HelixCore

Helix Core Extensions Developer Guide

2020.2

November 2020

PERFORCE

www.perforce.com

Copyright © 2018-2020 Perforce Software, Inc..

All rights reserved.

All software and documentation of Perforce Software, Inc. is available from www.perforce.com. You can download and use Perforce programs, but you can not sell or redistribute them. You can download, print, copy, edit, and redistribute the documentation, but you can not sell it, or sell any documentation derived from it. You can not modify or attempt to reverse engineer the programs.

This product is subject to U.S. export control laws and regulations including, but not limited to, the U.S. Export Administration Regulations, the International Traffic in Arms Regulation requirements, and all applicable end-use, end-user and destination restrictions. Licensee shall not permit, directly or indirectly, use of any Perforce technology in or by any U.S. embargoed country or otherwise in violation of any U.S. export control laws and regulations.

Perforce programs and documents are available from our Web site as is. No warranty or support is provided. Warranties and support, along with higher capacity servers, are sold by Perforce.

Perforce assumes no responsibility or liability for any errors or inaccuracies that might appear in this book. By downloading and using our programs and documents you agree to these terms.

Perforce and Inter-File Branching are trademarks of Perforce.

All other brands or product names are trademarks or registered trademarks of their respective companies or organizations.

Any additional software included within Perforce is listed in "License Statements" on page 70.

Contents

How to use this developer guide

This section provides information on typographical conventions, feedback options, and additional 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.
`-a -b`	Both *a* and *b* are required.
`{-a | -b}`	Either *a* or *b* is required. Omit the curly braces when you compose the command.
`[-a -b]`	Any combination of the enclosed elements is optional. None is also optional. Omit the brackets when you compose the command.
`[-a | -b]`	Any one of the enclosed elements is optional. None is also optional. Omit the brackets when you compose the command.
`...`	Previous argument can be repeated. ▪ `p4 [g-opts] streamlog [ -l -L -t -m max ] stream1 ...` means `1` or more stream arguments separated by a space ▪ See also the use on `...` in Command alias syntax in the *Helix Core P4 Command Reference* **Tip** `...` has a different meaning for directories. See Wildcards in the *Helix Core P4 Command Reference*.

Feedback

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

Other documentation

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

> **Tip**
> You can also search for Support articles in the Perforce Knowledgebase.

Earlier versions of this guide

2020.2, 2020.1, 2019.2, 2019.1

Extension Overview

Helix Core server extensions are a means for administrators to customize workflow. These extensions allow you to extend product behavior in a close integration between the Helix Core server runtime and your custom logic.

Server extensions are self-contained bundles of code, metadata, and other assets that interface with Helix Core server through the extensions "Classes and methods" on page 34. The extension code runtime is embedded within the Helix Core server.

Server extensions are versioned in a special **`extensions`** depot. If you invoke the **`p4 depots -t extension`** command, the output lists the Helix Core Extensions depot with **`.p4-extensions`** as its name.

Note
The output of the `p4 depots` command does not list the Extensions depot because this special depot is not for the ordinary user. Instead, the Extensions depot is for the administrative user with **`super`** access (see p4 protect). The command to manage server extensions, `p4 extension`, requires **`super`** access.

Some built-in advantages of server extensions when compared with triggers:

- a single scripting language supports portability to any platform that Helix Core server supports
- programmatic API allows for integration of an extension with the Helix Core server
- configurable on a global or per depot basis
- users that the superuser has authorized to configure extensions, can do so without super-user involvement within the repo or depot that user owns
- forward compatible across product upgrades (API/runtime pinning)
- server-managed installation, execution, and replication
- Internationalization (i18n) compatibility
- can issue pre-authenticated Perforce client commands, so no need to manage a ticket for the server extension
- easy administration for installation, update or removal, by using the p4 extension command and the global and instance specs (instead of the flat triggers table). See "Server extension configuration (global and instance specs)" on page 10.
- includes libraries for issuing web requests, sending email, storage

Note
See also:

- the general comparison of Triggers and Extensions in *Helix Core Server Administrator Guide*
- "Example usage of a server extension compared to a trigger" on page 17.

Server extension creation

Workflow to create and deploy a server extension

Create

1. On the client, run **`p4 extension --sample extName`** to create a skeleton of a server extension under the **`extName`** directory.
2. Edit the placeholder data in the **`extName/manifest.json`** file. (See "Server Extension JSON manifest fields" on page 12.)
3. Code the server extension by editing **`extName/main.lua`** to put in the logic.

   See the information about examples at
   https://swarm.workshop.perforce.com/files/guest/perforce_software/extensions/main/README.md
   and the examples at a specific release, such as
   https://swarm.workshop.perforce.com/files/guest/perforce_software/extensions/2019.1
4. Run `p4 extension` **`--package extName`** to create ***`extName`*`.p4-extension`**.

Test

Test that the server extension does what you expect, make changes if necessary, and retest until the server extension is ready for production. See also "Server extension errors and troubleshooting" on page 32.

Deploy

Note
To actually perform the install or delete of an extension, the `--yes` option is required. Without `--yes`, the `p4 extension` command merely reports what it would do without actually performing the install or delete.

1. Install the server extension with `p4 extension --install extName.p4-extension --yes`
2. Configure the server extension:
   a. Configure the **global** settings with `p4 extension --configure namespace::extName`
   b. Configure the **instance** settings with `p4 extension --configure extName --name extCfg`

   See "Server extension configuration (global and instance specs)" on the next page

About versions and code lines

- Multiple versions of a server extension can be installed. For example, you can keep the first version in the bug-fix branch and install a different version in the new-feature branch.
- Multiple versions of a server extension can be running simultaneously. For example, `Release1.0-fileSizeCheck` might be running in `//depot/main` while `Release1.1-fileSizeCheck` is running in `//depot/dev`

About the data directory

- Each version of a server extension can have a different data directory.
- For a given version of a server extension, the data directory is shared between all instances. Consider whether concurrent access to data could affect your server extensions. For example, you might need a write lock on a log file.
- A server extension's data directory is not replicated.

Note
- The server extension is responsible for parsing and using any data the user enters in response to the global GlobalConfigFields and the instance `InstanceConfigFields` functions. See *Helix Core Extensions Developer Guide* > Class Helix.Core.Server.
- A server extension is loaded into the Helix Core server memory when an event occurs that launches that server extension. For example, when a client submits a file to the server's depot. The server extension persists in server memory for the lifetime of that client's connection to the server.

Other examples

See the server extension examples at https://swarm.workshop.perforce.com/files/guest/perforce_software/extensions

Server extension configuration (global and instance specs)

After you have installed the server extension, configure the extension specs:

global	Usage: `p4 extension --configure namespace::extensionName` The global spec applies to all instances of the server extension. For example, to enable all instances of a server extension to send an email, the name of the mail server would be configured globally. The `super` user supplies the **global** details about the server extension configuration that apply to all instances of this extension, such as: ■ the list of groups whose members can create instances of this particular extension ■ runtime limits, such as maximum number of users or maximum number of files Note: The default namespace is `ExampleInc` and the the default extension name is `extName` See .
instance	Usage: `p4 extension --configure ExampleInc::extName --name instanceOfExtensionName` For example, `p4 extension --configure ExampleInc::extName --name Release1.0-fileSizeCheck` One or more "instance" specs are required. For example, `Release1.0-fileSizeCheck-instance1` might apply to `//depot/test` and specify a certain maximum file size and maximum number of files, while `Release1.0-fileSizeCheck-instance2` might apply to `//depot/main` and specify a different maximum file size and maximum number of files. Use the `-configure` and `-name` flags together to create a named instance of the server extension, parameterizing the server extension to be run with specific settings: ■ the `-name` flag takes the name of the configuration to create or modify ■ the `-configure` flag takes the name of the extension. See .

Pre-populated, read-only fields of the extension spec

Some read-only fields are pre-populated with data from the "Server Extension JSON manifest fields" on the next page.

Spec field	Corresponding JSON field	Meaning
`ExtName`	`name`	The name of the server extension.
`ExtDescription`	`description`	The description of the server extension.
`ExtVersion`	`version`	The version number of the server extensions.
`ExtUUID`	`key`	The universally unique identifier of the server extension.

Fields that can be modified

`ExtMaxScriptTime`	The number of seconds the server extension is allowed to run for before the server terminates it.
`ExtMaxScriptMem`	The number of bytes of RAM the server extension is allowed to allocate before the server terminates it.
`ExtAllowedGroups`	The list of groups whose depot/repo owner members are allowed to create instances of the server extension. This applies to file-based events, such as `change-submit`. **Note** This does not apply to a global event, such as `form-out` that occurs whenever the server generates a form for display to the user.
`ExtEnabled`	Enable/disable an instance configuration.
`ExtP4USER`	The user account to use for the automatic logins.
`Name`	The name of the instance configuration.
`Owner`	The user who created the config.
`Update`	When the config was modified.
`Description`	User-supplied description of the instance config.

`ExtConfig`	A server extension can supply its own user-input fields here. These fields are key / value types.
`ExtRev`	The depot-rev of the extension installed in the depot. For example, `3`

Server Extension JSON manifest fields

The server extension manifest is a UTF-8 encoded JSON file containing supporting metadata that the Helix Core Server uses.

Fields

api_version	The version of the API exposed to the runtime. Valid values are: '1' (2018.2) and '20191' (2019.1). This is a required field.
compatible_ products	An array enumerating the list of Helix products the server extension works with. Valid values are 'p4' or 'p4d'. This is a required field.
default_locale	When no locale is specified or detected, use this value for translated messages. This is a required field.
description	A block of text giving a high-level description of the server extension. This is a required field.
developer	The name of the server extension developer (or company). This is a required field.
homepage_url	URL where information regarding the server extension can be found. This is a required field.
key	UUID for the server extension. The sample server extension created by the `p4 extension --sample` command creates a random value for this, but any valid UUID can be used. This is a required field.
license	The name of the license the server extension is released under. This is a required field.
license_body	The body of the license text.
manifest_ version	This number specifies the format of the manifest. Incompatible changes to the manifest in future server releases will increment it. Valid value is `1`. This is a required field.

name	This is the name of the server extension. It must consist of the characters in [`0-9a-zA-Z_-`], that is, number, letters, underscore, and hyphen. No other special characters are allowed. The name is a required field.
namespace	This is the organization that authored the server extension. For example, `ExampleInc`. This, combined with the `name` field form the fully-qualified name for the server extension. For example, `ExampleInc::extName` and this is a required field.

script_runtime	Name and version of the scripting runtime. Valid values are: 'Lua' and '5.3'. For example,
	`"language":"Lua",`
	`"version":"5.3"`

supported_ locales	List of locales the server extension will work with.
update_url	URL where automatic updates can be checked for.
version	This is the numeric version of the server extension. For example, `1.2.3` and this is a required field.
version_name	This is the named version string of the server extension. For example, `'1.2.3 alpha'` or `"1.2.3 alpha"` The 'Perforce' and 'Helix' names are reserved for Perforce, so do not use them for server extension that you write.

Example manifest.json

```
{
    "manifest_version":1,
    "api_version":1,
    "script_runtime":{
    "language":"Lua",
    "version":"5.3"
    },
    "key":"aaaaa-aaaa-aaaa-aaaa-aaaaa",
    "name":"ExtName",
    "namespace":"ExampleInc",
    "version":"1.0",
    "version_name":"1.0 beta",
```

```
    "description":"Example Extension to illustrate concepts.",
    "compatible_products":[
          "p4d"
    ],
    "default_locale":"en",
    "supported_locales":[
        "en",
        "jp"
    ],
    "developer":{
        "name":"Example Extensions Inc.",
        "url":"https://p4-extensions.example.com/"
    },
    "homepage_url":"https://p4-extensions.example.org/ExtName",
    "license":"BSD",
    "license_body":"Redistribution and use in source and binary
    forms..."
}
```

Extension basics

Installation

When you run **`p4 extension --install`**, the server extension is transferred from your client machine to the Helix Core server, where built-in validation occurs. For example, the server notifies you if the manifest is missing or is invalid.

The server stores the server extension in a depot:

- with the default name of **`.p4-extensions`** - see p4 extension in *Helix Core P4 Command Reference*,
- that is of Type **`extension`** - see p4 depot for **`Form Fields`** > **`Type:`** > **`extension`**.

The server extensions depot is a new depot type because the files associated with a server extension are separate from the files in other types of Helix Core depots.

Viewing the history of extensions

A super user can run p4 changes on the `extension` depot to see the history of installs, upgrades, and deletes.

Listing extensions

See `p4 extension` in *Helix Core P4 Command Reference*, which explains the `--list` option.

Disabling and re-enabling an extension

To disable or re-enable a server extension, in the global configuration or in the instance configuration, change the value of `ExtEnabled`. See "Fields that can be modified" on page 11.

Deleting an extension

> **Note**
> To actually perform the install or delete of an extension, the `--yes` option is required. Without `--yes`, the `p4 extension` command merely reports what it would do without actually performing the install or delete.

To delete a server extension, use `p4 extension --delete extName --yes`

This deletes the depot file and removes the server extension's resource directory.

Directories

When a server extension is committed to the extensions depot, the server creates a subdirectory for the server extension. This directory is specified by the `server.extensions.dir` configurable. By default, the name of this directory matches the configurable's name, `server.extensions.dir`, and is located under the `P4ROOT` directory. Each server extension gets two subdirectories under the directory specified at `server.extensions.dir`, one for its resources and one that can be used as a persistent scratch space.

See also:

- "GetArchDirFileName(string) -> string" on page 47
- "GetDataDirFileName(string) -> string" on page 47

Additional information

Prerequisite	Knowledge of Lua.
Limitation	Any plain-Lua library is compatible, provided that it matches version of the server extension runtime. However, external Lua libraries that require native machine code for a specific processor are not compatible.
Replication supported, but not DVCS	Server extensions are stored as normal depot files and use the existing file replication mechanism for transfer to replicas. Replicas install extensions as soon they received them from the commit server. However, if your server extension has files that are stored in the `server.extension.dir` directory, those files will not be replicated. DVCS instances do not receive server extensions.
Backup and restore	Server extensions are depot files and server extension specs are included in checkpoints. However, the contents of the `server.extensions.dir` directory should be added to the list of depot directories to back up. Some of these files might be in an inconsistent state if they are accessed while the server is running because they are owned by your server extensions rather than by the Helix Core server. In the case of a loss of the `server.extensions.dir` data, the server will automatically recreate the files on first access because that data is on the server in the depot of type `extension`.
Internationalization	The server extensions API and manifest allow extensions to return messages localized by locale to the user. Message strings are encoded as UTF-8 and can be converted to other character sets. See the *Internationalization Notes*.
Diagnostics / debugging	To log line-delimited JSON from within a server extension, use the `Helix.Core.Server.log()` function. See *Helix Core Extensions Developer Guide* > Class Helix.Core.Server.
Documentation from the command-line	See `p4 help serverextensionintro`

Server extension callbacks

This section compares the function syntax of a server extension to the equivalent logic in a trigger.

Example usage of a server extension compared to a trigger

Triggers	Server extension

`Triggers:`
`name form-in change script.pl`
	`function InstanceConfigEvents()`
	`    return { ["form-in"] = "change" }`
	`end`
	`function FormIn()`
	`    ...`
	`end`

`Triggers:`
`... change-submit //a/b/c/... script.sh`
`... change-submit //a/c/d/... script.sh`
`... change-submit -//a/d/e/... script.sh`
	`function InstanceConfigEvents()`
	`    return { ["change-submit"] = {`
	`"//a/b/c/...", "//a/c/d/...", "-//a/d/e/..."`
	`} }`
	`end`
	`function ChangeSubmit()`
	`    ...`
	`end`

Event Callbacks

The following table lists the Helix Core events that can cause a server extension to run. Instead of managing a trigger table, the server extensions use `InstanceConfigFields()` function to bind themselves to events.

> **Note**
> In the following table,
>
> - a **form type** can be: "branch", "change", "client", "depot", "group", "job", "label", "protect", "server", "spec", "stream", "triggers", "typemap", "user"
> - **server IDs** might be similar to “build-123”, "commit1"

- a “**pre-user-” and “post-user-”** example might be similar to {“pre-user-obliterate”, “post-user-add”}

Event (same as trigger name)	Param	Method	Returns
archive	depot paths	Archive()	boolean
auth-check-sso	"auth"	AuthCheckSSO()	boolean
auth-invalidate	"auth"	AuthInvalidate()	boolean
auth-pre-sso	"auth"	AuthPreSSO()	boolean, optional string, optional boolean
auth-init-2fa	"auth"	MFABegin()	`Helix.Core.Server.MFA.Status`, `Helix.Core.Server.MFA.Scheme`, string, string
auth-check-2fa	"auth"	MFACheck()	`Helix.Core.Server.MFA.Status`, string
auth-pre-2fa	"auth"	MFAPre()	`Helix.Core.Server.MFA.Status`, table
bgtask	unset	BGTask()	boolean
change-commit	depot paths	ChangeCommit()	boolean
change-content	depot paths	ChangeContent()	boolean
change-failed	depot paths	ChangeFailed()	boolean
change-submit	depot paths	ChangeSubmit()	boolean

Event (same as trigger name)	Param	Method	Returns
command	“pre-user-*” and “post-user-*”	Command()	boolean
edge-content	depot paths	EdgeContent()	boolean
edge-submit	depot paths	EdgeSubmit()	boolean
fix-add	"fix"	FixAdd()	boolean
fix-delete	"fix"	FixDelete()	boolean
form-commit	a form type	FormCommit()	boolean
form-delete	a form type	FormDelete()	boolean
form-in	a form type	FormIn()	boolean
form-out	a form type	FormOut()	boolean
form-save	a form type	FormSave()	boolean
graph-fork-repo	repo paths	GraphForkRepo()	boolean
graph-lfs-push	repo paths	GraphLFSPush()	boolean
graph-push-complete	repo paths	GraphComplete()	boolean
graph-push-reference-complete	repo paths	GraphPushReferenceComplete()	boolean

Event (same as trigger name)	Param	Method	Returns
graph-push-reference	repo paths	GraphPushReference()	boolean
graph-push-start	repo paths	GraphPushStart()	boolean
heartbeat-missing	heartbeat	HeartbreatMissing()	boolean
heartbeat-resumed	heartbeat	HeartbreatResumed()	boolean
heartbeat-dead	heartbeat	HeartbreatDead()	boolean
journal-rotate-lock	server IDs	JnlRotateLock()	boolean
journal-rotate	server IDs	JnlRotate()	boolean
pull-archive	“pull”	PullArchive()	boolean
push-commit	depot paths	PushCommit()	boolean
push-content	depot paths	PushContent()	boolean
push-submit	depot paths	PushSubmit()	boolean
service-check	"auth"	ServiceCheck()	boolean
shelve-commit	depot paths	ShelveCommit()	boolean
shelve-delete	depot paths	ShelveDelete()	boolean
shelve-submit	depot paths	ShelveSubmit()	boolean

Server extension session variables

This section assumes you have read the "Server extension callbacks" on page 17 page.

A server extension can get the data about the current command context from the server. The server extension is responsible for interpreting and using these appropriately.

The ***maxError...*** variables refer to circumstances that prevented the server from completing a command. For example, an operating system resource issue. Note also that client-side errors are not always visible to the server and might not be included in the **maxError** count.

The **terminated** variable indicates whether the command exited early and why.

Argument	Description	Available for type
%action%	Either null or a string reflecting an action taken to a changelist or job. For example,"**pending change 123 added**" or "**submitted change 124 deleted**" are possible **%action%** values on **change** forms, and "**job000123 created**" or "**job000123 edited**" are possible **%action%** values for **job** forms.	**form-commit**
%archiveList%	Filename containing files to be pulled	**pull-archive**
%argc%	Command argument count.	all except **archive**
%args%	Command argument string.	all except **archive**
%argsQuoted%	Command argument string that contains the command arguments as a percent-encoded comma-separated list.	all except **archive**

Argument	Description	Available for type

`%changelist%`, `%change%`	The number of the changelist being submitted. The abbreviated form `%change%` is equivalent to `%changelist%`.
	A `change-submit` event is passed the pending changelist number; a `change-commit` event receives the committed changelist number.
	A `shelve-commit` or `shelve-delete` event receives the changelist number of the shelf.
		`change-submit`
		`push-submit`
		`change-content`
		`push-content`
		`change-commit`
		`push-commit`
		`fix-add`
		`fix-delete`
		`form-commit`
		`shelve-commit`
		`shelve-delete`

`%changeroot%`	The root path of files submitted.	`change-commit` `push-commit`
`%client%`	Calling user's client workspace name.	all
`%clientcwd%`	Client's current working directory.	all except `archive`
`%clienthost%`	Hostname of the user's workstation (even if connected through a proxy, broker, replica, or an edge server.)	all
`%clientip%`	The IP address of the user's workstation (even if connected through a proxy, broker, replica, or an edge server.)	all
`%clientprog%`	The name of the user's client application. For example, P4V, P4Win	all

Argument	Description	Available for type
`%clientversion%`	The version of the user's client application.	all
`%command%`	Command name.	all except `archive`

`%depotName%`	The graph depot in which the repo resides.	`graph-push-start`
		`graph-push-reference`
		`graph-push-reference-complete`
		`graph-push-complete`

`%email%`	The user's email address.	`auth-pre-2fa`
		`auth-init-2fa`
		`auth-check-2fa`

`%file%`	Path of archive file based on depot's `Map:` field. If the `Map:` field is relative to `P4ROOT`, the `%file%` is a server-side path relative to `P4ROOT`. If the `Map:` field is an absolute path, the `%file%` is an absolute server-side path.	`archive`
`%firstPushedChange%`	First new changelist number.	`command`

`%formfile%`	Path to temporary form specification file. To modify the form from an `in` or `out` event, overwrite this file. The file is read-only for events of type `save` and `delete`.	`form-commit`
		`form-save`
		`form-in`
		`form-out`
		`form-delete`

`%formname%`	Name of form (for instance, a branch name or a changelist number).	`form-commit,`
		`form-save`
		`form-in`
		`form-out`
		`form-delete`

Argument	Description	Available for type
`%formtype%`	Type of form (for instance, `branch`, `change`, and so on).	`form-commit`, `form-save` `form-in` `form-out` `form-delete`
`%fullname%`	The user's fullname.	`auth-pre-2fa` `auth-init-2fa` `auth-check-2fa`
`%groups%`	List of groups to which the user belongs, space-separated.	all except `archive`
`%host%`	The IP address of the host of the user.	`auth-pre-2fa` `auth-init-2fa` `auth-check-2fa`
`%intermediateService%`	A broker or proxy is present.	all except `archive`
`%jobs%`	A string of job numbers, expanded to one argument for each job number specified on a `p4 fix` command or for each job number added to (or removed from) the `Jobs:` field in a `p4 submit`, or `p4 change` form.	`fix-add`, `fix-delete`
`%lastPushedChange%`	Last new changelist number.	`command`
`%maxErrorSeverity%`	One of `empty`, `error`, or `warning`.	all except `archive`
`%maxErrorText%`	Error number and text.	all except `archive`

Argument	Description	Available for type
`%maxLockTime%`	A user-specified value that specifies the number of milliseconds for the longest permissible database lock. If this variable is set, it means the user has overridden the group setting for this value.	all except `archive`
`%maxResults%`	A user-specified value that specifies the amount of data buffered during command execution. If this variable is set, it means the user has overridden the group setting for this value.	all except `archive`
`%maxScanRows%`	A user-specified value that specifies the maximum number of rows scanned in a single operation. If this variable is set, it means the user has overridden the group setting for this value.	all except `archive`
`%method%`	The authentication method from list-methods (may be set to "unknown").	
`%newValue%`	Graph depot new SHA value.	`graph-push-reference`
`%oldchangelist%`	If a changelist is renumbered on submit, this variable contains the old changelist number.	`change-commit` `push-commit`
`%oldPassword%`	The old value of the password.	`auth-set`

Argument	Description	Available for type
`%oldValue%`	Graph depot previous SHA value.	`graph-push-reference`
`%op%`	Operation: `read`, `write`, or `delete`.	`archive`
`%password%`	The value of the password.	`auth-check`
`%peerhost%`	If the command was sent through a proxy, broker, replica, or edge server, the hostname of the proxy, broker, replica, or edge server. (If the command was sent directly, `%peerhost%` matches `%clienthost%`)	all
`%peerip%`	If the command was sent through a proxy, broker, replica, or edge server, the IP address of the proxy, broker, replica, or edge server. (If the command was sent directly, `%peerip%` matches `%clientip%`)	all
`%P4PORT%`	The host port to which the client connects. If the client connects to the server through an intermediary, this will hold the port number of the intermediary. If there's no intermediary, this will hold the same value as the `%serverAddress%` variable.	`auth-check-sso` (client-side script only)

Argument	Description	Available for type
`%pusher%`	The user credited with the push.	`graph-push-start` `graph-push-reference` `graph-push-reference-complete` `graph-push-complete`
`%quote%`	A double quote character.	all
`%reference%`	Graph depot reference information.	`graph-push-reference`
`%refFlags%`		
`%refType%`		
`%repo%`	The repo, which has `.git` as a suffix, but otherwise is identical to `%repoName%`.	`graph-push-start` `graph-push-reference` `graph-push-reference-complete` `graph-push-complete`
`%repoName%`	The name of the repo.	
`%rev%`	Revision of archive file	`archive`
`%scheme%`	The authentication scheme set by init-auth (can be set to "unknown").	`auth-init-2fa`
`%serverAddress%`	The IP address and port of the Helix Core server, passable only in the context of a client-side script specified by `P4LOGINSSO`.	`auth-check-sso` (client-side script only)
`%serverhost%`	Hostname of the Helix Core server.	all
`%serverid%`	The value of the Helix Core server's `server.id`. See `p4 serverid` in the *Helix Core P4 Command Reference* for details.	all
`%serverip%`	The IP address of the server.	all

Argument	Description	Available for type
`%servername%`	The value of the Helix Core server's `P4NAME`.	all
`%serverport%`	The transport, IP address and port of the Helix Core server, in the format ***`prefix`*`:ip_address:port`**. ***`prefix`*** can be one of `ssl`, `tcp6`, or `ssl6`. This means that the command `p4 -p %serverport%` can be used to connect to the server no matter which type of connection the server uses.	all
`%serverroot%`	The `P4ROOT` directory of the Helix Core server.	all
`%serverservices%`	A string specifying the role of the server. One of the following: ■ `standard` ■ `replica` ■ `broker` ■ `proxy` ■ `commit-server` ■ `edge-server` ■ `forwarding-replica` ■ `build-server` ■ `P4AUTH` ■ `P4CHANGE`	all except `archive`

Argument	Description	Available for type
`%serverVersion%`	Version string for the server that terminated if the command exited early. Reason for termination is given in `%termType%`.	all except `archive`
`%specdef%`	Expanded to the spec string of the form in question.	`form`

Argument	Description	Available for type

`%submitserverid%`	If this is not a multi-server installation, `%submitserverid%` is always empty.
	
	In a multi-server installation, for any change event:
	
	■ if the submit was run on the commit server, `%submitserverid%` equals `%serverid%`.
	
	■ if the submit was run on the edge server, `%submitserverid%` does not equal `%serverid%`. In this case, `%submitserverid%` holds the edge server's server id.
	
	If there is a forwarding replica between the commit server and the edge server, then `%submitserverid%` actually holds the forwarding replica's server id.
	
	See `p4 serverid` in the *Helix Core P4 Command Reference*.
		`change-submit`
		`change-content`
		`change-commit`
		
		Not available for `push-`* events.

Argument	Description	Available for type
`%targetport%`	The serverport of the target server being monitored. Corresponds to the P4TARGET or the `-t target` value that the p4 heartbeat command uses.	`heartbeat-missing` `heartbeat-resumed` `heartbeat-dead`
`%terminated%`	The value of `0` indicates that the command completed. A value of `1` indicates that the command did not complete.	
`%termType%`	The reason for early termination. This might be one of the following: ▪ `'p4 monitor terminate'` ▪ `client disconnect` ▪ `maxScanRows` ▪ `maxLockTime` ▪ `maxResults` See also `%serverVersion%`.	all except `archive`
`%token%`	The stashed token from the last init-auth (can be empty).	`auth-init-2fa`
`%triggerdir%`	Pull.trigger.dir used for tmp files.	`edge-content`

Argument	Description	Available for type
`%triggerMeta_depotFile%`	Third field in server extension definition. For a change-submit event, it is the path for which the server extension is expected to match. For a form-out event, it might be the form type to which the server extension is expected to apply.	all except `archive`
`%triggerMeta_name%`	Server extension name: first field from server extension definition.	all except `archive`
`%triggerMeta_trigger%`	Server extension type: second field in server extension definition.	all except `archive`
`%user%`	Helix server username of the calling user.	all

Server extension errors and troubleshooting

As seen from the client-side, here are some errors and where to look for answers:

Error	Action
`extName validation failed: logged by server`	When a trigger or server extension has a failure, the `logged by server` message can be presented. It is an intentionally uninformative message because trigger or server extension failures can include stack traces for that user code, which might be sensitive information. The server's log file will contain the full output.
`Partner exited unexpectedly`	This message means that the remote procedure call (RPC) connection between the client and server was not gracefully closed. The cause could be in the physical network connection or the server. The server's log file might provide more details about the problem. For example, the following error indicates a server crash: `Perforce server error: Process 1234 exited on a signal 11`! This should be reported to Perforce Technical Support.

Third party libraries for Extensions

The following libraries are bundled with the Helix Core server for use with Extensions:

The cURL data transfer library via the Lua-cURLv3 binding.

The SQLite3 SQL database via the LuaSQLite3 binding.

The cjson JSON encoder/decoder.

Classes and methods

The "Class P4.P4 " belowdeals with general things, such as connecting to the server, and the "Class Helix.Core.Server" on page 46 class is specific to server extensions.

Class P4.P4

An interface to the Helix server client API.

See the information about examples at https://swarm.workshop.perforce.com/files/guest/perforce_software/extensions/main/README.md
and the examples at a specific release, such as
https://swarm.workshop.perforce.com/files/guest/perforce_software/extensions/2019.1

Class methods

P4.P4:new -> P4.P4

Constructs a new **P4.P4** object.

```
p4 = P4.P4:new()
```

Instance Methods

p4.api_level= number -> number

Sets the API compatibility level desired. Using this method allows you to lock your script to the output format of an older Helix server release and facilitate seamless upgrades. This method, if called, should be called prior to calling `p4:connect()`

```
p4 = P4.P4:new()
p4.api_level = 86 # Lock to 2019.1 format
p4:connect()
```

For more information about the API levels, see the Support Knowledgebase article, "Helix Client Protocol Levels".

p4.api_level -> number

Returns the current Helix C/C++ API compatibility level. Each iteration of the Helix Core server is given a level number. As part of the initial communication, the client protocol level is passed between client application and the Helix Core server. This value, defined in the Helix C/C++ API, determines the communication protocol level that the Helix server client will understand. All subsequent responses from the Helix Core server can be tailored to meet the requirements of that client protocol level.

For more information, see "Helix Client Protocol Levels"

p4.charset= string -> string

Sets the character set to use when connecting to a Unicode-enabled server. Do not use when working with non-Unicode-enabled servers.

```
p4 = P4.P4:new()
p4.charset = "utf-8"
p4:connect()
```

p4.charset -> string

Get the name of the character set in use when working with Unicode-enabled servers.

p4.client= string

Set the name of the client workspace you wish to use. This method, if called, should be called prior to calling `p4:connect()`

```
p4 = P4.P4:new()
p4.client = "www"
p4:connect()
```

p4.client -> string

Get the name of the Helix server client currently in use.

```
p4 = P4.P4:new()
print( p4.client )
```

p4:connect() -> boolean

Connect to the Helix Core server. You must connect before you can execute commands.

```
p4 = P4.P4:new()
p4:connect()
```

p4:is_connected() -> boolean

Test whether or not the session is connected.

```
p4 = P4.P4:new()
p4:is_connected()
```

p4.cwd -> string

Get the current working directory for this server extension.

```
p4 = P4.P4:new()
print( p4.cwd )
```

p4:disconnect() -> boolean

Disconnect from the Helix Core server.

```
p4 = P4.P4:new()
p4:connect()
p4:disconnect()
```

p4.env(string) -> string

Get the value of a Helix server environment variable, taking into account `P4CONFIG` files and (on Windows and OS X) the registry or user preferences.

```
p4 = P4.P4:new()
print p4.env( "P4PORT" )
```

p4.errors -> table

Returns the table of errors which occurred during execution of the previous command.

p4.exception_level = number

Enables or disables the throwing of exceptions. The following two levels are supported:

- `0` disables all exception raising and makes the interface completely procedural.
- `1` causes exceptions to be raised for both errors and warnings. This is the default.

p4.exception_level -> number

Returns the current exception level.

If 0, exceptions are not used. When set to 1, exceptions are enabled.

p4.format_spec("<spectype>", table) -> string

Converts the fields in a table containing the elements of a Helix server form (spec) into the string representation familiar to users.

The first argument is the type of spec to format: for example, **`client`**, **`branch`**, **`label`**, and so on. The second argument is the table to convert to a string.

p4.graph= boolean

Enable or disable support for graph depots. You can enable or disable support for graph depots both before and after connecting to the server.

```
p4 = P4.P4:new()
p4.graph = false
```

p4.graph -> boolean

Returns whether or not support for Helix server graph depots is enabled.

```
p4 = P4.P4:new()
print( p4.graph )
p4.graph = false
print( p4.graph )
```

p4.host= string

Set the name of the current host. If not called, defaults to the value of **`P4HOST`** taken from any **`P4CONFIG`** file present, or from the environment as per the usual Helix server convention. This method, if called, should be called prior to calling **`p4:connect()`**

```
p4 = P4.P4:new()
p4.host = "workstation123.example.com"
p4:connect()
```

p4.host -> string

Get the current hostname.

```
p4 = P4.P4:new()
print( p4.host )
```

p4.input= (string|table) -> boolean

(For forms or specs) - Store input for the next command.

Use this method to submit a client spec, or create a job. The structure of the input can be a string representing the plain text version of a form you would edit in the Helix Visual Client (P4V) or a Lua data structure.

Call this method prior to running a command requiring input from the user. When the command requests input, the specified data will be supplied to the command.

You can pass a string. For commands that take multiple inputs from the user, you can pass a table of strings. If you pass a table, the table will be shifted each time Helix server asks the user for input.

> **Note**
> Use this method for forms, and not for cases where you have a list of arguments. In this regard, let us compare the Extensions API to the command line client. The command line client has the `-x` option, which allows you to feed additional arguments to commands from a file or standard input. For example,
>
> `p4 -x - sync < file-list.txt`
>
> This is useful if the operating system has a limit on the length of a shell command. However, if you use the Extensions API, there is no such limit. For example,
>
> `p4.run('sync', file1, file2, ...)`
>
> works for any number of arguments. In effect, the equivalent of command arguments is explicit in the `p4:run()` call.
>
> The `p4.input` method is somewhat analogous to how the command-line client supports the use of the `-i` option for standard input with a p4 shell command, such as p4 client -i

p4.messages -> table

Returns a table of `P4.Message` objects.

p4.p4config_file -> string

Get the path to the current **P4CONFIG** file.

```
p4 = P4.P4:new()
print( p4.p4config_file )
```

p4.parse_spec("<spectype>", string) -> table

Parses a Helix server form (spec) in text form into a table using the spec definition obtained from the server.

The first argument is the type of spec to parse: **client**, **branch**, **label**, and so on. The second argument is the string buffer to parse.

p4.password= string

Set your Helix server password or the ticket value to be used for this connection. If no password or ticket is given, it uses the value of P4PASSWD from any P4CONFIG file in effect, or from the environment according to the normal Helix server conventions.

```
p4 = P4.P4:new()
p4.password = "mypass"
p4:connect()
```

p4.password -> string

Get the current password or ticket. This may be the password in plain text, or if you've used **P4#run_login()**, it'll be the value of the ticket you've been allocated by the server.

```
p4 = P4.P4:new()
print( p4.password )
```

p4.port= string

Set the host and port of the Helix server you want to connect to. If not used, the value defaults to the value of **P4PORT** in any **P4CONFIG** file in effect. If there is no such value, it defaults to the value of **P4PORT** taken from the environment.

```
p4 = P4.P4:new()
p4.port = "localhost:1666"
p4:connect()
...
p4.disconnect()
```

p4.port -> string

Get the value of the **P4PORT** of the current Helix server.

```
p4 = P4.P4:new()
print( p4.port )
```

p4.prog= string

Set the name of the script, as reported to Helix server system administrators.

```
p4 = P4.P4:new()
p4.prog = "sync-script"
p4:connect()
...
p4.disconnect()
```

p4.prog -> string

Get the name of the program as reported to the Helix server.

```
p4 = P4.P4:new()
p4.prog = "sync-script"
print( p4.prog )
```

p4:reset()

Reset messages, warnings, and errors from a previous run() call to its default value.

p4:run(command, [arguments…]) -> table

Runs the specified Helix server method call with the arguments supplied. The arguments should be passed as quoted and comma-separated strings, with no leading space. For example:

```
p4:run("print","-o","test-print","-q","//depot/Jam/MAIN/src/expand.c")
```

The method returns a table of results whether the command succeeds or fails. The array table, however, can be empty. Whether the elements of the table are strings or tables depends on (a) server support for tagged output for the command, and (b) whether tagged output was disabled by calling **p4.tagged = false**.

In the event of errors or warnings, and depending on the exception level in force at the time, the method will throw an exception. If the current exception level is below the threshold for the error or warning, the method returns the output as normal, and the caller must explicitly review **p4.errors** and **p4.warnings**.

p4.server_level -> number

Returns the current Helix server level. Each iteration of the Helix server is given a level number. As part of the initial communication this value is passed between the client application and the Helix server. This value is used to determine the communication that the Helix server will understand. All subsequent requests can therefore be tailored to meet the requirements of this Server level.

For more information about the Helix server version levels, see the Support Knowledgebase article, "Helix server Version Levels".

p4.server_unicode -> boolean

Detects whether or not the server is in unicode mode.

p4.streams= boolean

Enable or disable support for streams. You can enable or disable support for streams both before and after connecting to the server.

```
p4 = P4.P4:new()
p4.streams = false
```

p4.streams -> boolean

Detects whether or not support for Helix server Streams is enabled.

```
p4 = P4.P4:new()
print ( p4.streams )
p4.streams = false
print ( p4.streams )
```

p4.tagged= boolean

Sets tagged output. By default, tagged output is on.

```
p4 = P4.P4:new()
p4.tagged = false
```

p4.tagged -> boolean

Detects whether or not you are in tagged mode.

```
p4 = P4.P4:new()
print ( p4.tagged )
p4.tagged = false
print ( p4.tagged )
```

p4.ticket_file = string

Sets the location of the **P4TICKETS** file.

```
p4 = P4.P4:new()
p4.ticket_file = "/home/bruno/tickets"
```

p4.ticket_file -> string

Get the path to the current **P4TICKETS** file.

```
p4 = P4.P4:new()
print( p4.ticket_file )
```

p4.track= -> boolean

Instruct the server to return messages containing performance tracking information. By default, server tracking is disabled.

```
p4 = P4.P4:new()
p4.track = true
```

p4.track -> boolean

Detects whether or not performance tracking is enabled.

```
p4 = P4.P4:new()
p4.track = true
print ( p4.track )
p4.track = false
print ( p4.track )
```

p4.user= string

Set the Helix server username. If not called, defaults to the value of **P4USER** taken from any **P4CONFIG** file present, or from the environment as per the usual Helix server convention. If used, should be called before connecting to the Helix server.

```
p4 = P4.P4:new()
p4.user = "bruno"
p4:connect()
...
p4:disconnect()
```

p4.user -> string

Returns the current Helix server username.

```
p4 = P4.P4:new()
print( p4.user )
```

p4.version= string

Set the version of your script, as reported to the Helix server.

p4.version -> string

Get the version of your script, as reported to the Helix server.

p4.warnings -> table

Returns a table of warnings that arose during execution of the last command.

Class P4.Map

Description

The `P4.Map` class allows users to create and work with Helix server mappings without requiring a connection to a Helix Core server.

Class Methods

Map.join (map1, map2) -> Map

Join two `P4.Map` objects and create a third.

The new map is composed of the left-hand side of the first mapping, as joined to the right-hand side of the second mapping. For example:

```
# Map depot syntax to client syntax
client_map = P4.Map:new()
client_map:insert( "//depot/main/...", "//client/..." )

# Map client syntax to local syntax
client_root = P4.Map:new()
client_root:insert( "//client/...", "/home/bruno/workspace/..." )
```

```
# Join the previous mappings to map depot syntax to local syntax
local_map = P4::Map:new()
local_map = local_map:join( client_map, client_root )
local_path = local_map:translate( "//depot/main/www/index.html" )

# local_path is now /home/bruno/workspace/www/index.html
```

Instance Methods

map:clear() -> boolean

Empty a map.

map:count () -> number

Return the number of entries in a map.

map:includes (string) -> boolean

Tests whether a path is mapped or not.

map:insert(string, [string]) -> Map

Inserts an entry into the map.

May be called with one or two arguments. If called with one argument, the string is assumed to be a string containing either a half-map, or a string containing both halves of the mapping. In this form, mappings with embedded spaces must be quoted. If called with two arguments, each argument is assumed to be half of the mapping, and quotes are optional.

```
# called with two arguments:
map:insert( "//depot/main/...", "//client/..." )

# called with one argument containing both halves of the mapping:
map:insert( "//depot/live/... //client/live/..." )

# called with one argument containing a half-map:
# This call produces the mapping "depot/... depot/..."
map:insert( "depot/..." )
```

map:isempty() -> boolean

Test whether a map object is empty.

map:lhs() -> table

Returns the left side of a mapping as a table.

map:reverse()

Reverses the `P4.Map` object with the left and right sides of the mapping swapped.

map:rhs() -> table

Returns the right side of a mapping as a table.

map:to_a() -> table

Returns the map as a table.

map:translate(string, [boolean]) -> string

Translates a file path from one side of a mapping to the other. If the optional second argument is true, translate forward, and if it is false, translate in the reverse direction. By default, translation is in the forward direction.

Class P4.Message

Description

`P4.Message` objects contain error or other diagnostic messages from the Helix Core server. Retrieve them by using the `messages()` method.

Instance methods

message.severity() -> number

Severity of the message.

message.generic() -> number

Returns the generic class of the error.

message.msgid() -> number

Returns the unique ID of the message.

message.to_s() -> string

Converts the message into a string.

message.inspect() -> string

To facilitate debugging, returns a string that holds a formatted representation of the entire `P4.Message` object.

Class Helix.Core.Server

Class methods

ClientEditData(string, function) -> bool, string

Allow Lua writer using RunCommand to send an input form to the User for editing.

Returns:

`bool` - return `1` on success, otherwise `0`

`string` - with edited content of the form if success, with unedited content of the form containing an error message on failure

Example

```
local ret, formin = Helix.Core.Server.ClientEditData( formout,
Validator )
if ret then
Helix.Core.Server.ClientOutputText( "Form-in is:\n" )
Helix.Core.Server.ClientOutputText( formin )
Helix.Core.Server.ClientOutputText( ".... Done\n" )
else
Helix.Core.Server.ClientOutputText( "Client Edit error " .. formin ..
"\n")
end
```

GetArchDirFileName(string) -> string

Gets the path to a file under the server extension archive directory.

Returns the path to the file in the server extension's unpacked archive directory. This path is relative to the `server.extensions.dir` configurable.

GetArchiveFileInfo() -> string, string, string, number, string

Get information about the current file in the Archive() event

Returns:

string	string	string	number	string
name of the file	a string that represents the revision number of the file. For example, `"1.9876.1"` where: ■ `1.` is the conventional prefix to the changelist number ■ `9876` represents the changelist number ■ `.1` is the initial suffix that can increment to uniquely identify a shelved revision in Helix Core server version 2020.2 and later	file type	size of file in bytes	digest of file

GetDataDirFileName(string) -> string

Gets the path to a file under the server extension data directory.

Returns:

Passed a file name argument, returns the path in which the server extension will store the files it creates.

GetGlobalConfigData() -> table

Gets the global config table.

Returns:

A table of the values the user provided when filling out the global server extension config.

GetInstanceConfigData() -> table

Gets the instance config table.

Returns:

A table of field names and sample values used during the instance configuration of an server extension. This should be information the server extension needs for this instance of itself. It is the responsibility of the server extension to validate and format this data. Field names must not contain whitespace.

GetVar(string) -> string

Returns:
The value of the requested server extension session variable. See "Server extension session variables" on page 21.

GlobalConfigFields() -> table

Gets the global configuration setup.

Returns:

A table of field names and sample values used during the global configuration of a server extension . This should be information the server extension needs for all instances of itself. It is the responsibility of the server extension to validate and format this data. Field names must not contain whitespace.

InstanceConfigFields() -> table

Gets the instance configuration setup

Returns:

A table of field names and sample values used during configuration of an instance of a server extension . This should be information the server extension needs to perform specific actions, such as the list of file paths to monitor. It is the responsibility of the server extension to validate and format this data. Field names must not contain whitespace.

InstanceConfigEvents() -> table

Server event registration. See "Server extension callbacks" on page 17.

Returns:

A table of events and parameters. The parameters are likely dependent on the data retrieved from the instance configuration.

log(table)

Accepts a table of user data and appends it to the `log.json` file in the server extension data directory as line-delimited JSON.

SetArchiveFileSys() -> FileSys

For archive server extensions, give the server a `FileSys` instance to use for accessing the archive content.

Returns:

The `FileSys` object.

SetClientMsg(string)

Sets the message to be sent to the user.

Class Helix.Core.Server.MFA

This class exposes the Helix Core Server multifactor authentication properties.

Class properties

Status

Values are:

- SUCCESS
- FAILURE
- NO_MFA

Scheme

Values are:

- OTP_GENERATED - A one-time-password generated by a user device
- OTP_REQUESTED - A one-time-password sent to the user
- CHALLENGE - A challenge/response based on a token displayed to the user
- EXTERNAL - A request to a 3rd-party prompting method, like an app-based push notification

Class Helix.Core.Server.i18n

Class methods

GetLocale() -> string

Get the current locale for the user translations.

Returns a string.

SetLocale(string)

Set the locale for the current user translations.

Only necessary if overriding the defaults.

GetMessage(string, strings, ...) -> string

Get a translated message.

Parameters: string messageName, strings substitutions

Glossary

A

access level

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

admin access

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

APC

The Alternative PHP Cache, a free, open, and robust framework for caching and optimizing PHP intermediate code.

archive

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

atomic change transaction

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

avatar

A visual representation of a Swarm user or group. Avatars are used in Swarm to show involvement in or ownership of projects, groups, changelists, reviews, comments, etc. See also the "Gravatar" entry in this glossary.

B

base

For files: The file revision that contains the most common edits or changes among the file revisions in the source file and target file paths. For checked out streams: The public have version from which the checked out version is derived.

binary file type

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

branch

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

branch form

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

branch mapping

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

branch view

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

broker

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

C

change review

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

changelist

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

changelist form

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

changelist number

An integer that identifies a changelist. Submitted changelist numbers are ordinal (increasing), but not necessarily consecutive. For example, 103, 105, 108, 109. A pending changelist number might be assigned a different value upon submission.

check in

To submit a file to the Helix server depot.

check out

To designate one or more files, or a stream, for edit.

checkpoint

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

classic depot

A repository of Helix server files that is not streams-based. Uses the Perforce file revision model, not the graph model. The default depot name is depot. See also default depot, stream depot, and graph depot.

client form

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

client name

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

client root

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

client side

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

client workspace

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

code review

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

codeline

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

comment

Feedback provided in Helix Swarm on a changelist, review, job, or a file within a changelist or review.

commit server

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

conflict

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

copy up

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

counter

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

CSRF

Cross-Site Request Forgery, a form of web-based attack that exploits the trust that a site has in a user's web browser.

D

default changelist

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

deleted file

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

delta

The differences between two files.

depot

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

depot root

The topmost (root) directory for a depot.

depot side

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

depot syntax

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

diff

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

donor file

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

E

edge server

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

exclusionary access

A permission that denies access to the specified files.

exclusionary mapping

A view mapping that excludes specific files or directories.

extension

Similar to a trigger, but more modern. See "Helix Core Server Administrator Guide" on "Extensions".

F

file conflict

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

file pattern

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

file repository

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

file revision

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

file tree

All the subdirectories and files under a given root directory.

file type

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

fix

A job that has been closed in a changelist.

form

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

forwarding replica

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

G

Git Fusion

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

graph depot

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

group

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

H

have list

The list of file revisions currently in the client workspace.

head revision

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

heartbeat

A process that allows one server to monitor another server, such as a standby server monitoring the master server (see the p4 heartbeat command).

Helix server

The Helix server depot and metadata; also, the program that manages the depot and metadata, also called Helix Core server.

Helix TeamHub

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

Helix4Git

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

hybrid workspace

A workspace that maps to files stored in a depot of the classic Perforce file revision model as well as to files stored in a repo of the graph model associated with git.

I

iconv

A PHP extension that performs character set conversion, and is an interface to the GNU libiconv library.

integrate

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

J

job

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

job daemon

A program that checks the Helix server machine daily to determine if any jobs are open. If so, the daemon sends an email message to interested users, informing them the number of jobs in each category, the severity of each job, and more.

job specification

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

job view

A syntax used for searching Helix server jobs.

journal

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

journal rotation

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

journaling

The process of recording changes made to the Helix server's metadata.

L

label

A named list of user-specified file revisions.

label view

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

lazy copy

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

license file

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

list access

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

local depot

Any depot located on the currently specified Helix server.

local syntax

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

lock

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

log

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

M

mapping

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

MDS checksum

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

merge

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

merge file

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

metadata

The data stored by the Helix server that describes the files in the depot, the current state of client workspaces, protections, users, labels, and branches. Metadata is stored in the Perforce database and is separate from the archive files that users submit.

modification time or modtime

The time a file was last changed.

MPM

Multi-Processing Module, a component of the Apache web server that is responsible for binding to network ports, accepting requests, and dispatch operations to handle the request.

N

nonexistent revision

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

numbered changelist

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

O

opened file

A file you have checked out in your client workspace as a result of a Helix Core server operation (such as an edit, add, delete, integrate). Opening a file from your operating system file browser is not tracked by Helix Core server.

owner

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

P

p4

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

p4d

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

P4PHP

The PHP interface to the Helix API, which enables you to write PHP code that interacts with a Helix server machine.

PECL

PHP Extension Community Library, a library of extensions that can be added to PHP to improve and extend its functionality.

pending changelist

A changelist that has not been submitted.

Perforce

Perforce Software, Inc., a leading provider of enterprise-scale software solutions to technology developers and development operations (“DevOps”) teams requiring productivity, visibility, and scale during all phases of the development lifecycle.

project

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

protections

The permissions stored in the Helix server’s protections table.

proxy server

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

R

RCS format

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

read access

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

remote depot

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

replica

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

repo

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

reresolve

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

resolve

The process you use to manage the differences between two revisions of a file, or two versions of a stream. You can choose to resolve file conflicts by selecting the source or target file to be submitted, by merging the contents of conflicting files, or by making additional changes. To resolve stream conflicts, you can choose to accept the public source, accept the checked out target, manually accept changes, or combine path fields of the public and checked out version while accepting all other changes made in the checked out version.

reverse delta storage

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

revert

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

review access

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

review daemon

A program that periodically checks the Helix server machine to determine if any changelists have been submitted. If so, the daemon sends an email message to users who have subscribed to any of the files

included in those changelists, informing them of changes in files they are interested in.

revision number

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

revision range

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

revision specification

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

RPM

RPM Package Manager. A tool, and package format, for managing the installation, updates, and removal of software packages for Linux distributions such as Red Hat Enterprise Linux, the Fedora Project, and the CentOS Project.

S

server data

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

server root

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

service

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

shelve

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

status

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

storage record

An entry within the db.storage table to track references to an archive file.

stream

A "branch" with built-in rules that determines what changes should be propagated and in what order they should be propagated.

stream depot

A depot used with streams and stream clients. Has structured branching, unlike the free-form branching of a "classic" depot. Uses the Perforce file revision model, not the graph model. See also classic depot and graph depot.

stream hierarchy

The set of parent-to-child relationships between streams in a stream depot.

submit

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

super access

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

symlink file type

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

sync

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

T

target file

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

text file type

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

theirs

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

three-way merge

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

trigger

A script that is automatically invoked by Helix server when various conditions are met. (See "Helix Core Server Administrator Guide" on "Triggers".)

two-way merge

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

typemap

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

U

user

The identifier that Helix server uses to determine who is performing an operation. The three types of users are standard, service, and operator.

V

versioned file

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

view

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

W

wildcard

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

workspace

See client workspace.

workspace view

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

write access

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

X

XSS

Cross-Site Scripting, a form of web-based attack that injects malicious code into a user's web browser.

Y

yours

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

License Statements

To get a listing of the third-party software licenses that Helix Core Server uses, at the command line, type the p4 help legal command.

To get a listing of the third-party software licenses that the local client uses, at the command line, type the p4 help -l legal command.