

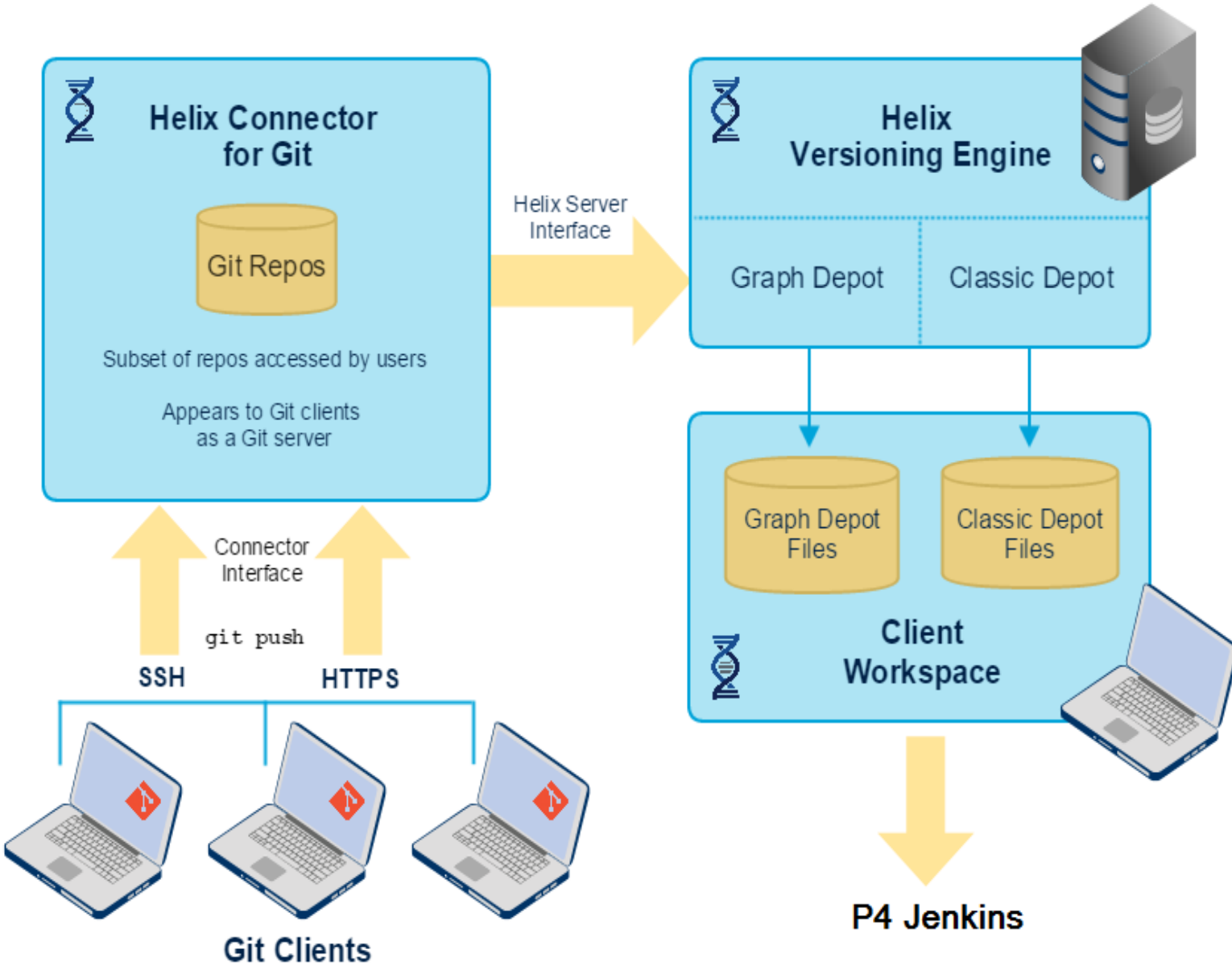
- implements both HTTPS and SSH remote protocols
- services requests from Git clients using a combination of cached data and requests to the Helix server
- serves as a Git LFS server

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 Connector.

In support of advanced workflows for blended assets, such as text and large binaries in build/test automation activities, you can also directly sync and view graph depot content through a Helix command line client into a single classic workspace. Note that you cannot *edit* files from a graph depot in a classic workspace; to edit, you need to use a Git client. The following steps describe a typical scenario:

1. A Git user pushes changes to the Connector.
2. The Connector pushes 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 may include multiple Git repos and/or one or more classic depot files.

The following diagram illustrates the flow of information from Git clients through the Connector to the Helix server and from there to the client workspace.



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 being 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

The Helix Connector for Git workflow comprises tasks you perform just once — when setting up a Connector installation — and tasks you perform multiple times during the lifetime of an installation. The section "One-time tasks" on page 4 summarizes one-time tasks. The section "Recurring tasks" on page 5 summarizes recurring tasks.

Workflow summary

- Install the Helix Connector for Git.
- Configure the Connector, including HTTPS and SSH authentication.
- Configure the Helix Versioning Engine (henceforth "Helix server") to work with the Connector; this includes depot, repo, and permissions configuration.
- Verify the configuration.
- Run **`p4 sync`** and a subset of other p4 commands against heterogeneous — Git and classic Helix — data.

One-time tasks

The following table summarizes one-time tasks:

Task	More information
Install the Helix Connector for Git.	"Install the Helix Connector for Git" on page 8
Configure the Connector. This includes configuring HTTPS and SSH authentication.	"Configure the Helix Connector for Git" on page 9
Configure the Helix server to work with the Connector. This includes depot, repo, and permissions configuration.	"Perform Connector-specific Helix server configurations" on page 11

Task	More information
Set up users.	"Set up Git users to work with the Connector" on page 14
Verify the Connector configuration.	"Verify the Connector configuration" on page 16

Recurring tasks

The following table summarizes recurring tasks:

Task	More information
Create and view graph depots.	"Create graph depots" on page 19
Create, view, and delete Git repos.	"Create and view repos" on page 20
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 21
Set up client workspaces.	"Set up client workspaces" on page 22
Run p4 sync and a subset of other p4 commands against both Git and classic Helix data.	"Sync files from graph depots" on page 23
Troubleshoot.	"Troubleshooting" on page 31

Git client tasks

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

- Obtain SSH and HTTPS URLs to communicate with the Connector. See "Set up Git users to work with the Connector" on page 14.
- Generate SSH keys to be added to the Connector, if they don't already have keys

More information

This section provides links to more detailed information on some of the general concepts discussed above.

- For help configuring Helix server for building from mixed clients, see the section Mixed client workspace in *P4 Command Reference*.
- For in-depth admin and usage information pertaining to the Helix server, see *Helix Versioning Engine Administrator Guide: Fundamentals* and *Helix Versioning Engine User Guide*.

Installation and configuration

This chapter describes how to install and configure the Helix Connector for Git (hereafter referred to as the Connector). The installation requires operating system-specific packages (see "System requirements" on page 7).

The procedure includes the following tasks:

1. Connector installation
2. Connector configuration
3. Connector-specific configuration of the Helix Versioning Engine (hereafter referred to as the Helix server)
4. User setup
5. Verification of Connector configuration
6. Pushing, cloning, and pulling repos

Installation instructions vary by operating system.

System requirements

The Helix Connector for Git requires an installation of the Helix Versioning Engine 2017.1 or later, hereafter referred to as the Helix server. In most cases, the Connector and the Helix server reside on separate machines.

The 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 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)
- CentOS or Red Hat 7.x

Note	If you want to do mirroring with a partner system then you cannot run the Connector on Centos6. See "One-way mirroring" on page 27.

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

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

Note

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

Install the Helix Connector for Git

Installing the 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 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 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.**

   The files contained in the package are installed, and information describing the main elements that have been installed is displayed.

5. **Next, configure the Helix Connector for Git.**

Configure the Helix Connector for Git

After the Connector package has been installed, you must perform additional configuration tasks.

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 begins by displaying the following summary of settings. Some settings have a default value while others 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 Connector configuration creates a Helix user called `gconn-user`. This user performs all Helix server requests required by the Connector. 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 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 be used to connect to the 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 should be updated. Updating the keys can take up to 10 minutes. You need to wait at least 10 minutes before you start using the Connector. Otherwise, the Connector will not have the 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, you can run the following command:

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

After you answer all prompts, the script creates the `GitConnector` configuration file and configures the Connector according to your choices. As it runs, the script displays information about the configurations taking place. In some cases, it may 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, you can run the following 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 created. 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 is done, read the details about what was done. Make sure nothing went wrong and note if anything still needs to be done.

Perform Connector-specific Helix server configurations

After installing and configuring the Connector, you must configure the Helix server to work with the 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 Connector authenticates Git users through HTTP or SSH (see "Set up Git users to work with the Connector" on page 14) and allows them to access resources by pull, push, and clone transactions through user or group permissions in the Helix server.

In addition, because the `gconn-user` performs all Helix server requests required by the 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 21.

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.

   Run the following command:

   ```
   $ 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.

   Run the following command:

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

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

Run the following command:

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

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

   Run the following command:

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

Tip

Instead of granting permissions to single users, you may want to create groups, assign users to groups, and then set up permissions per group. For details on how to do this, 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 19. For any additional `graph` depots that you create, you must make sure to grant `admin` permission to the user `gconn-user` (for details, see Granting permissions).

To view a list of existing depots, you can run the `p4 depots` command. For details, 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 20.

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 22.

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 folder 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 once the repo has been pushed to the Helix server, run the following command:

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

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

For information on pushing a repo to the Helix server, see "Push, clone, and pull repos via the Connector" on page 16.

Set up Git users to work with the Connector

Depending on the network protocol you selected during the 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.

Once this setup is complete, provide SSH or HTTPS URLs to Git client users. These URLs include the IP address or host name of the 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 simply send the public key to their administrator for further handling.

Once you have the SSH key, you can share the public key with the Helix server machine and then verify the key in the Connector server. By default, it takes 10 minutes for the SSH key shared with the Helix server to be authorized in the 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 keys, you can define a scope for each key to be able to quickly distinguish between them if needed, for example when you need to delete a key. The scope name could be the location or the machine name. To look for public keys along with their scope, you can run the `p4 -ztag pubkeys` command.

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; then have Git client users run the following command to verify that they can successfully connect to the 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 31.

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 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 Connector configuration

You already verified that the SSH key was added to the list of authorized keys in the Connector server as part of "Set up Git users to work with the Connector" on page 14). In addition, you can verify the 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 via the Connector

Once you have installed and configured the 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 via the Connector. For details, see Granting permissions.

The syntax for these commands varies slightly depending on whether you use SSH or HTTPS, as you will see in the next section. At this point, the Git user should have obtained the SSH

or HTTPS URLs from their administrator. For details, see "Set up Git users to work with the Connector" on page 14.

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 (hereafter referred to as 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.

This chapter describes the recurring tasks you commonly perform when directly working with graph depots and Git repos. These tasks include:

- Creating and viewing graph depots
- Creating and viewing repos
- Managing access to graph depots and repos
- Setting up client workspaces
- Syncing files from a depot

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 page 21.

- 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" on page 27.

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 page 21 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 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 21). 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

This chapter explains how to set up the Helix Connector for Git as a mirror of a repo managed by an external server.

Note	If you want to do mirroring with a partner system then you cannot run the Connector on Centos6.

Introduction

The Connector can act as a "mirror" of a repo managed by an external server such as GitHub or GitLab (CE/EE). This mirroring is achieved by configuring a webhook in the external system which fires when the external repo receives a commit. The Connector then receives the wehook message, fetches the commit from the upstream repo, and updates the Helix server.

The mirroring is one-way; the Helix server rejects attempts to directly modify the mirrored repo. Similarly, while you can use the Connector to perform additional clones of mirrored repos (as could any repo managed by the Helix server), you cannot then push commits made to these clones into the Helix server; the Connector rejects them.

The configuration of the webhook in the partner system is the responsibility of a system administrator or other user. The Helix server handles only the initial fetch of the repo, the creation of the mirrored repo in the Helix server, and the ongoing listening for webhook events.

The repo to be mirrored must be accessible for both cloning and fetching, and may be served over HTTP(S) or SSH. If the repo requires credentials, you must provide these to the Helix server in the configuration.

Assumptions

The configuration steps described in the sections that follow make these assumptions:

- The external Git server — such as GitLab or GitHub — is running on a host named `GitHost.com` and contains a repo named `repo01`.
- The repo is public (does not require credentials) and is served over HTTP.
- The Connector is running on a host named `ConnectorHost.com` and has been installed such that `/opt/perforce/git-connector` is the root of the Connector installation.
- You have a graph depot named `repo` and you are mirroring a repo named `repo01` in the partner system, named `GitHost`.

Mirror repos served over HTTP(S)

1. Log into the host `ConnectorHost.com`, where the Connector is installed as the `root` user and has write access to `/opt/perforce/git-connector`.

2. In the `gconn.conf` file, set the `log-level` for the connector to `5` — at least until mirroring is working — so that the Helix server writes any mirroring errors to `gconn.log`.

3. In the shell, set the environment variable `GCONN_CONFIG` to the absolute path to the `gconn.conf` file; if you followed the standard install `GConn` configuration script, this would be `export GCONN_CONFIG=/opt/perforce/git-connector/gconn.conf`.

4. To mirror `repo01` in the Helix server as `//repo/repo01`, invoke the gconn mirror setup task as follows:

   ```
   $ bin/gconn --mirrorhooks add repo/repo01 http://GitHost.com/repo01.git
   ```

   The precise format of the URL for the repo in the partner system varies depending on the partner system and how it is configured. In particular, if the partner system requires credentials (for example, `partner_user` and `partner_pass`) for cloning or fetching, as is the case with `private` projects in GitLab, include the credentials in the URL:

   ```
   $ bin/gconn --mirrorhooks add repo/repo01 http://partner_user:partner_pass@GitHost.com/
   repo01.git
   ```

   The Connector does the initial clone of the repo from the partner system. The partner system:

   - Delivers the repo contents and metadata as it would during a clone by a Git user; based on this, the Helix server creates the repo cache in `/opt/perforce/git-connector/repo`.
   - Creates the repo spec in the Helix server.
   - Does the initial population of the repo content in Helix.

     The Connector also writes the mirror configuration — both to a file and to the terminal — and prompts you to set up the webhook in the partner system.

     Be sure to note the `secret` string that is generated, as you you will need to provide this string in the webhook setup for proper authentication of the webhook to the Connector. This information is stored in `/opt/perforce/git-connector/repo/repo01/.mirror.config`.

5. Set up the webhook in the partner system per that system's procedure. The webhook should hit the following URL for the Connector: https://ConnectorHost.com/mirrorhooks.

   Be sure to disable SSH certificate validation in the partner system's webhook configuration.

   Some systems — such as GitLab — let you choose from a set of events for which the webhook should fire; you should at least include `push` events, but you can also include `tag` so that creation and updates to tags in the partner are mirrored to the Helix server.

6. Test the webhook (if the partner system provides this). Anything other than a `200` success code indicates a configuration problem.

To diagnose problems with mirroring, examine `/opt/perforce/git-connector/gconn.log` or `/opt/perforce/git-connector/repo/repo01/.mirror.log`.

Mirror repos served over SSH

Additional configuration steps are required for both the Connector and the partner system in order to mirror repos over SSH.

On the Connector host, as the `root` user:

- Create a `.ssh` directory for the *webserver-user*.

```
$ mkdir /webserver-user-homedir/.ssh
$ chown webserver_user:gconn-auth /webserver-user-homedir/.ssh
```

- Generate an SSH key pair for the *webserver-user*.

  1. As the `root` user on the Connector host, issue the following command:

     ```
     $ su -s /bin/bash - webserver-user
     $ ssh-keygen -t rsa -b 4096 -C webserver-user@ConnectorHost
     ```

     Hit return when prompted for a passphrase.

  2. Locate the public key thus created in `.ssh/id_rsa.pub` in the web server user's home directory and add it to the partner system.

You may want to have a service user be the bearer of this SSH key, and you must ensure that this user has permission to clone and fetch from the repo or project in the partner system you are mirroring with the Connector. Consult your partner system's documentation for this procedure.

Once the keys have been generated and added to the partner system, the steps for mirroring are the same as for those described in "Mirror repos served over HTTP(S)" on page 27, with these exceptions:

- The steps must be performed as the *webserver-user*.
- When invoking the gconn mirror setup task, use the SSH string instead of the HTTPS URL, as in the following example:

```
$ bin/gconn --mirrorhooks add repo/repo01 git@GitHost:repo01.git
```

Troubleshooting

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

Connection problems:

Permissions problems:

Branch problems:

To facilitate troubleshooting: "Special Git commands" on page 38.

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'…	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 * //...`

Problem	Solution

`git@ConnectorHost's password:`	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 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	Grant the permission to delete a repo in the graph depot: `p4 grant-permission -u bruno -p delete-ref -d gD1`

Problem	Solution
`remote:  ! [remote rejected] dev (User bruno does not have delete-ref  privilege for reference refs/heads/dev.)`	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 21.
- 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 21.
- 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 Connector special command help.
- **git clone git@ConnectorHost:@info**: Shows 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/).