HelixCore

Helix Core P4Ruby Developer Guide

2020.1

June 2020

PERFORCE

www.perforce.com

Contents

How to use this guide **4**

Syntax conventions 4

Feedback 4

Other documentation 5

P4Ruby **6**

System Requirements and Release Notes 6

Installing P4Ruby 6

Programming with P4Ruby 7

Connecting to SSL-enabled servers 8

P4Ruby classes 8

P4 8

P4Exception 12

P4::DepotFile 12

P4::Revision 12

P4::Integration 13

P4::Map 13

P4::MergeData 14

P4::Message 15

P4::OutputHandler 15

P4::Progress 15

P4::Spec 16

Class P4 16

Class P4Exception 37

Class P4::DepotFile 38

Class P4::Revision 38

Class P4::Integration 40

Class P4::Map 40

Class P4::MergeData 43

Class P4::Message 45

Class P4::OutputHandler 46

Class P4::Progress 47

Class P4::Spec 48

Glossary **50**

License Statements **69**

How to use this guide

This guide contains details about using the derived API for Ruby to create scripts that interact with Helix Core server. You can download the API from the Perforce web site. The derived API depends on the Helix C/C++ API. For details, see the*Helix Core C/C++ 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.
[`-f`]	The enclosed elements are 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*.
element1 | *element2*	Either *element1* or *element2* is required.

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.

To find earlier versions of this guide, use the following URL and replace *v17.1* with the version number you are looking for: https://www.perforce.com/manuals/*v17.1*/p4ruby/index.html

P4Ruby

P4Ruby is an extension to the Ruby programming language that allows you to run Helix Core server commands from within Ruby scripts, and get the results in a Ruby-friendly format.

The main features are:

- Get Helix server data and forms in hashes and arrays.
- Edit Helix server forms by modifying hashes.
- Exception based error handling.
- Controllable handling of warnings such as "File(s) up-to-date." on a sync.
- Run as many commands on a connection as required.
- The output of a command is returned as a Ruby array. For non-tagged output, the elements of the array are strings. For tagged output, the elements of the array are Ruby hashes. For forms, the output is an array of `P4::Spec` objects.
- Thread-safe and thread-friendly; you can have multiple instances of the `P4` class running in different threads.
- Exception-based error handling. Trap `P4Exception`s for complete, high-level error handling.

System Requirements and Release Notes

P4Ruby is supported on Windows, Linux, and OS X.

For system requirements, see the release notes at https://www.perforce.com/perforce/doc.current/user/p4rubynotes.txt.

> **Note**
> When passing arguments, make sure to omit the space between the argument and its value, such as in the value pair `-u` and ***`username`*** in the following example:
>
> ```
> change = p4.run_changes("-uusername", "-m1")[0]
> ```
>
> If you include a space (`"-u username"`), the command fails.

Installing P4Ruby

As of version 2015.1, the recommended mechanism for installing P4Ruby is via gems.

Outside of Windows, the `p4ruby` gem installs must be compiled locally against your installation of Ruby. If you can build the core Ruby distribution locally, you likely can install P4Ruby without incident. On Windows, precompiled gems will be made available.

```
$ gem install p4ruby
```

The gem will attempt to download a correct version of the Helix C/C++ API from ftp.perforce.com.

Helix C/C++ API can be specified with the `--with-p4api-dir` option. The Helix C/C++ API should match the major and minor version of P4Ruby.

Download from:

- https://www.perforce.com/downloads/helix-core-api-ruby

  or

- https://github.com/perforce/p4ruby

Programming with P4Ruby

The following example shows how to create a new client workspace based on an existing template:

```
require "P4"
template = "my-client-template"
client_root = 'c:\p4-work'
p4 = P4.new
p4.connect

begin

    # Run a "p4 client -t template -o" and convert it into a Ruby hash
    spec = p4.fetch_client( "-t", template, "my-new-client")

    # Now edit the fields in the form
    spec[ "Root" ]  = client_root
    spec[ "Options" ] = spec[ "Options" ].sub( "normdir", "rmdir" )

    # Now save the updated spec
    p4.save_client( spec )

    # Point to the newly-created client
    p4.client="my-new-client"

    # And sync it.
    p4.run_sync
```

```
rescue P4Exception
    # If any errors occur, we'll jump in here. Just log them
    # and raise the exception up to the higher level

    p4.errors.each { |e| $stderr.puts( e ) }
    raise

end
```

Connecting to SSL-enabled servers

Scripts written with P4Ruby use any existing `P4TRUST` file present in their operating environment (by default, `.p4trust` in the home directory of the user that runs the script).

If the fingerprint returned by the server fails to match the one installed in the `P4TRUST` file associated with the script's run-time environment, your script will (and should!) fail to connect to the server.

P4Ruby classes

The P4 module consists of several public classes:

- P4
- P4Exception
- P4::DepotFile
- P4::Revision
- P4::Integration
- P4::Map
- P4::MergeData
- P4::Message
- P4::OutputHandler
- P4::Progress
- P4::Spec

The following tables provide brief details about each public class.

P4

The main class used for executing Perforce commands. Almost everything you do with P4Ruby will involve this class.

Method	Description
`identify`	Return the version of P4Ruby in use (class method).
`new`	Construct a new **`P4`** object (class method).
`api_level=`	Set desired API compatibility level.
`api_level`	Return current API compatibility level.
`at_exception_level`	Execute the associated block under a specific exception level, returning to previous exception level when block returns.
`charset=`	Set character set when connecting to Unicode servers.
`charset`	Get character set when connecting to Unicode servers.
`client=`	Set client workspace (**`P4CLIENT`**).
`client`	Get current client workspace (**`P4CLIENT`**).
`connect`	Connect to the Helix Core server, raise **`P4Exception`** on failure.
`connected?`	Test whether or not session has been connected and/or has been dropped.
`cwd=`	Set current working directory.
`cwd`	Get current working directory.
`delete_<spectype>`	Shortcut methods for deleting clients, labels, etc.
`disconnect`	Disconnect from the Helix Core server.
`each_<spectype>`	Shortcut methods for iterating through clients, labels, etc.
`env`	Get the value of a Perforce environment variable, taking into account **`P4CONFIG`** files and (on Windows or OS X) the registry or user preferences.
`errors`	Return the array of errors that occurred during execution of previous command.
`exception_level=`	Control which types of events give rise to exceptions (**`P4::RAISE_NONE`**, **`RAISE_ERRORS`**, or **`RAISE_ALL`**).
`exception_level`	Return the current exception level.
`fetch_<spectype>`	Shortcut methods for retrieving the definitions of clients, labels, etc.

Method	Description
`format_spec`	Convert fields in a hash containing the elements of a Perforce form (spec) into the string representation familiar to users.
`format_<spectype>`	Shortcut method; equivalent to: `p4.format_spec( "<spectype>", aHash )`
`handler=`	Set output handler.
`handler`	Get output handler.
`host=`	Set the name of the current host (`P4HOST`).
`host`	Get the current hostname.
`input=`	Store input for next command.
`maxlocktime=`	Set MaxLockTime used for all following commands.
`maxlocktime`	Get MaxLockTime used for all following commands.
`maxresults=`	Set MaxResults used for all following commands.
`maxresults`	Get MaxResults used for all following commands.
`maxscanrows=`	Set MaxScanRows used for all following commands.
`maxscanrows`	Get MaxScanRows used for all following commands.
`messages`	Returns all messages from the server as `P4::Message` objects.
`p4config_file`	Get the location of the configuration file used (`P4CONFIG`).
`parse_<spectype>`	Shortcut method; equivalent to: `p4.parse_spec( "<spectype>", aString )`
`parse_spec`	Parses a Perforce form (spec) in text form into a Ruby hash using the spec definition obtained from the server.
`password=`	Set Perforce password (`P4PASSWD`).
`password`	Get the current password or ticket.
`port=`	Set host and port (`P4PORT`).
`port`	Get host and port (`P4PORT`) of the current Perforce server.
`prog=`	Set program name as shown by `p4 monitor show -e`.

Method	Description
`prog`	Get program name as shown by **`p4 monitor show -e`**.
`progress=`	Set progress indicator.
`progress`	Get progress indicator.
`run_cmd`	Shortcut method; equivalent to: `p4.run( "cmd", arguments... )`
`run`	Runs the specified Perforce command with the arguments supplied.
`run_filelog`	Runs a **`p4 filelog`** on the fileSpec provided, returns an array of **`P4::DepotFile`** objects.
`run_login`	Runs **`p4 login`** using a password or ticket set by the user.
`run_password`	A thin wrapper to make it easy to change your password.
`run_resolve`	Interface to **`p4 resolve`**.
`run_submit`	Submit a changelist to the server.
`run_tickets`	Get a list of tickets from the local tickets file.

`save_<spectype>`	Shortcut method; equivalent to:
	`p4.input = hashOrString`
	`p4.run( "<spectype>", "-i" )`

`server_case_sensitive?`	Detects whether or not the server is case sensitive.
`server_level`	Returns the current Perforce server level.
`server_unicode?`	Detects whether or not the server is in unicode mode.
`set_env`	On Windows or OS X, set a variable in the registry or user preferences.
`streams=`	Enable or disable support for streams.
`streams?`	Test whether or not the server supports streams
`tagged`	Toggles tagged output (true or false). By default, tagged output is on.
`tagged=`	Sets tagged output. By default, tagged output is on.
`tagged?`	Detects whether or not tagged output is enabled.
`ticketfile=`	Set the location of the **`P4TICKETS`** file.

Method	Description
`ticketfile`	Get the location of the `P4TICKETS` file.
`track=`	Activate or disable server performance tracking.
`track?`	Detect whether server performance tracking is active.
`track_output`	Returns server tracking output.
`user=`	Set the Perforce username (`P4USER`).
`user`	Get the Perforce username (`P4USER`).
`version=`	Set your script's version as reported to the server.
`version`	Get your script's version as reported by the server.
`warnings`	Returns the array of warnings that arose during execution of the last command.

P4Exception

Used as part of error reporting and is derived from the Ruby `RuntimeError` class.

P4::DepotFile

Utility class allowing access to the attributes of a file in the depot. Returned by `P4#run_filelog()`.

Method	Description
`depot_file`	Name of the depot file to which this object refers.
`each_revision`	Iterates over each revision of the depot file.
`revisions`	Returns an array of revision objects for the depot file.

P4::Revision

Utility class allowing access to the attributes of a revision `P4::DepotFile` object. Returned by `P4#run_filelog()`.

Method	Description
`action`	Action that created the revision.
`change`	Changelist number.

Method	Description
`client`	Client workspace used to create this revision.
`depot_file`	Name of the file in the depot.
`desc`	Short changelist description.
`digest`	MD5 digest of this revision.
`filesize`	Returns the size of this revision.
`integrations`	Array of `P4::Integration` objects.
`rev`	Revision number.
`time`	Timestamp.
`type`	Perforce file type.
`user`	User that created this revision.

P4::Integration

Utility class allowing access to the attributes of an integration record for a `P4::Revision` object. Returned by `P4#run_filelog()`.

Method	Description
`how`	Integration method (merge/branch/copy/ignored).
`file`	Integrated file.
`srev`	Start revision.
`erev`	End revision.

P4::Map

A class that allows users to create and work with Perforce mappings without requiring a connection to the Helix Core server.

Method	Description
`new`	Construct a new map object (class method).
`join`	Joins two maps to create a third (class method).
`clear`	Empties a map.

Method	Description
`count`	Returns the number of entries in a map.
`empty?`	Tests whether or not a map object is empty.
`insert`	Inserts an entry into the map.
`translate`	Translate a string through a map.
`includes?`	Tests whether a path is mapped.
`reverse`	Returns a new mapping with the left and right sides reversed.
`lhs`	Returns the left side as an array.
`rhs`	Returns the right side as an array.
`to_a`	Returns the map as an array.

P4::MergeData

Class encapsulating the context of an individual merge during execution of a **`p4 resolve`** command. Passed as a parameter to the block passed to **`P4#run_resolve()`**.

Method	Description
`your_name`	Returns the name of "your" file in the merge. (file in workspace)
`their_name`	Returns the name of "their" file in the merge. (file in the depot)
`base_name`	Returns the name of "base" file in the merge. (file in the depot)
`your_path`	Returns the path of "your" file in the merge. (file in workspace)
`their_path`	Returns the path of "their" file in the merge. (temporary file on workstation into which **`their_name`** has been loaded)
`base_path`	Returns the path of the base file in the merge. (temporary file on workstation into which **`base_name`** has been loaded)
`result_path`	Returns the path to the merge result. (temporary file on workstation into which the automatic merge performed by the server has been loaded)
`merge_hint`	Returns hint from server as to how user might best resolve merge.

Method	Description
`run_merge`	If the environment variable `P4MERGE` is defined, run it and return a boolean based on the return value of that program.

P4::Message

Utility class allowing access to the attributes of a message object returned by `P4#messages()`.

Method	Description
`severity`	Returns the severity of the message.
`generic`	Returns the generic class of the error.
`msgid`	Returns the unique ID of the error message.
`to_s`	Returns the error message as a string.
`inspect`	Converts the error object into a string for debugging purposes.

P4::OutputHandler

Handler class that provides access to streaming output from the server; set `P4#handler()` to an instance of a subclass of `P4::OutputHandler` to enable callbacks:

Method	Description
`outputBinary`	Process binary data.
`outputInfo`	Process tabular data.
`outputMessage`	Process information or errors.
`outputStat`	Process tagged output.
`outputText`	Process text data.

P4::Progress

Handler class that provides access to progress indicators from the server; set `P4#progress()` to an instance of a subclass of `P4::Progress` with the following methods (even if the implementations are empty) to enable callbacks:

Method	Description
`init`	Initialize progress indicator as designated type.
`total`	Total number of units (if known).
`description`	Description and type of units to be used for progress reporting.
`update`	If non-zero, user has requested a cancellation of the operation.
`done`	If non-zero, operation has failed.

P4::Spec

Subclass of hash allowing access to the fields in a Perforce specification form. Also checks that the fields that are set are valid fields for the given type of spec. Returned by `P4#fetch__<spectype>_()`.

Method	Description
`spec._fieldname`	Return the value associated with the field named *fieldname*.
`spec._fieldname=`	Set the value associated with the field named *fieldname*.
`spec.permitted_fields`	Returns an array containing the names of fields that are valid in this spec object.

Class P4

Main interface to the Helix server client API. Each `P4` object provides you with a thread-safe API level interface to Helix server. The basic model is to:

1. Instantiate your `P4` object.
2. Specify your Helix server client environment.
   - `client`
   - `host`
   - `password`
   - `port`
   - `user`
3. Set any options to control output or error handling:
   - `exception_level`
4. Connect to the Perforce service.

The Helix server protocol is not designed to support multiple concurrent queries over the same connection. Multithreaded applications that use the C++ API or derived APIs (including P4Ruby) should ensure that a separate connection is used for each thread, or that only one thread may use a shared connection at a time.

5. Run your Helix server commands.
6. Disconnect from the Perforce service.

Class Methods

P4.identify -> aString

Return the version of P4Ruby that you are using. Also reports the version of the OpenSSL library used for building the underlying Helix C/C++ API with which P4Ruby was built.

```
ruby -rP4 -e 'puts( P4.identify )'
```

Some of this information is already made available through the predefined constants **`P4::VERSION`**, **`P4::OS`**, and **`P4::PATCHLEVEL`**.

P4.new -> aP4

Constructs a new P4 object.

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

Instance Methods

p4.api_level= anInteger -> anInteger

Sets the API compatibility level desired. This is useful when writing scripts using Helix server commands that do not yet support tagged output. In these cases, upgrading to a later server that supports tagged output for the commands in question can break your script. 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 *must* be called prior to calling **`P4#connect()`**.

```
p4 = P4.new
p4.api_level = 67 # Lock to 2010.1 format
p4.connect
...
```

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

p4.api_level -> anInteger

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:

http://kb.perforce.com/article/512

p4.at_exception_level(lev) { ... } -> self

Executes the associated block under a specific exception level. Returns to the previous exception level when the block returns.

```
p4 = P4.new
p4.client = "www"
p4.connect

p4.at_exception_level( P4::RAISE_ERRORS ) do
    p4.run_sync
end

p4.disconnect
```

p4.charset= aString -> aString

Sets the character set to use when connect to a Unicode enabled server. Do not use when working with non-Unicode-enabled servers. By default, the character set is the value of the **`P4CHARSET`** environment variable. If the character set is invalid, this method raises a **`P4Exception`**.

```
p4 = P4.new
p4.client = "www"
p4.charset = "iso8859-1"
p4.connect
p4.run_sync
p4.disconnect
```

p4.charset -> aString

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

```
p4 = P4.new
p4.charset = "utf8"
puts( p4.charset )
```

p4.client= aString -> aString

Set the name of the client workspace you wish to use. If not called, defaults to the value of **`P4CLIENT`** taken from any **`P4CONFIG`** file present, or from the environment as per the usual Helix server convention. Must be called before connecting to the Helix server.

```
p4 = P4.new
p4.client = "www"
p4.connect
p4.run_sync
p4.disconnect
```

p4.client -> aString

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

```
p4 = P4.new
puts( p4.client )
```

p4.connect -> aBool

Connect to the Helix Core server. You must connect before you can execute commands. Raises a **`P4Exception`** if the connection attempt fails.

```
p4 = P4.new
p4.connect
```

p4.connected? -> aBool

Test whether or not the session has been connected, and if the connection has not been dropped.

```
p4 = P4.newp4.connected?
```

p4.cwd= aString -> aString

Sets the current working directly. Can be called prior to executing any Helix server command. Sometimes necessary if your script executes a **`chdir()`** as part of its processing.

```
p4 = P4.new
p4.cwd = "/home/bruno"
```

p4.cwd -> aString

Get the current working directory.

```
p4 = P4.new
puts( p4.cwd )
```

p4.delete_<spectype>([options], name) -> anArray

The delete methods are simply shortcut methods that allow you to quickly delete the definitions of clients, labels, branches, etc. These methods are equivalent to:

```
p4.run( "<spectype>", '-d', [options], "spec name" )
```

For example:

```
require "P4"
require "parsedate"
include ParseDate
now = Time.now
p4 = P4.new
begin
  p4.connect
  p4.run_clients.each do
    |client|
    atime = parsedate( client[ "Access" ] )
    if( (atime + 24 * 3600 * 365 ) < now )
      p4.delete_client( '-f', client[ "client" ] )
    end
  end
rescue P4Exception
  p4.errors.each { |e| puts( e ) }
ensure
  p4.disconnect
end
```

p4.disconnect -> true

Disconnect from the Helix Core server.

```
p4 = P4.new
p4.connect
p4.disconnect
```

p4.each_<spectype<(arguments) -> anArray

The **`each_<spectype>`** methods are shortcut methods that allow you to quickly iterate through clients, labels, branches, etc. Valid <*spectype*>s are **`clients`**, **`labels`**, **`branches`**, **`changes`**, **`streams`**, **`jobs`**, **`users`**, **`groups`**, **`depots`** and **`servers`**. Valid arguments are any arguments that would be valid for the corresponding **`run_<spectype>`** command.

For example, to iterate through clients:

```
p4.each_clients do
|c|
  # work with the retrieved client spec
end
```

is equivalent to:

```
clients = p4.run_clients
clients.each do
|c|
  client = p4.fetch_client( c['client'] )
  # work with the retrieved client spec
end
```

p4.env -> 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.new
puts p4.env( "P4PORT" )
```

p4.errors -> anArray

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

```
p4 = P4.new
begin
  p4.connect
  p4.exception_level( P4::RAISE_ERRORS ) # ignore "File(s) up-to-date"
  files = p4.run_sync
```

```
rescue P4Exception
  p4.errors.each { |e| puts( e ) }
ensure
  p4.disconnect
end
```

p4.exception_level= anInteger -> anInteger

Configures the events which give rise to exceptions. The following three levels are supported:

- **`P4::RAISE_NONE`** disables all exception raising and makes the interface completely procedural.
- **`P4::RAISE_ERRORS`** causes exceptions to be raised only when errors are encountered.
- **`P4::RAISE_ALL`** causes exceptions to be raised for both errors and warnings. This is the default.

```
p4 = P4.new
p4.exception_level = P4::RAISE_ERRORS
p4.connect   # P4Exception on failure
p4.run_sync  # File(s) up-to-date is a warning so no exception is
raised
p4.disconnect
```

p4.exception_level -> aNumber

Returns the current exception level.

p4.fetch_<spectype>([name]) -> aP4::Spec

The **`fetch_<spectype>`** methods are shortcut methods that allow you to quickly fetch the definitions of clients, labels, branches, etc. They're equivalent to:

```
p4.run( "<spectype>", '-o', ... ).shift
```

For example:

```
p4 = P4.new
begin
  p4.connect
  client       = p4.fetch_client()
  other_client = p4.fetch_client( "other" )
  label        = p4.fetch_label( "somelabel" )
```

```
rescue P4Exception
  p4.errors.each { |e| puts( e ) }
ensure
  p4.disconnect
end
```

p4.format_spec("<spectype>", aHash)-> aString

Converts the fields in a hash 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 hash to parse.

There are shortcuts available for this method. You can use:

```
p4.format_<spectype>( hash )
```

instead of:

```
p4.format_spec( "<spectype>", hash )
```

where <*spectype*> is the name of a Helix server spec, such as **`client`**, **`label`**, etc.

p4.format_<spectype> aHash -> aHash

The **`format_<spectype>`** methods are shortcut methods that allow you to quickly fetch the definitions of clients, labels, branches, etc. They're equivalent to:

```
p4.format_spec( "<spectype>", aHash )
```

p4.graph= -> aBool

Enable or disable support for graph depots. By default, support for depots of type graph is enabled at 2017.1 or higher (**`P4#api_level()`** >= 82). Raises a **`P4Exception`** if you attempt to enable graph depots on a pre-2017.1 server. You can enable or disable support for graph depots both before and after connecting to the server.

```
p4 = P4.new
p4.graph = false
```

p4.graph? -> aBool

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

```
p4 = P4.new
puts ( p4.graph? )
```

```
p4.graph = false
puts ( p4.graph? )
```

p4.handler= aHandler -> aHandler

Set the current output handler. This should be a subclass of **`P4::OutputHandler`**.

p4.handler -> aHandler

Get the current output handler.

p4.host= aString -> aString

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. Must be called before connecting to the Helix server.

```
p4 = P4.new
p4.host = "workstation123.perforce.com"
p4.connect
...
p4.disconnect
```

p4.host -> aString

Get the current hostname.

```
p4 = P4.new
puts( p4.host )
```

p4.input= (aString|aHash|anArray) -> aString|aHash|anArray

Store input for the next command.

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. Typically, commands of the form **`p4 cmd -i`** are invoked using the **`P4#save_<spectype>()`** methods, which call **`P4#input()`** internally; there is no need to call **`P4#input()`** when using the **`P4#save_<spectype>()`** shortcuts.

You may pass a string, a hash, or (for commands that take multiple inputs from the user) an array of strings or hashes. If you pass an array, note that the array will be shifted each time Helix server asks the user for input.

```
p4 = P4.new
p4.connect
```

```
change = p4.run_change( "-o" ).shift
change[ "Description" ] = "Autosubmitted changelist"

p4.input = change
p4.run_submit( "-i" )

p4.disconnect
```

p4.maxlocktime= anInteger -> anInteger

Limit the amount of time (in milliseconds) spent during data scans to prevent the server from locking tables for too long. Commands that take longer than the limit will be aborted. The limit remains in force until you disable it by setting it to zero. See **`p4 help maxlocktime`** for information on the commands that support this limit.

```
p4 = P4.new
begin
  p4.connect
  p4.maxlocktime = 10000 # 10 seconds
  files = p4.run_sync
rescue P4Exception => ex
  p4.errors.each { |e| $stderr.puts( e ) }
ensure
  p4.disconnectend
```

p4.maxlocktime -> anInteger

Get the current **`maxlocktime`** setting.

```
p4 = P4.new
puts( p4.maxlocktime )
```

p4.maxresults= anInteger -> anInteger

Limit the number of results Helix server permits for subsequent commands. Commands that produce more than this number of results will be aborted. The limit remains in force until you disable it by setting it to zero. See **`p4 help maxresults`** for information on the commands that support this limit.

```
p4 = P4.new
begin
```

```
  p4.connect
  p4.maxresults = 100
  files = p4.run_sync
rescue P4Exception => ex
  p4.errors.each { |e| $stderr.puts( e ) }
ensure
  p4.disconnect
end
```

p4.maxresults -> anInteger

Get the current **`maxresults`** setting.

```
p4 = P4.new
puts( p4.maxresults )
```

p4.maxscanrows= anInteger -> anInteger

Limit the number of database records Helix server will scan for subsequent commands. Commands that attempt to scan more than this number of records will be aborted. The limit remains in force until you disable it by setting it to zero. See **`p4 help maxscanrows`** for information on the commands that support this limit.

```
p4 = P4.new
begin
  p4.connect
  p4.maxscanrows = 100
  files = p4.run_sync
rescue P4Exception => ex
  p4.errors.each { |e| $stderr.puts( e ) }
ensure
  p4.disconnectend
```

p4.maxscanrows -> anInteger

Get the current **`maxscanrows`** setting.

```
p4 = P4.new
puts( p4.maxscanrows )
```

p4.messages -> aP4::Message

Returns a message from the Helix server in the form of a **`P4::Message`** object.

```
p4 = P4.new
p4.exception_level = P4::RAISE_NONE
p4.run_sync
p4.run_sync    # this second sync should return "File(s) up-to-date."
w = p4.messages[0]
puts ( w.to_s )
```

p4.p4config_file -> aString

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

```
p4 = P4.new
puts( p4.p4config_file )
```

p4.parse_<spectype>(aString) -> aP4::Spec

This is equivalent to:

```
p4.parse_spec( "<spectype>", aString )
```

p4.parse_spec("<spectype>", aString) -> aP4::Spec

Parses a Helix server form (spec) in text form into a Ruby hash 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.

Note that there are shortcuts available for this method. You can use:

```
p4.parse_<spectype>( buf )
```

instead of:

```
p4.parse_spec( "<spectype>", buf )
```

Where <*spectype*> is one of **`client`**, **`branch`**, **`label`**, and so on.

p4.password= aString -> aString

Set your Helix server password, in plain text. If not used, takes the value of **`P4PASSWD`** from any **`P4CONFIG`** file in effect, or from the environment according to the normal Helix server conventions. This password will also be used if you later call **`p4.run_login`** to login using the 2003.2 and later ticket system.

```
p4 = P4.new
p4.password = "mypass"
p4.connect
p4.run_login
```

p4.password -> aString

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.new
puts( p4.password )
```

p4.port= aString -> aString

Set the host and port of the Helix server you want to connect to. If not called, defaults to the value of **`P4PORT`** in any **`P4CONFIG`** file in effect, and then to the value of **`P4PORT`** taken from the environment.

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

p4.port -> aString

Get the host and port of the current Helix server.

```
p4 = P4.new
puts( p4.port )
```

p4.prog= aString -> aString

Set the name of the program, as reported to Helix server system administrators running **`p4 monitor show -e`** in Helix server 2004.2 or later releases.

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

p4.prog -> aString

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

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

p4.progress= aProgress -> aProgress

Set the current progress indicator. This should be a subclass of **`P4::Progress`**.

p4.progress -> aProgress

Get the current progress indicator.

p4.run_<cmd>(arguments) -> anArray

This is equivalent to:

```
p4.run( "cmd", arguments... )
```

p4.run(aCommand, arguments...) -> anArray

Base interface to all the run methods in this API. Runs the specified Helix server command with the arguments supplied. Arguments may be in any form as long as they can be converted to strings by **`to_s`**. However, each command's options 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")
```

Failing to pass options in this way can result in confusing error messages.

The **`P4#run()`** method returns an array of results whether the command succeeds or fails; the array may, however, be empty. Whether the elements of the array are strings or hashes 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, **`P4#run()`** will raise a **`P4Exception`**. If the current exception level is below the threshold for the error/warning, **`P4#run()`** returns the output as normal and the caller must explicitly review **`P4#errors()`** and **`P4#warnings()`** to check for errors or warnings.

```
p4 = P4.new
p4.connect
spec = p4.run( "client", "-o" ).shift
p4.disconnect
```

Shortcuts are available for **`P4#run()`**. For example:

```
p4.run_command( args )
```

is equivalent to:

```
p4.run( "command", args )
```

There are also some shortcuts for common commands such as editing Helix server forms and submitting. Consequently, this:

```
p4 = P4.new
p4.connect
clientspec = p4.run_client( "-o" ).shift
clientspec[ "Description" ] = "Build client"
p4.input = clientspec
p4.run_client( "-i" )
p4.disconnect
```

may be shortened to:

```
p4 = P4.new
p4.connect
clientspec = p4.fetch_client
clientspec[ "Description" ] = "Build client"
p4.save_client( clientspec )
p4.disconnect
```

The following are equivalent:

`p4.delete_<spectype>()`	**`p4.run( "<spectype>", "-d" )`**
`p4.fetch_<spectype>()`	**`p4.run( "<spectype>", "-o" ).shift`**
`p4.save_<spectype>( spec )`	**`p4.input = specp4.run( "<spectype>", "-i" )`**

As the commands associated with **`P4#fetch_<spectype>()`** typically return only one item, these methods do not return an array, but instead return the first result element.

For convenience in submitting changelists, changes returned by **`P4#fetch_change()`** can be passed to **`P4#run_submit`**. For example:

```
p4 = P4.new
p4.connect
spec = p4.fetch_changespec[ "Description" ] = "Automated change"
p4.run_submit( spec )
p4.disconnect
```

p4.run_filelog(fileSpec) -> anArray

Runs a **`p4 filelog`** on the **`fileSpec`** provided and returns an array of **`P4::DepotFile`** results when executed in tagged mode, and an array of strings when executed in non-tagged mode. By default, the raw output of **`p4 filelog`** is tagged; this method restructures the output into a more user-friendly (and object-oriented) form.

```
p4 = P4.new
begin
  p4.connect
  p4.run_filelog( "index.html" ).shift.each_revision do
    |r|
    r.each_integration do
      |i|
      # Do something
    end
  end
rescue P4Exception
  p4.errors.each { |e| puts( e ) }
ensure
  p4.disconnect
end
```

p4.run_login(arg...) -> anArray

Runs **`p4 login`** using a password or ticket set by the user.

p4.run_password(oldpass, newpass) -> anArray

A thin wrapper to make it easy to change your password. This method is (literally) equivalent to the following code:

```
p4.input( [ oldpass, newpass, newpass ] )
p4.run( "password" )
```

For example:

```
p4 = P4.new
p4.password = "myoldpass"
begin
  p4.connect
  p4.run_password( "myoldpass", "mynewpass" )
```

```
rescue P4Exception
  p4.errors.each { |e| puts( e ) }
ensure
  p4.disconnect
end
```

p4.run_resolve(args) [block] -> anArray

Interface to **`p4 resolve`**. Without a block, simply runs a non-interactive resolve (typically an automatic resolve).

```
p4.run_resolve( "-at" )
```

When a block is supplied, the block is invoked once for each merge scheduled by Helix server. For each merge, a **`P4::MergeData`** object is passed to the block. This object contains the context of the merge.

The block determines the outcome of the merge by evaluating to one of the following strings:

Block string	Meaning
`ay`	Accept Yours.
`at`	Accept Theirs.
`am`	Accept Merge result.
`ae`	Accept Edited result.
`s`	Skip this merge.
`q`	Abort the merge.

For example:

```
p4.run_resolve() do
  |md|
  puts( "Merging..." )
  puts( "Yours: #{md.your_name}" )
  puts( "Theirs: #{md.their_name}" )
  puts( "Base: #{md.base_name}" )
  puts( "Yours file: #{md.your_path}" )
  puts( "Theirs file: #{md.their_path}" )
  puts( "Base file: #{md.base_path}" )
  puts( "Result file: #{md.result_path}" )
  puts( "Merge Hint: #{md.merge_hint}" )
```

```
  result = md.merge_hint
  if( result == "e" )
      puts( "Invoking external merge application" )
      result = "s"    # If the merge doesn't work, we'll skip
      result = "am" if md.run_merge()
  end
  result
end
```

p4.run_submit([aHash], [arg...]) -> anArray

Submit a changelist to the server. To submit a changelist, set the fields of the changelist as required and supply any flags:.

```
change = p4.fetch_change
change._description = "Some description"
p4.run_submit( "-r", change )
```

You can also submit a changelist by supplying the arguments as you would on the command line:

```
p4.run_submit( "-d", "Some description", "somedir/..." )
```

p4.run_tickets( ) -> anArray

Get a list of tickets from the local tickets file. Each ticket is a hash object with fields for **`Host`**, **`User`**, and **`Ticket`**.

p4.save_<spectype>(hashOrString, [options]) -> anArray

The **`save_<spectype>`** methods are shortcut methods that allow you to quickly update the definitions of clients, labels, branches, etc. They are equivalent to:

```
p4.input = hashOrStringp4.run( "<spectype>", "-i" )
```

For example:

```
p4 = P4.new
begin
  p4.connect
  client            = p4.fetch_client()
  client[ "Owner" ] = p4.user
  p4.save_client( client )
```

```
rescue P4Exception
  p4.errors.each { |e| puts( e ) }
ensure
  p4.disconnect
end
```

p4.server_case_sensitive? -> aBool

Detects whether or not the server is case-sensitive.

p4.server_level -> anInteger

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? -> aBool

Detects whether or not the server is in unicode mode.

p4.set_env= (aString, aString) -> aBool

On Windows or OS X, set a variable in the registry or user preferences. To unset a variable, pass an empty string as the second argument. On other platforms, an exception is raised.

```
p4 = P4.new
p4.set_env = ( "P4CLIENT", "my_workspace" )
p4.set_env = ( "P4CLIENT", "" )
```

p4.streams= -> aBool

Enable or disable support for streams. By default, streams support is enabled at 2011.1 or higher (**`P4#api_level()`** >= 70). Raises a **`P4Exception`** if you attempt to enable streams on a pre-2011.1 server. You can enable or disable support for streams both before and after connecting to the server.

```
p4 = P4.new
p4.streams = false
```

p4.streams? -> aBool

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

```
p4 = P4.new
puts ( p4.streams? )
p4.streams = false
puts ( p4.streams? )
```

p4.tagged(aBool) { block }

Temporarily toggles the use of tagged output for the duration of the block, and then resets it when the block terminates.

p4.tagged= aBool -> aBool

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

```
p4 = P4.new
p4.tagged = false
```

p4.tagged? -> aBool

Detects whether or not you are in tagged mode.

```
p4 = P4.new
puts ( p4.tagged? )
p4.tagged = false
puts ( p4.tagged? )
```

p4.ticketfile= aString -> aString

Sets the location of the `P4TICKETS` file.

```
p4 = P4.new
p4.ticketfile = "/home/bruno/tickets"
```

p4.ticketfile -> aString

Get the path to the current `P4TICKETS` file.

```
p4 = P4.new
puts( p4.ticketfile )
```

p4.track= -> aBool

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

```
p4 = P4.new
p4.track = true
```

p4.track? -> aBool

Detects whether or not performance tracking is enabled.

```
p4 = P4.new
p4.track = true
puts ( p4.track? )
p4.track = false
puts ( p4.track? )
```

p4.track_output -> anArray

If performance tracking is enabled with **`p4.track=`**, returns a list of strings corresponding to the performance tracking output for the most recently-executed command.

```
p4 = P4.new
p4.track = true
p4.run_info
puts ( p4.track_output[0].slice(0,3) )    # should be "rpc"
```

p4.user= aString -> aString

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. Must be called before connecting to the Helix server.

```
p4 = P4.new
p4.user = "bruno"
p4.connect
...
p4.disconnect
```

p4.user -> aString

Returns the current Helix server username.

```
p4 = P4.new
puts( p4.user )
```

p4.version= aString -> aString

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

p4.version -> aString

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

p4.warnings -> anArray

Returns the array of warnings that arose during execution of the last command.

```
p4 = P4.new
begin
  p4.connect
  p4.exception_level( P4::RAISE_ALL ) # File(s) up-to-date is a warning
  files = p4.run_sync
rescue P4Exception => ex
  p4.warnings.each { |w| puts( w ) }
ensure
  p4.disconnect
end
```

Class P4Exception

Shallow subclass of **`RuntimeError`** to be used for catching Helix server-specific errors. Doesn't contain any extra information. See **`P4#errors()`** and **`P4#warnings`** for details of the errors giving rise to the exception.

Class Methods

None.

Instance Methods

None.

Class P4::DepotFile

Description

Utility class providing easy access to the attributes of a file in a Helix server depot. Each `P4::DepotFile` object contains summary information about the file, and a list of revisions (`P4::Revision` objects) of that file. Currently, only the `P4#run_filelog()` method returns an array of `P4::DepotFile` objects.

Class Methods

None.

Instance Methods

df.depot_file -> aString

Returns the name of the depot file to which this object refers.

df.each_revision { |rev| block } -> revArray

Iterates over each revision of the depot file.

df.revisions -> aArray

Returns an array of revisions of the depot file.

Class P4::Revision

Description

Utility class providing easy access to the revisions of a file in a Helix server depot. `P4::Revision` objects can store basic information about revisions and a list of the integrations for that revision. Created by `P4#run_filelog()`.

Class Methods

None.

Instance Methods

rev.action -> aString

Returns the name of the action which gave rise to this revision of the file.

rev.change -> aNumber

Returns the change number that gave rise to this revision of the file.

rev.client -> aString

Returns the name of the client from which this revision was submitted.

rev.depot_file -> aString

Returns the name of the depot file to which this object refers.

rev.desc -> aString

Returns the description of the change which created this revision. Note that only the first 31 characters are returned unless you use **`p4 filelog -L`** for the first 250 characters, or **`p4 filelog -l`** for the full text.

rev.digest -> aString

Returns the MD5 digest for this revision of the file.

rev.each_integration { |integ| block } -> integArray

Iterates over each the integration records for this revision of the depot file.

rev.filesize -> aNumber

Returns size of this revision.

rev.integrations -> integArray

Returns the list of integrations for this revision.

rev.rev -> aNumber

Returns the number of this revision of the file.

rev.time -> aTime

Returns the date/time that this revision was created.

rev.type -> aString

Returns this revision's Helix server filetype.

rev.user -> aString

Returns the name of the user who created this revision.

Class P4::Integration

Description

Utility class providing easy access to the details of an integration record. Created by `P4#run_filelog()`.

Class Methods

None.

Instance Methods

integ.how -> aString

Returns the type of the integration record - how that record was created.

integ.file -> aPath

Returns the path to the file being integrated to/from.

integ.srev -> aNumber

Returns the start revision number used for this integration.

integ.erev -> aNumber

Returns the end revision number used for this integration.

Class P4::Map

Description

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

Class Methods

Map.new ([anArray]) -> aMap

Constructs a new **`P4::Map`** object.

Map.join (map1, map2) -> aMap

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.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 -> true

Empty a map.

map.count -> anInteger

Return the number of entries in a map.

map.empty? -> aBool

Test whether a map object is empty.

map.insert(aString, [aString]) -> aMap

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.translate (aString, [aBool])-> aString

Translate a string through a map, and return the result. 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.

map.includes? (aString) -> aBool

Tests whether a path is mapped or not.

```
if( map.includes?( "//depot/main/..." ) )
  ...
end
```

map.reverse -> aMap

Return a new **`P4::Map`** object with the left and right sides of the mapping swapped. The original object is unchanged.

map.lhs -> anArray

Returns the left side of a mapping as an array.

map.rhs -> anArray

Returns the right side of a mapping as an array.

map.to_a -> anArray

Returns the map as an array.

Class P4::MergeData

Description

Class containing the context for an individual merge during execution of a **`p4 resolve`**.

Class Methods

None.

Instance Methods

md.your_name() -> aString

Returns the name of "your" file in the merge. This is typically a path to a file in the workspace.

```
p4.run_resolve() do
  |md|
  yours = md.your_name
  md.merge_hint # merge result
end
```

md.their_name() -> aString

Returns the name of "their" file in the merge. This is typically a path to a file in the depot.

```
p4.run_resolve() do
  |md|
  theirs = md.their_name
  md.merge_hint # merge result
end
```

md.base_name() -> aString

Returns the name of the "base" file in the merge. This is typically a path to a file in the depot.

```
p4.run_resolve() do
  |md|
  base = md.base_name
```

```
  md.merge_hint # merge result
end
```

md.your_path() -> aString

Returns the path of "your" file in the merge. This is typically a path to a file in the workspace.

```
p4.run_resolve() do
  |md|
  your_path = md.your_path
  md.merge_hint # merge result
end
```

md.their_path() -> aString

Returns the path of "their" file in the merge. This is typically a path to a temporary file on your local machine in which the contents of **`P4::MergeData#their_name()`** have been loaded.

```
p4.run_resolve() do
  |md|
  their_name = md.their_name
  their_file = File.open( md.their_path )
  md.merge_hint # merge result
end
```

md.base_path() -> aString

Returns the path of the base file in the merge. This is typically a path to a temporary file on your local machine in which the contents of **`P4::MergeData#base_name()`** have been loaded.

```
p4.run_resolve() do
  |md|
  base_name = md.base_name
  base_file = File.open( md.base_path )
  md.merge_hint # merge result
end
```

md.result_path() -> aString

Returns the path to the merge result. This is typically a path to a temporary file on your local machine in which the contents of the automatic merge performed by the server have been loaded.

```
p4.run_resolve() do
  |md|
  result_file = File.open( md.result_path )
  md.merge_hint # merge resultend
```

md.merge_hint() -> aString

Returns the hint from the server as to how it thinks you might best resolve this merge.

```
p4.run_resolve() do
  |md|
  puts ( md.merge_hint ) # merge result
end
```

md.run_merge() -> aBool

If the environment variable **P4MERGE** is defined, **P4::MergeData#run_merge()** invokes the specified program and returns a boolean based on the return value of that program.

```
p4.run_resolve() do
  |md|
  if ( md.run_merge() )
    "am"
  else
    "s"
  end
end
```

Class P4::Message

Description

P4::Message objects contain error or other diagnostic messages from the Helix Core server; retrieve them by using the **P4#messages()** method.

Script writers can test the severity of the messages in order to determine if the server message consisted of command output (**E_INFO**), warnings, (**E_WARN**), or errors (**E_FAILED**/**E_FATAL**).

Class methods

None.

Instance methods

message.severity() -> anInteger

Severity of the message, which is one of the following values:

Value	Meaning
`E_EMPTY`	No error
`E_INFO`	Informational message only
`E_WARN`	Warning message only
`E_FAILED`	Command failed
`E_FATAL`	Severe error; cannot continue.

message.generic() -> anInteger

Returns the generic class of the error.

message.msgid() -> anInteger

Returns the unique ID of the message.

message.to_s() -> aString

Converts the message into a string.

message.inspect() -> aString

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

Class P4::OutputHandler

Description

The `P4::OutputHandler` class is a handler class that provides access to streaming output from the server. After defining the output handler, set `P4#handler()` to an instance of a subclass of `P4::OutputHandler` (or use a `p4.with_handler( handler )` block) to enable callbacks.

By default, `P4::OutputHandler` returns `P4::REPORT` for all output methods. The different return options are:

Value	Meaning
`P4::REPORT`	Messages added to output.
`P4::HANDLED`	Output is handled by class (don't add message to output).
`P4::CANCEL`	Operation is marked for cancel, message is added to output.

Class Methods

new P4::MyHandler.new -> aP4::OutputHandler

Constructs a new subclass of `P4::OutputHandler`.

Instance Methods

outputBinary -> int

Process binary data.

outputInfo -> int

Process tabular data.

outputMessage -> int

Process informational or error messages.

outputStat -> int

Process tagged data.

outputText -> int

Process text data.

Class P4::Progress

Description

The `P4::Progress` class is a handler class that provides access to progress indicators from the server. After defining the output handler, set `P4#progress()` to an instance of a subclass of `P4::Progress` (or use a `p4.with_progress( progress )` block) to enable callbacks.

You must implement all five of the following methods: `init()`, `description()`, `update()`, `total()`, and `done()`, even if the implementation consists of trivially returning `0`.

Class Methods

new P4::MyProgress.new -> aP4::Progress

Constructs a new subclass of `P4::Progress`.

Instance Methods

init -> int

Initialize progress indicator.

description -> int

Description and type of units to be used for progress reporting.

update -> int

If non-zero, user has requested a cancellation of the operation.

total -> int

Total number of units expected (if known).

done -> int

If non-zero, operation has failed.

Class P4::Spec

Description

The `P4::Spec` class is a hash containing key/value pairs for all the fields in a Helix server form. It provides two things over and above its parent class (Hash):

- Fieldname validation. Only valid field names may be set in a `P4::Spec` object. Note that only the field name is validated, not the content.
- Accessor methods for easy access to the fields.

Class Methods

new P4::Spec.new(anArray) -> aP4::Spec

Constructs a new **`P4::Spec`** object given an array of valid fieldnames.

Instance Methods

spec._<fieldname> -> aValue

Returns the value associated with the field named <*fieldname*>. This is equivalent to **`spec[ "<fieldname>" ]`** with the exception that when used as a method, the fieldnames may be in lowercase regardless of the actual case of the fieldname.

```
client = p4.fetch_client()
root   = client._root
desc   = client._description
```

spec._<fieldname>= aValue -> aValue

Updates the value of the named field in the spec. Raises a **`P4Exception`** if the fieldname is not valid for specs of this type.

```
client               = p4.fetch_client()
client._root         = "/home/bruno/new-client"
client._description  = "My new client spec"
p4.save_client( client )
```

spec.permitted_fields -> anArray

Returns an array containing the names of fields that are valid in this spec object. This does not imply that values for all of these fields are actually set in this object, merely that you may choose to set values for any of these fields if you want to.

```
client = p4.fetch_client()
spec.permitted_fields.each do
  | field |
  printf ( "%14s = %s\n", field, client[ field ] )
end
```

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

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.

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

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

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

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