HelixCore

APIs for Scripting

2017.1

December 2017

PERFORCE

www.perforce.com

Contents

How to use this guide

This guide contains details about using the derived APIs for Perl, PHP, Python, and Ruby, to create scripts that interact with Helix Core. You can download these APIs from the Perforce web site at https://www.perforce.com/downloads:

- Helix Core API for Perl (P4Perl) - https://www.perforce.com/downloads/helix-core-api-perl
- Helix Core API for PHP (P4PHP) - https://www.perforce.com/downloads/helix-core-api-php
- Helix Core API for Python (P4Python) - https://www.perforce.com/downloads/helix-core-api-python
- Helix Core API for Ruby (P4Ruby) - https://www.perforce.com/downloads/helix-core-api-ruby

These derived APIs depend on the Helix C/C++ API, details for which are at *Helix C/C++ API User Guide* at https://www.perforce.com/perforce/doc.current/manuals/p4api/.

Feedback

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

Other Helix Core documentation

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

Syntax conventions

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

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

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

P4Ruby

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

The main features are:

- Get Helix Core data and forms in hashes and arrays.
- Edit Helix Core 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, Solaris, OS X, and FreeBSD.

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:

```
anges = p4.run_changes("-uusername", "-m1").shift
```

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 -- --with-p4api-dir=__DIR__
```

In the example above, the *DIR* is the path to a local copy of the Helix C/C++ API distribution. The Helix C/C++ API should match the major and minor version of P4Ruby. If you omit the `--with-p4api-dir` option, the gem attempts to download a version of the API itself from ftp.perforce.com.

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

More installation options are described in the P4Ruby project in the Perforce Workshop:

https://swarm.workshop.perforce.com/projects/perforce-software-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 Versioning Engine, 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 Versioning Engine.
`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`.
`prog`	Get program name as shown by `p4 monitor show -e`.

Method	Description
`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.
`ticketfile`	Get the location of the **`P4TICKETS`** file.

Method	Description
`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.
`client`	Client workspace used to create this revision.
`depot_file`	Name of the file in the depot.

Method	Description
`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 Versioning Engine.

Method	Description
`new`	Construct a new map object (class method).
`join`	Joins two maps to create a third (class method).
`clear`	Empties a map.
`count`	Returns the number of entries in a map.
`empty?`	Tests whether or not a map object is empty.

Method	Description
`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.
`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 Core client API. Each **P4** object provides you with a thread-safe API level interface to Helix Core. The basic model is to:

1. Instantiate your `P4` object.
2. Specify your Helix Core 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 Core 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 Core 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 Core 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 Core 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 the API integer levels that correspond to each Helix Core release, see:

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

p4.api_level -> anInteger

Returns the current Helix C/C++ API compatibility level. Each iteration of the Helix Versioning Engine is given a level number. As part of the initial communication, the client protocol level is passed between client application and the Helix Versioning Engine. This value, defined in the Helix C/C++ API, determines the communication protocol level that the Helix Core client will understand. All subsequent responses from the Helix Versioning Engine 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 Core 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 Core client currently in use.

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

p4.connect -> aBool

Connect to the Helix Versioning Engine. 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 Core 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 Versioning Engine.

```
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 Core 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 Core 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 Core 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 Core 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 Core 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 Core 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 Core 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 Core 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 Core system administrators running **`p4 monitor show -e`** in Helix Core 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.reset() -> anArray

Reset messages, warnings, and errors from a previous run() call. The returned array is always empty.

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 Core 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 Core. 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, see:

http://kb.perforce.com/article/571

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 Core 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 Core 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 Core 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 Core 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 Core-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 Core 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 Core 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 Core 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 Core 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 Versioning Engine; 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
```

P4Perl

P4Perl is a Perl module that provides an object-oriented API to Helix Core. Using P4Perl is faster than using the command-line interface in scripts, because multiple command can be executed on a single connection, and because it returns Helix Core responses as Perl hashes and arrays.

The main features are:

- Get Helix Core data and forms in hashes and arrays.
- Edit Helix Core forms by modifying hashes.
- Run as many commands on a connection as required.
- The output of commands is returned as a Perl array.
- The elements of the array returned are strings or, where appropriate, hash references.

System Requirements and Release Notes

P4Perl is supported on Windows, Linux, Solaris, OS X, and FreeBSD.

For system requirements, see the release notes at https://www.perforce.com/perforce/doc.current/user/p4perlnotes.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:

```
anges = p4.run_changes("-uusername", "-m1").shift
```

If you include a space (`"-u username"`), the command fails.

Installing P4Perl

You can download P4Perl from the Perforce web site at https://www.perforce.com/downloads/helix-core-api-perl.

After downloading, you can either run the installer or build the interface from source, as described in the release notes.

Programming with P4Perl

The following example shows how to connect to a Helix server, run a `p4 info` command, and open a file for edit.

```
#!/opt/local/bin/perl -w
use strict;
use P4;
my $p4 = new P4;
$p4->SetClient('bruno_ws');
$p4->SetUser('smoon');
$p4->SetPort('localhost:20081');
$p4->SetVersion("EnvTest 1.0");
$p4->Connect() or die("Was not able to connect\n");
my $info = $p4->Run("info"); #passing array ref
print "\n\nP4 Info Output:\n\n";
foreach my $akey (@{$info}) {
my @infos = keys %$akey; # $akey is hash ref
foreach my $hkey (@infos) {
print "$hkey => $akey->{$hkey}\n";
}
}
my $client_name = $p4->FetchClient($p4->GetClient());
print "\n\nClient Specification:\n\n";
foreach my $chkey (keys %{$client_name}) {
if ($client_name->{$chkey} =~ /^ARRAY(.+)$/) {
my $avals = $client_name->{$chkey};
foreach my $achkey (@{$avals})
{ print "$chkey => $achkey\n"; }
} elsif($client_name->{$chkey} =~ /^HASH(.+)$/) {
my $hvals = $client_name->{$chkey};
foreach my $hchkey (keys %{$hvals}) {
print "$chkey => $hvals->{$hchkey}\n";
}
} else {
print "$chkey => $client_name->{$chkey}\n";
}
}
my $changes = $p4->Run("changes","-m2");
print "\n\nTwo Most Recent Changes:\n\n";
foreach my $each_chg (@{$changes}) {
my @chg_key = keys %$each_chg; # $each_chg is hash ref
foreach my $hchg (@chg_key) {
print "$hchg => $each_chg->{$hchg}\n";
}
print "\n";
}
print "\n" . $p4->GetVersion() . "\n";
$p4->Disconnect();
```

Connecting to Helix Core over SSL

Scripts written with P4Perl 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.

P4Perl Classes

The P4 module consists of several public classes:

- "P4" below
- "P4::DepotFile" on page 59
- "P4::Revision" on page 60
- "P4::Integration" on page 60
- "P4::Map" on page 60
- "P4::MergeData" on page 61
- "P4::Message" on page 62
- "P4::OutputHandler" on page 62
- "P4::Progress" on page 62
- "P4::Spec" on page 63

The following tables provide brief details about each public class.

P4

The main class used for executing Helix Core commands. Almost everything you do with P4Perl will involve this class.

Method	Description
`new()`	Construct a new **P4** object.

Method	Description
`Identify()`	Print build information including P4Perl version and Helix C/C++ API version.
`ClearHandler()`	Clear the output handler.
`Connect()`	Initialize the Helix Core client and connect to the Server.
`Disconnect()`	Disconnect from the Helix Versioning Engine.
`ErrorCount()`	Returns the number of errors encountered during execution of the last command.
`Errors()`	Returns a list of the error strings received during execution of the last command.
`Fetch_<Spectype>()`	Shorthand for running: `$p4->Run( "<spectype>", "-o" );`
`Format_<Spectype>_()`	Shorthand for running: `$p4->FormatSpec( "<spectype>", hash );`
`FormatSpec()`	Converts a Helix server form of the specified type (client/label etc.) held in the supplied hash into its string representation.
`GetApiLevel()`	Get current API compatibility level.
`GetCharset()`	Get character set when connecting to Unicode servers.
`GetClient()`	Get current client workspace (`P4CLIENT`).
`GetCwd()`	Get current working directory.
`GetEnv()`	Get the value of a Helix Core environment variable, taking into account `P4CONFIG` files and (on Windows or OS X) the registry or user preferences.
`GetHandler()`	Get the output handler.
`GetHost()`	Get the current hostname.
`GetMaxLockTime()`	Get `MaxLockTime` used for all following commands.

Method	Description
`GetMaxResults()`	Get `MaxResults` used for all following commands.
`GetMaxScanRows()`	Get `MaxScanRows` used for all following commands.
`GetPassword()`	Get the current password or ticket.
`GetPort()`	Get host and port (`P4PORT`).
`GetProg()`	Get the program name as shown by the `p4 monitor show -e` command.
`GetProgress()`	Get the progress indicator.
`GetTicketFile()`	Get the location of the `P4TICKETS` file.
`GetUser()`	Get the current username (`P4USER`).
`GetVersion()`	Get the version of your script, as reported to the Helix Versioning Engine.
`IsConnected()`	Test whether or not session has been connected and/or has been dropped.
`IsStreams()`	Test whether or not streams are enabled.
`IsTagged()`	Test whether or not tagged output is enabled.
`IsTrack()`	Test whether or not server performance tracking is enabled.
`Iterate_<`*`Spectype`*`>()`	Iterate through spec results.
`Messages()`	Return an array of `P4::Message` objects, one for each message sent by the server.
`P4ConfigFile()`	Get the location of the configuration file used (`P4CONFIG`).
`Parse_<`*`Spectype`*`>()`	Shorthand for running: `$p4-ParseSpec( "<spectype>", buffer );`
`ParseSpec()`	Converts a Helix server form of the specified type (`client`, `label`, etc.) held in the supplied string into a hash and returns a reference to that hash.

Method	Description
`Run`*`Cmd`*`()`	Shorthand for running: `$p4-Run( "`*`cmd`*`", arg, ... );`
`Run()`	Run a Helix Core command and return its results. Check for errors with `P4::ErrorCount()`.
`RunFilelog()`	Runs a `p4 filelog` on the `fileSpec` provided and returns an array of `P4::DepotFile` objects.
`RunLogin()`	Runs `p4 login` using a password or ticket set by the user.
`RunPassword()`	A thin wrapper for changing your password.
`RunResolve()`	Interface to `p4 resolve`.
`RunSubmit()`	Submit a changelist to the server.
`RunTickets()`	Get a list of tickets from the local tickets file.

`Save_`*`<Spectype>`*`()`	Shorthand for running:
	`$p4->SetInput( $spectype );`
	`$p4->Run( "`*`<spectype>`*`", "-i" );`

`ServerCaseSensitive()`	Returns an integer specifying whether or not the server is case-sensitive.
`ServerLevel()`	Returns an integer specifying the server protocol level.
`ServerUnicode()`	Returns an integer specifying whether or not the server is in Unicode mode.
`SetApiLevel()`	Specify the API compatibility level to use for this script.
`SetCharset()`	Set character set when connecting to Unicode servers.
`SetClient()`	Set current client workspace (`P4CLIENT`).
`SetCwd()`	Set current working directory.
`SetEnv()`	On Windows or OS X, set an environment variable in the registry or user preferences.
`SetHandler()`	Set the output handler.
`SetHost()`	Set the name of the current host (`P4HOST`).

Method	Description
`SetInput()`	Save the supplied argument as input to be supplied to a subsequent command.
`SetMaxLockTime()`	Set `MaxLockTime` used for all following commands.
`SetMaxResults()`	Set `MaxResults` used for all following commands.
`SetMaxScanRows()`	Set `MaxScanRows` used for all following commands.
`SetPassword()`	Set Helix Core password (`P4PASSWD`).
`SetPort()`	Set host and port (`P4PORT`).
`SetProg()`	Set the program name as shown by the `p4 monitor show -e` command.
`SetProgress()`	Set the progress indicator.
`SetStreams()`	Enable or disable streams support.
`SetTicketFile()`	Set the location of the `P4TICKETS` file.
`SetTrack()`	Activate or deactivate server performance tracking. By default, tracking is off (0).
`SetUser()`	Set the Helix Core username (`P4USER`).
`SetVersion()`	Set the version of your script, as reported to the Helix Versioning Engine.
`Tagged()`	Toggles tagged output (1 or 0). By default, tagged output is on (1).
`TrackOutput()`	If performance tracking is enabled with `SetTrack()` returns an array of strings with tracking output.
`WarningCount()`	Returns the number of warnings issued by the last command.
`Warnings()`	Returns a list of the warning strings received during execution of the last command.

P4::DepotFile

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

Method	Description
`DepotFile()`	Name of the depot file to which this object refers.
`Revisions()`	Returns an array of revision objects for the depot file.

P4::Revision

Utility class allowing access to the attributes of a revision of a file in the depot. Returned by `P4::RunFilelog()`.

Method	Description
`Action()`	Returns the action that created the revision.
`Change()`	Returns the changelist number that gave rise to this revision of the file.
`Client()`	Returns the name of the client from which this revision was submitted.
`DepotFile()`	Returns the name of the depot file to which this object refers.
`Desc()`	Returns the description of the change which created this revision.
`Digest()`	Returns the MD5 digest for this revision.
`FileSize()`	Returns the size of this revision.
`Integrations()`	Returns an array of `P4::Integration` objects representing all integration records for this revision.
`Rev()`	Returns the number of this revision.
`Time()`	Returns date/time this revision was created.
`Type()`	Returns the Helix Core filetype of this revision.
`User()`	Returns the name of the user who created this revision.

P4::Integration

Utility class allowing access to the attributes of an integration record for a revision of a file in the depot. Returned by `P4::RunFilelog()`.

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 Helix Core mappings without requiring a connection to the Helix Versioning Engine.

Method	Description
`New()`	Construct a new Map object (class method).
`Join()`	Joins two maps to create a third (class method).
`Clear()`	Empties a map.
`Count()`	Returns the number of entries in a map.
`IsEmpty()`	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.
`AsArray()`	Returns the map as an array.

P4::MergeData

Class encapsulating the context of an individual merge during execution of a `p4 resolve` command. Passed to `P4::RunResolve`.

Method	Description
`YourName()`	Returns the name of "your" file in the merge. (file in workspace)
`TheirName()`	Returns the name of "their" file in the merge. (file in the depot)
`BaseName()`	Returns the name of "base" file in the merge. (file in the depot)
`YourPath()`	Returns the path of "your" file in the merge. (file in workspace)
`TheirPath()`	Returns the path of "their" file in the merge. (temporary file on workstation into which `TheirName()` has been loaded)
`BasePath()`	Returns the path of the base file in the merge. (temporary file on workstation into which `BaseName()` has been loaded)
`ResultPath()`	Returns the path to the merge result. (temporary file on workstation into which the automatic merge performed by the server has been loaded)
`MergeHint()`	Returns hint from server as to how user might best resolve merge.
`RunMergeTool()`	If the environment variable `P4MERGE` is defined, run it and indicate whether or not the merge tool successfully executed.

P4::Message

Class encapsulating the context of an individual error during execution of Helix Core commands. Passed to `P4::Messages()`.

Method	Description
`GetSeverity()`	Returns the severity class of the error.
`GetGeneric()`	Returns the generic class of the error message.
`GetId()`	Returns the unique ID of the error message.
`GetText()`	Get the text of the error message.

P4::OutputHandler

Handler class that provides access to streaming output from the server; call `P4::SetHandler()` with an implementation 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; call `P4::SetProgress()` with an implementation of `P4::Progress` 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::Resolver

Class for handling resolves in Helix Core.

Method	Description
`Resolve()`	Perform a resolve and return the resolve decision as a string.

P4::Spec

Utility class allowing access to the attributes of the fields in a Helix server form.

Method	Description
`_fieldname()`	Return the value associated with the field named *fieldname*.
`_fieldname()`	Set the value associated with the field named *fieldname*.
`PermittedFields()`	Lists the fields that are permitted for specs of this type.

Class P4

Description

Main interface to the Helix Core client API.

This module provides an object-oriented interface to Helix Core, the Perforce version control system. Data is returned in Perl arrays and hashes and input can also be supplied in these formats.

Each **P4** object represents a connection to the Helix Versioning Engine, also called Helix server, and multiple commands may be executed (serially) over a single connection.

The basic model is to:

1. Instantiate your **P4** object.
2. Specify your Helix Core client environment.
    - `SetClient()`
    - `SetHost()`
    - `SetPassword()`
    - `SetPort()`
    - `SetUser()`

3. Connect to the Perforce service.

   The Helix Core protocol is not designed to support multiple concurrent queries over the same connection. Multithreaded applications that use the C++ API or derived APIs (including P4Perl) should ensure that a separate connection is used for each thread, or that only one thread may use a shared connection at a time.
4. Run your Helix Core commands.
5. Disconnect from the Perforce service.

Class methods

P4::new() -> P4

Construct a new `P4` object. For example:

```
my $p4 = new P4;
```

P4::Identify() -> string

Print build information including P4Perl version and Helix C/C++ API version.

```
print P4::Identify();
```

The constants `OS`, `PATCHLEVEL` and `VERSION` are also available to test an installation of P4Perl without having to parse the output of `P4::Identify()`. Also reports the version of the OpenSSL library used for building the underlying Helix C/C++ API with which P4Perl was built.

P4::ClearHandler() -> undef

Clear any configured output handler.

P4::Connect() -> bool

Initializes the Helix Core client and connects to the server. Returns `false` on failure and `true` on success.

P4::Disconnect() -> undef

Terminate the connection and clean up. Should be called before exiting.

P4::ErrorCount() -> integer

Returns the number of errors encountered during execution of the last command.

P4::Errors() -> list

Returns a list of the error strings received during execution of the last command.

P4::Fetch<Spectype>([name]) -> hashref

Shorthand for running:

```
$p4->Run( "<spectype>", "-o" );
```

and returning the first element of the result array. For example:

```
$label      = $p4->FetchLabel( $labelname );
$change     = $p4->FetchChange( $changeno );
$clientspec = $p4->FetchClient( $clientname );
```

P4::Format<Spectype>(hash) -> string

Shorthand for running:

```
$p4->FormatSpec( "<spectype>", hash);
```

and returning the results. For example:

```
$change     = $p4->FetchChange();
$change->{ 'Description' } = 'Some description';
$form       = $p4->FormatChange( $change );
printf( "Submitting this change:\n\n%s\n", $form );
$p4->RunSubmit( $change );
```

P4::FormatSpec($spectype, $string) -> string

Converts a Helix server form of the specified type (`client`, `label`, etc.) held in the supplied hash into its string representation. Shortcut methods are available that obviate the need to supply the type argument. The following two examples are equivalent:

```
my $client = $p4->FormatSpec( "client", $hash );
```

```
my $client = $p4->FormatClient( $hash );
```

P4::GetApiLevel() -> integer

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

For more information, see:

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

P4::GetCharset() -> string

Return the name of the current charset in use. Applicable only when used withHelix servers running in unicode mode.

P4::GetClient() -> string

Returns the current Helix Core client name. This may have previously been set by `P4::SetClient()`, or may be taken from the environment or `P4CONFIG` file if any. If all that fails, it will be your hostname.

P4::GetCwd() -> string

Returns the current working directory as your Helix Core client sees it.

P4::GetEnv($var) -> string

Returns the value of a Helix Core environment variable, taking into account the settings of Helix Core variables in `P4CONFIG` files, and, on Windows or OS X, in the registry or user preferences.

P4::GetHandler() -> Handler

Returns the output handler.

P4::GetHost() -> string

Returns the client hostname. Defaults to your hostname, but can be overridden with `P4::SetHost()`

P4::GetMaxLockTime($value) -> integer

Get the current `maxlocktime` setting.

P4::GetMaxResults($value) -> integer

Get the current `maxresults` setting.

P4::GetMaxScanRows($value) -> integer

Get the current `maxscanrows` setting.

P4::GetPassword() -> string

Returns your Helix Core password. Taken from a previous call to `P4::SetPassword()` or extracted from the environment (`$ENV{P4PASSWD}`), or a `P4CONFIG` file.

P4::GetPort() -> string

Returns the current address for your Helix Core server. Taken from a previous call to `P4::SetPort()`, or from `$ENV{P4PORT}` or a `P4CONFIG` file.

P4::GetProg() -> string

Get the name of the program as reported to the Helix Versioning Engine.

P4::GetProgress() -> Progress

Returns the progress indicator.

P4::GetTicketFile([$string]) -> string

Return the path of the current `P4TICKETS` file.

P4::GetUser() -> String

Get the current user name. Taken from a previous call to `P4::SetUser()`, or from `$ENV{P4USER}` or a `P4CONFIG` file.

P4::GetVersion($string) -> string

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

P4::IsConnected() -> bool

Returns true if the session has been connected, and has not been dropped.

P4::IsStreams() -> bool

Returns true if streams support is enabled on this server.

P4::IsTagged() -> bool

Returns true if Tagged mode is enabled on this client.

P4::IsTrack() -> bool

Returns true if server performance tracking is enabled for this connection.

P4::Iterate<Spectype>(arguments) -> object

Iterate over spec results. Returns an iterable object with `next()` and `hasNext()` methods.

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 `P4::Run`*`Cmd`*`()` command.

Arguments can be passed to the iterator to filter the results, for example, to iterate over only the first two client workspace specifications:

```
$p4->IterateClients( "-m2" );
```

You can also pass the spec type as an argument:

```
$p4->Iterate( "changes" );
```

For example, to iterate through client specs:

```
use P4;

my $p4 = P4->new;
$p4->Connect or die "Couldn't connect";
my $i = $p4->IterateClients();
while($i->hasNext) {
  my $spec = $i->next;
  print( "Client: " . ($spec->{Client} or "<undef>") . "\n" );
}
```

P4::Messages() -> list

Returns an array of `P4::Message()` objects, one for each message (info, warning or error) sent by the server.

P4::P4ConfigFile() -> string

Get the path to the current `P4CONFIG` file.

P4::Parse<Spectype>($string) -> hashref

Shorthand for running:

```
$p4-ParseSpec( "<spectype>", buffer);
```

and returning the results. For example:

```
$p4 = new P4;
$p4->Connect() or die( "Failed to connect to server" );
$client = $p4->FetchClient();

# Returns a hashref
```

```
$client = $p4->FormatClient( $client );

# Convert to string
$client = $p4->ParseClient( $client );
# Convert back to hashref
```

Comments in forms are preserved. Comments are stored as a `comment` key in the spec hash and are accessible. For example:

```
my $spec = $pc->ParseGroup( 'my_group' );
print $spec->{'comment'};
```

P4::ParseSpec($spectype, $string) -> hashref

Converts a Helix server form of the specified type (client/label etc.) held in the supplied string into a hash and returns a reference to that hash. Shortcut methods are available to avoid the need to supply the type argument. The following two examples are equivalent:

```
my $hash = $p4->ParseSpec( "client", $clientspec );
```

```
my $hash = $p4->ParseClient( $clientspec );
```

Important
Custom specifications require that you call `Fetch` first so that the `specDefs` can be determined by the API and later used by `ParseSpec`.

P4::Run<Cmd>([$arg...]) -> list | arrayref

Shorthand for running:

```
$p4-Run( "cmd", arg, ...);
```

and returning the results.

P4::Run("<cmd>", [$arg...]) -> list | arrayref

Run a Helix Core command and return its results. Because Helix Core commands can partially succeed and partially fail, it is good practice to check for errors using `P4::ErrorCount()`.

Results are returned as follows:

- A list of results in array context
- An array reference in scalar context

The AutoLoader enables you to treat Helix Core commands as methods:

```
p4->RunEdit( "filename.txt" );
```

is equivalent to:

```
$p4->Run( "edit", "filename.txt" );
```

Note that the content of the array of results you get depends on (a) whether you're using tagged mode, (b) the command you've executed, (c) the arguments you supplied, and (d) your Helix server version.

Tagged mode and form parsing mode are turned on by default; each result element is a hashref, but this is dependent on the command you ran and your server version.

In non-tagged mode, each result element is a string. In this case, because the Helix server sometimes asks the client to write a blank line between result elements, some of these result elements can be empty.

Note that the return values of individual Helix Core commands are not documented because they may vary between server releases.

To correlate the results returned by the P4 interface with those sent to the command line client, try running your command with RPC tracing enabled. For example:

Tagged mode: **`p4 -Ztag -vrpc=1 describe -s 4321`**

Non-Tagged mode: **`p4 -vrpc=1 describe -s 4321`**

Pay attention to the calls to `client-FstatInfo()`, `client-OutputText()`, `client-OutputData()` and `client-HandleError()`. Each call to one of these functions results in either a result element, or an error element.

P4::RunFilelog([$args ...], $fileSpec ...) -> list | arrayref

Runs a **`p4 filelog`** on the `fileSpec` provided and returns an array of **`P4::DepotFile`** objects when executed in tagged mode.

P4::RunLogin(...) -> list | arrayref

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

P4::RunPassword($oldpass, $newpass) -> list | arrayref

A thin wrapper for changing your password from `$oldpass` to `$newpass`. Not to be confused with `P4::SetPassword()`.

P4::RunResolve([$resolver], [$args ...]) -> string

Run a **`p4 resolve`** command. Interactive resolves require the `$resolver` parameter to be an object of a class derived from **`P4::Resolver`**. In these cases, the `P4::Resolve()` method of this class is called to handle the resolve. For example:

```
$resolver = new MyResolver;
$p4->RunResolve( $resolver );
```

To perform an automated merge that skips whenever conflicts are detected:

```
use P4;

package MyResolver;
our @ISA = qw( P4::Resolver );

sub Resolve( $ ) {
  my $self      = shift;
  my $mergeData = shift;

  # "s"kip if server-recommended hint is to "e"dit the file,
  # because such a recommendation implies the existence of a conflict
  return "s" if ( $mergeData->Hint() eq "e" );
  return $mergeData->Hint();
}
1;

package main;

$p4 = new P4;
$resolver = new MyResolver;

$p4->Connect() or die( "Failed to connect to Perforce" );
$p4->RunResolve( $resolver, ... );
```

In non-interactive resolves, no `P4::Resolver` object is required. For example:

```
$p4->RunResolve( "at" );
```

P4::RunSubmit($arg | $hashref, ...) -> list | arrayref

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

```
$change = $p4->FetchChange();
$change->{ 'Description' } = "Some description";
$p4->RunSubmit( "-r", $change );
```

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

```
$p4->RunSubmit( "-d", "Some description", "somedir/..." );
```

P4::RunTickets() -> list

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>() -> list | arrayref

Shorthand for running:

```
$p4->SetInput( $spectype );
$p4->Run( "<spectype>", "-i");
```

For example:

```
$p4->SaveLabel( $label );
$p4->SaveChange( $changeno );
$p4->SaveClient( $clientspec );
```

P4::ServerCaseSensitive() -> integer

Returns an integer specifying whether or not the server is case-sensitive.

P4::ServerLevel() -> integer

Returns an integer specifying the server protocol level. This is not the same as, but is closely aligned to, the server version. To find out your server's protocol level, run `p4 -vrpc=5 info` and look for the `server2` protocol variable in the output. For more information, see:

http://kb.perforce.com/article/571

P4::ServerUnicode() -> integer

Returns an integer specifying whether or not the server is in Unicode mode.

P4::SetApiLevel($integer) -> undef

Specify the API compatibility level to use for this script. This is useful when you want your script to continue to work on newer server versions, even if the new server adds tagged output to previously unsupported commands.

The additional tagged output support can change the server's output, and confound your scripts. Setting the API level to a specific value allows you to lock the output to an older format, thus increasing the compatibility of your script.

Must be called before calling `P4::Connect()`. For example:

```
$p4->SetApiLevel( 67 ); # Lock to 2010.1 format
$p4->Connect() or die( "Failed to connect to Perforce" );
# etc.
```

P4::SetCharset($charset) -> undef

Specify the character set to use for local files when used with aHelix server running in unicode mode. Do not use unless yourHelix serveris in unicode mode. Must be called before calling `P4::Connect()`. For example:

```
$p4->SetCharset( "winansi" );
$p4->SetCharset( "iso8859-1" );
$p4->SetCharset( "utf8" );
# etc.
```

P4::SetClient($client) -> undef

Sets the name of your Helix Core client workspace. If you don't call this method, then the client workspace name will default according to the normal Helix Core conventions:

1. Value from file specified by `P4CONFIG`
2. Value from `$ENV{P4CLIENT}`
3. Hostname

P4::SetCwd($path) -> undef

Sets the current working directory for the client.

P4::SetEnv($var, $value) -> undef

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->SetEnv( "P4CLIENT", "my_workspace" );
$P4->SetEnv( "P4CLIENT", "");
```

P4::SetHandler(Handler) -> Handler

Sets the output handler.

P4::SetHost($hostname) -> undef

Sets the name of the client host, overriding the actual hostname. This is equivalent to **`p4 -H`** ***`hostname`***, and only useful when you want to run commands as if you were on another machine.

P4::SetInput($string | $hashref | $arrayref) -> undef

Save the supplied argument as input to be supplied to a subsequent command. The input may be a hashref, a scalar string, or an array of hashrefs or scalar strings. If you pass an array, the array will be shifted once each time the Helix Core command being executed asks for user input.

P4::SetMaxLockTime($integer) -> undef

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 maxresults` for information on the commands that support this limit.

P4::SetMaxResults($integer) -> undef

Limit the number of results for subsequent commands to the value specified. Helix Core will abort the command if continuing would produce more than this number of results. Once set, this limit remains in force unless you remove the restriction by setting it to a value of 0.

P4::SetMaxScanRows($integer) -> undef

Limit the number of records Helix Core will scan when processing subsequent commands to the value specified. Helix Core will abort the command once this number of records has been scanned. Once set, this limit remains in force unless you remove the restriction by setting it to a value of 0.

P4::SetPassword($password) -> undef

Specify the password to use when authenticating this user against the Helix Versioning Engine - overrides all defaults. Not to be confused with `P4::Password()`.

P4::SetPort($port) -> undef

Set the port on which your Helix server is listening. Defaults to:

1. Value from file specified by `P4CONFIG`
2. Value from `$ENV{P4PORT}`
3. `perforce:1666`

P4::SetProg($program_name) -> undef

Set the name of your script. This value is displayed in the server log on 2004.2 or later servers.

P4::SetProgress(Progress) -> Progress

Sets the progress indicator.

P4::SetStreams(0 | 1) -> undef

Enable or disable support for streams. By default, streams support is enabled at 2011.1 or higher (`P4::GetApiLevel()` >= 70). Streams support requires a server at 2011.1 or higher. You can enable or disable support for streams both before and after connecting to the server.

P4::SetTicketFile([$string]) -> string

Set the path to the current `P4TICKETS` file (and return it).

P4::SetTrack(0 | 1) -> undef

Enable (1) or disable (0) server performance tracking for this connection. By default, performance tracking is disabled.

P4::SetUser($username) -> undef

Set your Helix Core username. Defaults to:

1. Value from file specified by `P4CONFIG`
2. Value from `C<$ENV{P4USER}>`
3. OS username

P4::SetVersion($version) -> undef

Specify the version of your script, as recorded in the Helix server log file.

P4::Tagged(0 | 1 | $coderef) -> undef

Enable (1) or disable (0) tagged output from the server, or temporarily toggle it.

By default, tagged output is enabled, but can be disabled (or re-enabled) by calling this method. If you provide a code reference, you can run a subroutine with the tagged status toggled for the duration of that reference. For example:

```
my $GetChangeCounter = sub{ $p4->RunCounter('change')->[ 0 ] );
my $changeno = $p4->Tagged( 0, $GetChangeCounter );
```

When running in tagged mode, responses from commands that support tagged output will be returned in the form of a hashref. When running in non-tagged mode, responses from commands are returned in the form of strings (that is, in plain text).

P4::TrackOutput() -> list

If performance tracking is enabled with `P4::SetTrack()`, returns a list of strings corresponding to the performance tracking output of the most recently-executed command.

P4::WarningCount() -> integer

Returns the number of warnings issued by the last command.

```
$p4->WarningCount();
```

P4::Warnings() -> list

Returns a list of warning strings from the last command

```
$p4->Warnings();
```

Class P4::DepotFile

Description

`P4::DepotFile` objects are used to present information about files in the Helix Core repository. They are returned by `P4::RunFilelog()`.

Class Methods

None.

Instance Methods

$df->DepotFile() -> string

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

$df->Revisions() -> array

Returns an array of `P4::Revision` objects, one for each revision of the depot file.

Class P4::Revision

Description

`P4::Revision` objects are represent individual revisions of files in the Helix Core repository. They are returned as part of the output of `P4::RunFilelog()`.

Class Methods

$rev->Integrations() -> array

Returns an array of `P4::Integration` objects representing all integration records for this revision.

Instance Methods

$rev->Action() -> string

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

$rev->Change() -> integer

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

$rev->Client() -> string

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

$rev->DepotFile() -> string

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

$rev->Desc() -> string

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() -> string

Returns the MD5 digest for this revision.

$rev->FileSize() -> string

Returns the size of this revision.

$rev->Rev() -> integer

Returns the number of this revision of the file.

$rev->Time() -> string

Returns the date/time that this revision was created.

$rev->Type() -> string

Returns this revision’s Helix Core filetype.

$rev->User() -> string

Returns the name of the user who created this revision.

Class P4::Integration

Description

`P4::Integration` objects represent Helix Core integration records. They are returned as part of the output of `P4::RunFilelog()`.

Class Methods

None.

Instance Methods

$integ->How() -> string

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

$integ->File() -> string

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

$integ->SRev() -> integer

Returns the start revision number used for this integration.

$integ->ERev() -> integer

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 Core mappings, without requiring a connection to a Helix server.

Class Methods

$map = new P4::Map([array]) -> 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 = new P4::Map;
$client_map->Insert( "//depot/main/...", "//client/..." );

# Map client syntax to local syntax
$client_root = new P4::Map;
$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() -> undef

Empty a map.

$map->Count() -> integer

Return the number of entries in a map.

$map->IsEmpty() -> bool

Test whether a map object is empty.

$map->Insert(string ...) -> undef

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(string, [bool]) -> string

Translate a string through a map, and return the result. If the optional second argument is 1, translate forward, and if it is 0, translate in the reverse direction. By default, translation is in the forward direction.

$map->Includes(string) -> bool

Tests whether a path is mapped or not.

```
if ( $map->Includes( "//depot/main/..." ) ) {
  ...
}
```

$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() -> array

Returns the left side of a mapping as an array.

$map->Rhs() -> array

Returns the right side of a mapping as an array.

$map->AsArray() -> array

Returns the map as an array.

Class P4::MergeData

Description

Class containing the context for an individual merge during execution of a `p4 resolve`. Users may not create objects of this class; they are created internally during `P4::RunResolve()`, and passed down to the `Resolve()` method of a `P4::Resolver` subclass.

Class Methods

None.

Instance Methods

$md.YourName() -> string

Returns the name of "your" file in the merge, in client syntax.

$md.TheirName() -> string

Returns the name of "their" file in the merge, in client syntax, including the revision number.

$md.BaseName() -> string

Returns the name of the "base" file in the merge, in depot syntax, including the revision number.

$md.YourPath() -> string

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

$md.TheirPath() -> string

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::TheirName()` have been loaded.

$md.BasePath() -> string

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::BaseName()` have been loaded.

$md.ResultPath() -> string

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.

$md.MergeHint() -> string

Returns a string containing the hint from Helix Core's merge algorithm, indicating the recommended action for performing the resolve.

$md.RunMergeTool() -> integer

If the environment variable `P4MERGE` is defined, `P4::MergeData::RunMergeTool()` invokes the specified program and returns true if the merge tool was successfully executed, otherwise returns false.

$md.MergeType() -> string

Returns a string describing the merge type, such as `Branch resolve`.

$md.YourAction() -> string

Returns the name of "your" action, such as `ignore`.

$md.TheirAction() -> string

Returns the name of "their" action, such as `branch`.

$md.MergeAction() -> string

Returns the name of the action used in the merge. For example, if `TheirAction` is `branch` and `YourAction` is `ignore`, then if you choose yours, you get an ignore, and if you choose theirs, you get a branch.

$md.MergeInfo() -> string

Returns an object containing details about the resolve. For example:

```
'clientFile' => '/Users/jdoe/Workspaces/main.p4-
perl/test/resolve/action/file-88.txt',
'fromFile' => '//depot/projA/src/file-88.txt',
'startFromRev' => 'none',
'resolveType' => 'branch',
'resolveFlag' => 'b',
'endFromRev' => '2'
```

Class P4::Message

Description

`P4::Message` objects contain error or other diagnostic messages from the Helix Versioning Engine; they are returned by `P4::Messages()`.

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.GetSeverity() -> int

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.GetGeneric() -> int

Returns the generic class of the error.

$message.GetId() -> int

Returns the unique ID of the message.

$message.GetText() -> int

Converts the message into a string.

Class P4::OutputHandler

Description

The `P4::OutputHandler` class provides access to streaming output from the server. After defining the output handler, call `P4::SetHandler()` with your implementation of `P4::OutputHandler`.

Because P4Perl does not provide a template or superclass, your output handler must implement all five of the following methods: `OutputMessage()`, `OutputText()`, `OutputInfo()`, `OutputBinary()`, and `OutputStat()`, even if the implementation consists of trivially returning `0` (report only: don't handle output, don't cancel operation).

These methods must return one of the following four values:

Value	Meaning
`0`	Messages added to output (don't handle, don't cancel).

Value	Meaning
`1`	Output is handled by class (don't add message to output).
`2`	Operation is marked for cancel, message is added to output.
`3`	Operation is marked for cancel, message not added to output.

Class Methods

None.

Instance Methods

$handler.OutputBinary() -> int

Process binary data.

$handler.OutputInfo() -> int

Process tabular data.

$handler.OutputMessage() -> int

Process informational or error messages.

$handler.OutputStat()-> int

Process tagged data.

$handler.OutputText() -> int

Process text data.

Class P4::Progress

Description

The **`P4::Progress`** provides access to progress indicators from the server. After defining the progress class, call `P4::SetProgress()` with your implementation of **`P4::Progress`**.

Because P4Perl does not provide a template or superclass, 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

None.

Instance Methods

$progress.Init() -> int

Initialize progress indicator.

$progress.Description(string, int) -> int

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

$progress.Update() -> int

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

$progress.Total()-> int

Total number of units expected (if known).

$progress.Done() -> int

If non-zero, operation has failed.

Class P4::Resolver

Description

`P4::Resolver` is a class for handling resolves in Helix Core. It is intended to be subclassed, and for subclasses to override the `Resolve()` method. When `P4::RunResolve()` is called with a `P4::Resolver` object, it calls the `P4::Resolver::Resolve()` method of the object once for each scheduled resolve.

Class Methods

$actionResolve() -> string

Enables support for resolves of branches, deletes, and file types. This method is invoked if an action resolve is required. It lets you add a callback in your Resolver implementation to determine what the resolve action should be after the automatic resolver has evaluated it. This is similar to resolves in P4V, when you are prompted to select what you want to do, given what the automatic resolver suggested. The `$resolver->ActionResolve()` method receives an argument (`mergeData`) and lets you return a string that specifies what to do. See $resolver.Resolve() for available strings.

The following example counts the number of times it has been called (line 5), stores the `mergeData` from the autoresolver (`type`, `hint`, and `info`) and returns what the automatic resolver suggested (`hint`) on line 10 as the answer.

```
sub ActionResolve( $ ) {
    my $self      = shift;
    my $mergeData = shift;

    $self->{'ActionResolve'} += 1;
    $self->{'type'} = $mergeData->Type();
    $self->{'hint'} = $mergeData->MergeHint();
    $self->{'info'} = $mergeData->MergeInfo();

    return $mergeData->MergeHint();
}
```

Instance Methods

$resolver.Resolve() -> string

Returns the resolve decision as a string. The standard Helix Core resolve strings apply:

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

By default, all automatic merges are accepted, and all merges with conflicts are skipped. The `P4::Resolver::Resolve()` method is called with a single parameter, which is a reference to a `P4::MergeData` object.

Class P4::Spec

Description

`P4::Spec` objects provide easy access to the attributes of the fields in a Helix server form.

The **`P4::Spec`** class uses Perl's AutoLoader to simplify form manipulation. Form fields can be accessed by calling a method with the same name as the field prefixed by an underscore (`_`).

Class Methods

$spec = new P4::Spec($fieldMap) -> array

Constructs a new **`P4::Spec`** object for a form containing the specified fields. (The object also contains a ***`fields`*** member that stores a list of field names that are valid in forms of this type.)

Instance Methods

$spec->_<fieldname> -> string

Returns the value associated with the field named <*fieldname*>.

```
$client = $p4->FetchClient( $clientname );
$client->_Root();    # Get client root
```

$spec->_<fieldname>($string)-> string

Updates the value of the named field in the spec.

```
$client = $p4->FetchClient( $clientname );
$client->_Root( $newroot );    # Set client root
```

$spec->PermittedFields() -> array

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.

```
my $client = $p4->FetchClient( $clientname );
my @permitted = $client->PermittedFields();
foreach $field (@permitted) {
print "$field\n";
}
```

P4Python

Introduction

P4Python, the Python interface to the Helix C/C++ API, enables you to write Python code that interacts with aHelix server. P4Python enables your Python scripts to:

- Get Helix Core data and forms in dictionaries and lists.
- Edit Helix Core forms by modifying dictionaries.
- Provide exception-based error handling and optionally ignore warnings.
- Issue multiple commands on a single connection (performs better than spawning single commands and parsing the results).

System Requirements and Release Notes

P4Python is supported on Windows, Linux, Solaris, OS X, and FreeBSD.

For system requirements, see the release notes at https://www.perforce.com/perforce/doc.current/user/p4pythonnotes.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:

```
anges = p4.run_changes("-uusername", "-m1").shift
```

If you include a space (`"-u username"`), the command fails.

Installing P4Python

Important
Before installing P4Python, any previously installed versions should be uninstalled.

As of P4Python 2015.1, the recommended mechanism for installing P4Python is via **`pip`**. For example:

```
$ pip install p4python
```

`pip` installs binary versions of P4Python where possible, otherwise it attempts to automatically build P4Python from source.

Windows users can download an installer containing pre-built packages for P4Python from the Perforce web site at https://www.perforce.com/downloads/helix-core-api-python.

Note
When P4Python is built without the `--apidir` option, setup attempts to connect to ftp.perforce.com to download the correct version of the P4API binary. If the P4API download is successful, it is unpacked into a temporary directory.

When P4Python is built and the `--ssl` is provided without a path, setup attempts to determine the correct path of the installed OpenSSL libraries by executing `openssl version`.

Programming with P4Python

P4Python provides an object-oriented interface to Helix Core that is intended to be intuitive for Python programmers. Data is loaded and returned in Python arrays and dictionaries. Each **P4** object represents a connection to the Helix server.

When instantiated, the **P4** instance is set up with the default environment settings just as the command line client **p4**, that is, using environment variables, the registry or user preferences (on Windows and OS X) and, if defined, the `P4CONFIG` file. The settings can be checked and changed before the connection to the server is established with the `P4.connect()` method. After your script connects, it can send multiple commands to the Helix server with the same **P4** instance. After the script is finished, it should disconnect from the server by calling the `P4.disconnect()` method.

The following example illustrates the basic structure of a P4Python script. The example establishes a connection, issues a command, and tests for errors resulting from the command.

```
from P4 import P4,P4Exception    # Import the module
p4 = P4()                        # Create the P4 instance
p4.port = "1666"
p4.user = "fred"
p4.client = "fred-ws"            # Set some environment variables

try:                             # Catch exceptions with try/except
  p4.connect()                   # Connect to the Perforce server
  info = p4.run( "info" )        # Run "p4 info" (returns a dict)
  for key in info[0]:            # and display all key-value pairs
    print key, "=", info[0][key]
  p4.run( "edit", "file.txt" )   # Run "p4 edit file.txt"
  p4.disconnect()                # Disconnect from the server
except P4Exception:
  for e in p4.errors:            # Display errors
      print e
```

This example creates a client workspace from a template and syncs it:.

```
from P4 import P4, P4Exception

template = "my-client-template"
client_root = "C:\work\my-root"
p4 = P4()

try:
  p4.connect()
  # Retrieve client spec as a Python dictionary
  client = p4.fetch_client( "-t", template )
  client._root = client_root
  p4.save_client( client )
  p4.run_sync()

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

Note
When extending the P4 class, be sure to match the method signatures used in the default class. P4Python uses both variable length arguments (`*args`) and keyword arguments (`**kwargs`). Review the P4.py in the source bundle for specifics. Example code:

```
class MyP4(P4.P4):
    def run(self, *args, **kargs):
        P4.P4.run(self, *args, **kargs)
```

Submitting a Changelist

This example creates a changelist, modifies it and then submits it:.

```
from P4 import P4

p4 = P4()
p4.connect()
change = p4.fetch_change()

# Files were opened elsewhere and we want to
# submit a subset that we already know about.

myfiles = ['//depot/some/path/file1.c', '//depot/some/path/file1.h']
change._description = "My changelist\nSubmitted from P4Python\n"
change._files = myfiles   # This attribute takes a Python list
p4.run_submit( change )
```

Logging into Helix Core using ticket-based authentication

On some servers, users might need to log in to Helix Core before issuing commands. The following example illustrates login using Helix Core tickets.

```
from P4 import P4

p4 = P4()
p4.user = "bruno"
p4.password = "my_password"
p4.connect()
p4.run_login()
opened = p4.run_opened()

...
```

Connecting to Helix Core over SSL

Scripts written with P4Python 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.

Changing your password

You can use P4Python to change your password, as shown in the following example:

```
from P4 import P4

p4 = P4()
p4.user = "bruno"
p4.password = "MyOldPassword"
```

```
p4.connect()

p4.run_password( "MyOldPassword", MyNewPassword" )

# p4.password is automatically updated with the encoded password
```

Timestamp conversion

Timestamp information in P4Python is normally represented as seconds since Epoch (with the exception of `P4.Revision`). To convert this data to a more useful format, use the following procedure:

```
import datetime

...

myDate = datetime.datetime.utcfromtimestamp( int( timestampValue ) )
```

Working with comments in specs

As of P4Python 2012.3, comments in specs are preserved in the `parse_<spectype>()` and `format_<spectype>()` methods. This behavior can be circumvented by using `parse_spec( '<spectype>', spec )` and `format_spec( '<spectype>', spec )` instead of `parse_<spectype>( spec )` and `format_<spectype>( spec )`. For example:

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

...

# fetch a client spec in raw format, no formatting:
specform = p4.run( 'client', '-o', tagged=False )[0]

# convert the raw document into a spec
client1 = p4.parse_client( specform )

# comments are preserved in the spec as well
print( client1.comment )

# comments can be updated
```

```
client1.comment += "# ... and now for something completely different"

# the comment is prepended to the spec ready to be sent to the user
formatted1 = p4.format_client( client1 )

# or you can strip the comments
client2 = p4.parse_spec( 'client', specform )
formatted2 = p4.format_spec( 'client', specform )
```

P4Python Classes

The **P4** module consists of several public classes:

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

The following tables provide more details about each public class, including methods and attributes. Attributes are readable and writable unless indicated otherwise. They can be strings, objects, or integers.

You can set attributes in the P4() constructor or by using their setters and getters. For example:

```
import P4
p4 = P4.P4(client="myclient", port="1666")
p4.user = 'me'
```

P4

Helix Core client class. Handles connection and interaction with the Helix server. There is one instance of each connection.

The following table lists attributes of the class **P4** in P4Python.

Attribute	Description
`api_level`	API compatibility level. (Lock server output to a specified server level.)
`charset`	Charset for Unicode servers.
`client`	`P4CLIENT`, the name of the client workspace to use.
`cwd`	Current working directory.
`disable_tmp_cleanup`	Disable cleanup of temporary objects.
`encoding`	Encoding to use when receiving strings from a non-Unicode server. If unset, use UTF8. Can be set to a legal Python encoding, or to `raw` to receive Python bytes instead of Unicode strings. Requires Python 3.
`errors`	An array containing the error messages received during execution of the last command.
`exception_level`	The exception level of the **P4** instance. Values can be: ▪ `0` : no exceptions are raised. ▪ `1` : only errors are raised as exceptions. ▪ `2` : warnings are also raised as exceptions. The default value is 2.
`handler`	An output handler.
`host`	`P4HOST`, the name of the host used.
`ignore_file`	The path of the ignore file, `P4IGNORE`.
`input`	Input for the next command. Can be a string, a list or a dictionary.
`maxlocktime`	MaxLockTime used for all following commands
`maxresults`	MaxResults used for all following commands
`maxscanrows`	MaxScanRows used for all following commands.
`messages`	An array of `P4.Message` objects, one for each message sent by the server.
`p4config_file`	The location of the configuration file used (`P4CONFIG`). This attribute is read-only.
`password`	`P4PASSWD`, the password used.
`port`	`P4PORT`, the port used for the connection.
`prog`	The name of the script.

Attribute	Description
`progress`	A progress indicator.
`server_case_insensitive`	Detect whether or not the server is case sensitive.
`server_level`	Returns the current Helix server level.
`server_unicode`	Detect whether or not the server is in Unicode mode.
`streams`	To disable streams support, set the value to `0` or `False`. By default, streams output is enabled for servers at 2011.1 or higher.
`tagged`	To disable tagged output for the following commands, set the value to 0 or False. By default, tagged output is enabled.
`track`	To enable performance tracking for the current connection, set the value to 1 or True. By default, server tracking is disabled.
`track_output`	If performance tracking is enabled, returns an array containing performance tracking information received during execution of the last command.
`ticket_file`	`P4TICKETS`, the ticket file location used.
`user`	`P4USER`, the user under which the connection is run.
`version`	The version of the script.
`warnings`	An array containing the warning messages received during execution of the last command.

The following table lists all public methods of the class `P4`. Many methods are wrappers around `P4.run()`, which sends a command to Helix server. Such methods are provided for your convenience.

Method	Description
`at_exception_level()`	In the context of a `with` statement, temporarily set the exception level for the duration of a block.
`clone()`	Clones from another Perforce service into a local Perforce service, and returns a new `P4` object.
`connect()`	Connects to the Helix server.
`connected()`	Returns `True` if connected and the connection is alive, otherwise `False`.

Method	Description
`delete_<spectype>()`	Deletes the spec <*spectype*>. Equivalent to: `P4.run( "<spectype>", "-d" )`
`disconnect()`	Disconnects from the Helix server.
`env()`	Get the value of a Helix Core environment variable, taking into account **`P4CONFIG`** files and (on Windows or OS X) the registry or user preferences.
`fetch_<spectype>()`	Fetches the spec <*spectype*>. Equivalent to: `p4.run( "<spectype>", "-o" ).pop( 0 )`
`format_<spectype>()`	Converts the spec <*spectype*> into a string.
`identify()`	Returns a string identifying the P4Python module.
`init()`	Initializes a new personal (local) Helix server, and returns a new **`P4`** object.
`is_ignored()`	Determines whether a particular file is ignored via the **`P4IGNORE`** feature.
`iterate_<spectype>()`	Iterate through specs of form <*spectype*>.
`P4()`	Returns a new **`P4`** object.
`parse_<spectype>()`	Parses a string representation of the spec <*spectype*> and returns a dictionary.
`run()`	Runs a command on the server. Needs to be connected, or an exception is raised.
`run_cmd()`	Runs the command *cmd*. Equivalent to: `P4.run( "command" )`
`run_filelog()`	This command returns a list of **`P4.DepotFile`** objects. Specialization for the **`P4.run()`** method.
`run_login()`	Logs in using the specified password or ticket.
`run_password()`	Convenience method: updates the password. Takes two arguments: *oldpassword*, *newpassword*

Method	Description
`run_resolve()`	Interface to `p4 resolve`.
`run_submit()`	Convenience method for submitting changelists. When invoked with a change spec, it submits the spec. Equivalent to: `p4.input = myspecp4.run( "submit", "-i" )`
`run_tickets()`	Interface to `p4 tickets`.
`save_<spectype>()`	Saves the spec <*spectype*>. Equivalent to: `P4.run( "<spectype>", "-i" )`
`set_env()`	On Windows or OS X, set a variable in the registry or user preferences.
`temp_client()`	Creates a temporary client.
`while_tagged()`	In the context of a `with` statement, temporarily toggle tagged behavior for the duration of a block.

P4.P4Exception

Exception class. Instances of this class are raised when errors and/or (depending on the `exception_level` setting) warnings are returned by the server. The exception contains the errors in the form of a string. `P4Exception` is a subclass of the standard Python `Exception` class.

P4.DepotFile

Container class returned by `P4.run_filelog()`. Contains the name of the depot file and a list of `P4.Revision` objects.

Attribute	Description
`depotFile`	Name of the depot file.
`revisions`	List of `P4.Revision` objects

P4.Revision

Container class containing one revision of a `P4.DepotFile` object.

Attribute	Description
`action`	Action that created the revision.
`change`	Changelist number
`client`	Client workspace used to create this revision.
`desc`	Short change list description.
`depotFile`	The name of the file in the depot.
`digest`	MD5 digest of the revision.
`fileSize`	File size of this revision.
`integrations`	List of `P4.Integration` objects.
`rev`	Revision.
`time`	Timestamp (as `datetime.datetime` object)
`type`	File type.
`user`	User that created this revision.

P4.Integration

Container class containing one integration for a `P4.Revision` object.

Attribute	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 Helix Core mappings without requiring a connection to the Helix server.

Method	Description
`P4.Map()`	Construct a new Map object (class method).
`join()`	Joins two maps to create a third (class method).

Method	Description
`clear()`	Empties a map.
`count()`	Returns the number of entries in a map.
`is_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.
`as_array()`	Returns the map as an array

P4.MergeData

Class encapsulating the context of an individual merge during execution of a `p4 resolve` command. Passed to `P4.run_resolve()`.

Attribute	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.

The `P4.MergeData` class also has one method:

`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

Class for handling error messages in Helix Core.

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.

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` to enable callbacks:

Method	Description
`init()`	Initialize progress indicator as designated type.
`setTotal()`	Total number of units (if known).
`setDescription()`	Description and type of units to be used for progress reporting.

Method	Description
`update()`	If non-zero, user has requested a cancellation of the operation.
`done()`	If non-zero, operation has failed.

P4.Resolver

Class for handling resolves in Helix Core.

Method	Description
`resolve()`	Perform a resolve and return the resolve decision as a string.

P4.Spec

Class allowing access to the fields in a Helix server specification form.

Attribute	Description
`_fieldname`	Value associated with the field named *fieldname*.
`comments`	Array containing comments in a spec object.
`permitted_fields`	Array containing the names of the fields that are valid for this spec object.

Class P4

Description

Main interface to the Python client API.

This module provides an object-oriented interface to Helix Core, the Perforce version constrol system. Data is returned in Python arrays and dictionaries (hashes) and input can also be supplied in these formats.

Each **P4** object represents a connection to the Helix server, and multiple commands may be executed (serially) over a single connection (which of itself can result in substantially improved performance if executing long sequences of Helix Core commands).

1. Instantiate your **P4** object.
2. Specify your Helix Core 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 Core protocol is not designed to support multiple concurrent queries over the same connection. Multithreaded applications that use the C++ API or derived APIs (including P4Python) 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 Core commands.
6. Disconnect from the Perforce service.

Instance Attributes

p4.api_level -> int

Contains the API compatibility level desired. This is useful when writing scripts using Helix Core 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 Core release and facilitate seamless upgrades. Must be called before calling `P4.connect()`.

```
from P4 import P4
p4 = P4()
p4.api_level = 67 # Lock to 2010.1 format
p4.connect()
...
p4.disconnect
```

For the API integer levels that correspond to each Helix Core release, see:

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

p4.charset -> string

Contains the character set to use when connecting 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`.

```
from P4 import P4
p4 = P4()
p4.client = "www"
p4.charset = "iso8859-1"
p4.connect()
p4.run_sync()
p4.disconnect()
```

p4.client -> string

Contains the name of your client workspace. By default, this is the value of the `P4CLIENT` taken from any `P4CONFIG` file present, or from the environment according to the normal Helix Core conventions.

p4.cwd -> string

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

```
from P4 import P4
p4 = P4()
p4.cwd = "/home/bruno"
```

p4.disable_tmp_cleanup -> string

Invoke this prior to connecting if you need to use multiple `P4` connections in parallel in a multi-threaded Python application.

```
from P4 import P4
p4 = P4()
p4.disable_tmp_cleanup()
p4.connect()
...
p4.disconnect()
```

p4.encoding -> string

When decoding strings from a non-Unicode server, strings are assumed to be encoded in UTF8. To use another encoding, set `p4.encoding` to a legal Python encoding, or `raw` to receive Python bytes instead of a Unicode string. Available only when compiled with Python 3.

p4.errors -> list (read-only)

Returns an array containing the error messages received during execution of the last command.

```
from P4 import P4, P4Exceptionp4 = P4()

try:
  p4.connect()
  p4.exception_level = 1
  # ignore "File(s) up-to-date"s
  files = p4.run_sync()

except P4Exception:
  for e in p4.errors:
    print e

finally:
  p4.disconnect()
```

p4.exception_level -> int

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

- `0` : disables all exception handling and makes the interface completely procedural; you are responsible for checking the `p4.errors` and `p4.warnings` arrays.
- `1` : causes exceptions to be raised only when errors are encountered.
- `2` : causes exceptions to be raised for both errors and warnings. This is the default.

For example:

```
from P4 import P4
p4 = P4()
p4.exception_level = 1
p4.connect()   # P4Exception on failure
p4.run_sync()  # File(s) up-to-date is a warning - no exception raised
p4.disconnect()
```

p4.handler -> handler

Set the output handler to a subclass of `P4.OutputHandler`.

p4.host -> string

Contains the name of the current host. It defaults to the value of `P4HOST` taken from any `P4CONFIG` file present, or from the environment as per the usual Helix Core convention. Must be called before connecting to the Helix server.

```
from P4 import P4
p4 = P4()
p4.host = "workstation123.perforce.com"
p4.connect()
...
p4.disconnect()
```

p4.ignore_file -> string

Contains the path of the ignore file. It defaults to the value of `P4IGNORE`. Set **`P4.ignore_file`** prior to calling `P4.is_ignored()`.

```
from P4 import P4
p4 = P4()
p4.connect()
p4.ignore_file = "/home/bruno/workspace/.ignore"
p4.disconnect()
```

p4.input -> string | dict | list

Contains input for the next command.

Set this attribute prior to running a command that requires input from the user. When the command requests input, the specified data is supplied to the command. Typically, commands of the form **`p4`** ***`cmd`*** **`-i`** are invoked using the **`P4.save_<`*`spectype`*`>()`** methods, which retrieve the value from **`p4.input`** internally; there is no need to set **`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 first element of the array will be popped each time Helix Core asks the user for input.

For example, the following code supplies a description for the default changelist and then submits it to the depot:

```
from P4 import P4
p4 = P4()
```

```
p4.connect()
change = p4.run_change( "-o" )[0]
change[ "Description" ] = "Autosubmitted changelist"
p4.input = change
p4.run_submit( "-i" )
p4.disconnect()
```

p4.maxlocktime -> int

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.maxresults -> int

Limit the number of results Helix Core 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.maxscanrows -> int

Limit the number of database records Helix Core scans 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.messages -> list (read-only)

Returns a list of `P4.Message` objects, one for each message (info, warning or error) sent by the server.

p4.p4config_file -> string (read-only)

Contains the name of the current `P4CONFIG` file, if any. This attribute cannot be set.

p4.password -> string

Contains your Helix Core password or login ticket. If not used, takes the value of `P4PASSWD` from any `P4CONFIG` file in effect, or from the environment according to the normal Helix Core conventions.

This password is also used if you later call `p4.run_login()` to log in using the 2003.2 and later ticket system. After running `p4.run_login()`, the attribute contains the ticket allocated by the server.

```
from P4 import P4
p4 = P4()
```

```
p4.password = "mypass"
p4.connect()
p4.run_login()
```

p4.port -> string

Contains the host and port of the Helix server to which you want to connect. It defaults to the value of `P4PORT` in any `P4CONFIG` file in effect, and then to the value of `P4PORT` taken from the environment.

```
from P4 import P4
p4 = P4()
p4.port = "localhost:1666"
p4.connect()
...
```

p4.prog -> string

Contains the name of the program, as reported to Helix Core system administrators running **`p4 monitor show -e`**. The default is `unnamed p4-python script`.

```
from P4 import P4
p4 = P4()
p4.prog = "sync-script"
print p4.prog
p4.connect
...
```

p4.progress -> progress

Set the progress indicator to a subclass of `P4.Progress`.

p4.server_case_insensitive -> boolean

Detects whether or not the server is case-sensitive.

p4.server_level -> int (read-only)

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.

This attribute is 0 before the first command is run, and is set automatically after the first communication with the server.

For the API integer levels that correspond to each Helix Core release, see:

http://kb.perforce.com/article/571

p4.server_unicode -> boolean

Detects whether or not the server is in Unicode mode.

p4.streams -> int

If 1 or True, `p4.streams` enables support for streams. By default, streams support is enabled at 2011.1 or higher (`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.

```
from P4 import P4
p4 = P4()
p4.streams = False
print p4.streams
```

p4.tagged -> int

If 1 or True, `p4.tagged` enables tagged output. By default, tagged output is on.

```
from P4 import P4
p4 = P4()
p4.tagged = False
print p4.tagged
```

p4.ticket_file -> string

Contains the location of the `P4TICKETS` file.

p4.track -> boolean

If set to 1 or True, `p4.track` indicates that server performance tracking is enabled for this connection. By default, performance tracking is disabled.

p4.track_output -> list (read-only)

If performance tracking is enabled with `p4.track`, returns an array containing the performance data received during execution of the last command.

```
from P4 import P4
p4 = P4()
p4.track = 1
```

```
p4.run_info()
print p4.track_output
```

p4.user -> string

Contains the Helix Core username. It defaults to the value of `P4USER` taken from any `P4CONFIG` file present, or from the environment as per the usual Helix Core convention.

```
from P4 import P4
p4 = P4()
p4.user = "bruno"
p4.connect()
...
p4.disconnect()
```

p4.version -> string

Contains the version of the program, as reported to Helix Core system administrators in the server log.

```
from P4 import P4
p4 = P4()
p4.version = "123"
print p4.version
p4.connect()
...
p4.disconnect()
```

p4.warnings -> list (read-only)

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

```
from P4 import P4, P4Exception
p4 = P4()

try:
  p4.connect()
  p4.exception_level = 2 # File(s) up-to-date is a warning
  files = p4.run_sync()

except P4Exception, ex:
  for w in p4.warnings:
      print w

finally:
  p4.disconnect()
```

Class Methods

P4.P4()

Construct a new **P4** object. For example:

```
import P4
p4 = P4.P4()
```

P4.clone(arguments...)

Clone from another Perforce service into a local Helix server, and returns a new **P4** object.

`P4.clone()` requires specification of the port of the source Perforce service from which files and version history should be cloned from, and either a *remotespec* or a *filespec* that specify which files and history to clone. For example, to clone using a *remotespec*:

```
import P4
p4 = P4.clone( "-p", "port", "-r", "remotespec" )
```

or to clone using a *filespec*:

```
import P4
p4 = P4.clone( "-p", "port", "-f", "filespec" )
```

The cloned instance inherits the case sensitivity and Unicode settings from the source Perforce service.

Note
All of the additional DVCS commands, such as `p4 push` or `p4 switch`, are available automatically in the usual fashion. For example: `p4.run_push()`. See p4.run_<cmd>() for details.

P4.identify()

Return the version of P4Python that you are using.

```
python -c "from P4 import P4; print P4.identify()"
```

The read-only string attributes `PATCHLEVEL` and `OS` are also available to test an installation of P4Python without having to parse the output of `P4.identify()`.

If applicable, `P4.identify()` also reports the version of the OpenSSL library used for building the underlying Helix C/C++ API with which P4Python was built.

P4.init([arguments])

Initializes a new, personal (local) Helix server, and returns a new **P4** object.

Without any arguments, `P4.init()` creates a new DVCS server in the current working directory, using the settings for case sensitivity and Unicode support from current environment variables.

`P4.init()` accepts the following keyword arguments:

Keyword	Explanation	Example
`client`	Workspace and server name	`client="sknop-dvcs"`
`user`	Helix Core username used for pushing	`user="sven_erik_knop"`
`directory`	local path of the root directory for the new server	`directory="/tmp/test-dvcs"`
`casesensitive`	specify case sensitivity	`casesensitive=False`
`unicode`	specify whether Unicode is enabled	`unicode=True`

```
import P4
p4 = P4.init( directory="/Users/sknop/dvcs/" )
p4.connect()
# ...
p4.disconnect()
```

The **P4** instance returned by `P4.init()` has the port, user, and client workspace already set; all that is required for you is to connect to the server to perform any commands. Connection is not automatic, to give you an opportunity to set any protocol parameters; these parameters can only be set once before a connection is established.

Note
All of the additional DVCS commands, such as `p4 push` or `p4 switch`, are available automatically in the usual fashion. For example: `p4.run_push()`. See p4.run_<cmd>() for details.

p4.iterate_<spectype>(arguments) -> P4.Spec

The `iterate_<spectype>()` methods are shortcut methods that allow you to quickly iterate through clients, labels, branches, etc. Valid <*spectypes*> 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:

```
for client in p4.iterate_clients():
  # do something with the client spec
```

is equivalent to:

```
for c in p4.run_clients():
  client = p4.fetch_client( c['client'] )
```

Instance Methods

p4.at_exception_level()

In the context of a `with` statement, temporarily set the exception level for a block. For example:

```
from P4 import P4
p4 = P4()
p4.connect()
with p4.at_exception_level( P4.RAISE_ERRORS ):
  # no exceptions for warnings
  p4.run_sync( "//depot/main/..." )

# exceptions back to normal...
```

p4.connect()

Initializes the Helix Core client and connects to the server.

If the connection is successfully established, returns `None`. If the connection fails and `P4.exception_level` is 0, returns False, otherwise raises a `P4Exception`. If already connected, prints a message.

```
from P4 import P4
p4 = P4()
p4.connect()
...
p4.disconnect()
```

`P4.connect()` returns a context management object that is usable with a `with` statement within a block; after the block is finished, the connection is automatically disconnected:

```
import P4
p4 = P4.P4()
with p4.connect():
  # block in context of connection
  ...

# p4 is disconnected outside the block
...
```

p4.connected() -> boolean

Returns `true` if connected to the Helix server and the connection is alive, otherwise `false`.

```
from P4 import P4
p4 = P4()

print p4.connected()
p4.connect()
print p4.connected()
```

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

The `delete_<spectype>()` methods are shortcut methods that allow you to delete the definitions of clients, labels, branches, etc. These methods are equivalent to:

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

The following code uses `P4.delete_client()` to delete client workspaces that have not been accessed in more than 365 days:

```
from P4 import P4, P4Exception
from datetime import datetime, timedelta

now = datetime.now()
p4 = P4()

try:
  p4.connect()
  for client in p4.run_clients():
    atime = datetime.utcfromtimestamp( int( client[ "Access" ] ) )
    # If the client has not been accessed for a year, delete it
    if ( atime + timedelta( 365 ) ) < now :
      p4.delete_client( '-f', client[ "client" ] )

except P4Exception:
  for e in p4.errors:
    print e

finally:
  p4.disconnect()
```

p4.disconnect()

Disconnect from the Helix server. Call this method before exiting your script.

```
from P4 import P4
p4 = P4()

p4.connect()
...
p4.disconnect()
```

p4.env(var)

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

```
from P4 import P4
p4 = P4()

print p4.env( "P4PORT" )
```

p4.fetch_<spectype>() -> P4.Spec

The `fetch_<spectype>()` methods are shortcuts for running `p4.run( "<spectype>", "-o" ).pop( 0 )`. For example:

```
label      = p4.fetch_label( "labelname" )
change     = p4.fetch_change( changeno )
clientspec = p4.fetch_client( "clientname" )
```

are equivalent to:

```
label      = p4.run( "label", "-o", "labelname" )[0]
change     = p4.run( "change", "-o", changeno )[0]
clientspec = p4.run( "client", "-o", "clientname" )[0]
```

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

Converts the fields in the dict 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>( dict )` instead of `p4.format_spec( "<spectype>", dict)`, where <*spectype*> is the name of a Helix server spec, such as client, label, etc.

p4.format_<spectype>(dict) -> string

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>", dict )
```

p4.is_ignored("<path>") -> boolean

Returns `true` if the <*path*> is ignored via the `P4IGNORE` feature. The <*path*> can be a local relative or absolute path.

```
from P4 import P4
p4 = P4()

p4.connect()
if ( p4.is_ignored( "/home/bruno/workspace/file.txt" ):
  print "Ignored."
else:
  print "Not ignored."

p4.disconnect()
```

p4.parse_spec("<spectype>", string) -> P4.Spec

Parses a Helix server form (spec) in text form into a Python dict 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.

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.parse_<spectype>(string) -> P4.Spec

This is equivalent to:

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

For example, `parse_job( myJob )` converts the String representation of a job spec into a Spec object.

To parse a spec, `P4` needs to have the spec available. When not connected to the Helix server, `P4` assumes the default format for the spec, which is hardcoded. This assumption can fail for jobs if the server's jobspec has been modified. In this case, your script can load a job from the server first with the command `p4.fetch_job( 'somename' )`, and `P4` will cache and use the spec format in subsequent `p4.parse_job()` calls.

p4.run("<cmd>", [arg, ...])

Base interface to all the run methods in this API. Runs the specified Helix Core command with the arguments supplied. Arguments may be in any form as long as they can be converted to strings by `str()`. 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 a list of results whether the command succeeds or fails; the list may, however, be empty. Whether the elements of the array are strings or dictionaries depends on:

1. server support for tagged output for the command, and
2. 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()` raises 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.

```
from P4 import P4
p4 = P4()
p4.connect()
spec = p4.run( "client", "-o" )[0]
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. For example, this:

```
from P4 import P4
p4 = P4()
p4.connect()
clientspec = p4.run_client( "-o" ).pop( 0 )
clientspec[ "Description" ] = "Build client"
p4.input = clientspec
p4.run_client( "-i" )
p4.disconnect()
```

…may be shortened to:

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

The following are equivalent:

Shortcut	Equivalent to
`p4.delete_<spectype>()`	`p4.run( "<spectype>", "-d ")`
`p4.fetch_<spectype>()`	`p4.run( "<spectype>", "-o ").shift`

`p4.save_<spectype>( spec )`	`p4.input = spec`
	`p4.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:

```
from P4 import P4
p4 = P4()
p4.connect()

spec = p4.fetch_change()
spec[ "Description" ] = "Automated change"
```

```
p4.run_submit( spec )
p4.disconnect()
```

p4.run_<cmd>()

Shorthand for:

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

p4.run_filelog(<fileSpec>) -> list

Runs a **`p4 filelog`** on the *fileSpec* provided and returns an array of **`P4.DepotFile`** results (when executed in tagged mode), or an array of strings when executed in nontagged 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.

For example:

```
from P4 import P4, P4Exception
p4 = P4()

try:
  p4.connect()
  for r in p4.run_filelog( "index.html" )[0].revisions:
    for i in r.integrations:
      # Do something

except P4Exception:
  for e in p4.errors:
    print e

finally:
  p4.disconnect()
```

p4.run_login(<arg>...) -> list

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

p4.run_password(oldpass, newpass) -> list

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

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

For example:

```
from P4 import P4, P4Exception
p4 = P4()
p4.password = "myoldpass"

try:
  p4.connect()
  p4.run_password( "myoldpass", "mynewpass" )

except P4Exception:
  for e in p4.errors:
    print e

finally:
  p4.disconnect()
```

p4.run_resolve([<resolver>], [arg…]) -> list

Run a `p4 resolve` command. Interactive resolves require the *<resolver>* parameter to be an object of a class derived from `P4.Resolver`. In these cases, the *P4.Resolver.resolve()* method is called to handle the resolve. For example:

```
p4.run_resolve ( resolver=MyResolver() )
```

To perform an automated merge that skips whenever conflicts are detected:

```
class MyResolver( P4.Resolver ):
  def resolve( self, mergeData ):
    if not mergeData.merge_hint == "e":
      return mergeData.merge_hint
    else:
      return "s" # skip the resolve, there is a conflict
```

In non-interactive resolves, no `P4.Resolver` object is required. For example:

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

p4.run_submit([hash], [arg...]) -> list

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( ) -> list

`p4.run_tickets()` returns an array of lists of the form (`p4port`, `user`, `ticket`) based on the contents of the local tickets file.

p4.save_<spectype>()>

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

For example:

```
from P4 import P4, P4Exception
p4 = P4()

try:
  p4.connect()
  client = p4.fetch_client()
  client[ "Owner" ] = p4.user
  p4.save_client( client )

except P4Exception:
  for e in p4.errors:
    print e

finally:
p4.disconnect()
```

p4.set_env(var, value)

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

p4.temp_client("<prefix>", "<template>")

Creates a temporary client, using the prefix <*prefix*> and based upon a client template named <*template*>, then switches `P4.client` to the new client, and provides a temporary root directory. The prefix makes is easy to exclude the workspace from the spec depot.

This is intended to be used with a `with` statement within a block; after the block is finished, the temp client is automatically deleted and the temporary root is removed.

For example:

```
from P4 import P4
p4 = P4()
p4.connect()
with p4.temp_client( "temp", "my_template" ) as t:
  p4.run_sync()
  p4.run_edit( "foo" )
  p4.run_submit( "-dcomment" )
```

p4.while_tagged(boolean)

In the context of a `with` statement, enable or disable tagged behavior for the duration of a block. For example:

```
from P4 import P4
p4 = P4()
p4.connect()
with p4.while_tagged( False ):
  # tagged output disabled for this block
  print p4.run_info()

# tagged output back to normal
...
```

Class P4.P4Exception

Description

Instances of this class are raised when `P4` encounters an error or a warning from the server. The exception contains the errors in the form of a string. `P4Exception` is a shallow subclass of the standard Python Exception class.

Class Attributes

None.

Class Methods

None.

Class P4.DepotFile

Description

Utility class providing easy access to the attributes of a file in a Helix Core 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 a list of `P4.DepotFile` objects.

Instance Attributes

df.depotFile -> string

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

df.revisions -> list

Returns a list of `P4.Revision` objects, one for each revision of the depot file.

Class Methods

None.

Instance Methods

None.

Class P4.Revision

Description

Utility class providing easy access to the revisions of `P4.DepotFile` objects. Created by `P4.run_filelog()`.

Instance Attributes

rev.action -> string

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

rev.change -> int

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

rev.client -> string

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

rev.depotFile -> string

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

rev.desc -> string

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 -> string

Returns the MD5 digest of this revision.

rev.fileSize -> string

Returns this revision's size in bytes.

rev.integrations -> list

Returns the list of `P4.Integration` objects for this revision.

rev.rev -> int

Returns the number of this revision of the file.

rev.time -> datetime

Returns the date/time that this revision was created.

rev.type -> string

Returns this revision's Helix Core filetype.

rev.user -> string

Returns the name of the user who created this revision.

Class Methods

None.

Instance Methods

None.

Class P4.Integration

Description

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

Instance Attributes

integ.how -> string

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

integ.file -> string

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

integ.srev -> int

Returns the start revision number used for this integration.

integ.erev -> int

Returns the end revision number used for this integration.

Class Methods

None.

Instance Methods

None.

Class P4.Map

Description

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

Instance Attributes

None.

Class Methods

P4.Map([list]) -> P4.Map

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

P4.Map.join (map1, map2) -> P4.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()
client_map.insert( "//depot/main/...", "//client/..." )

# Map client syntax to local syntax
client_root = P4.Map()
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()

Empty a map.

map.count() -> int

Return the number of entries in a map.

map.is_empty() -> boolean

Test whether a map object is empty.

map.insert(string ...)

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 (string, [boolean]) -> string

Translate a string through a map, and return the result. If the optional second argument is `1`, translate forward, and if it is `0`, translate in the reverse direction. By default, translation is in the forward direction.

map.includes(string) -> boolean

Tests whether a path is mapped or not.

```
if map.includes( "//depot/main/..." ):
  ...
```

map.reverse() -> P4.Map

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

map.lhs() -> list

Returns the left side of a mapping as an array.

map.rhs() -> list

Returns the right side of a mapping as an array.

map.as_array() -> list

Returns the map as an array.

Class P4.MergeData

Description

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

Instance Attributes

md.your_name -> string

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

md.their_name -> string

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

md.base_name -> string

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

md.your_path -> string

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

md.their_path -> string

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 `their_name` have been loaded.

md.base_path -> string

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 `base_name` have been loaded.

md.result_path -> string

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.

md.merge_hint -> string

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

Instance Methods

md.run_merge() -> boolean

If the environment variable `P4MERGE` is defined, `md.run_merge()` invokes the specified program and returns a boolean based on the return value of that program.

Class P4.Message

Description

`P4.Message` objects contain error or other diagnostic messages from the Helix server; they are returned in `P4.messages`.

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 Attributes

message.severity -> int

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 -> int

Returns the generic class of the error.

message.msgid -> int

Returns the unique ID of the message.

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`, use `p4.using_handler( MyHandler() )`, or pass the handler as a named parameter for one statement only.

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

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

Class Methods

class MyHandler(P4.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 progress class, set `P4.progress` to an instance of a subclass of `P4.Progress`, use `p4.using_progress( MyProgress() )`, or pass the progress indicator as a named parameter for one statement only.

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

Instance Attributes

None.

Class Methods

class MyProgress(P4.Progress)

Constructs a new subclass of `P4.Progress`.

Instance Methods

progress.init() -> int

Initialize progress indicator.

progress.setDescription(string, int) -> int

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

progress.update() -> int

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

progress.setTotal(<total>) -> int

Total number of units expected (if known).

progress.done() -> int

If non-zero, operation has failed.

Class P4.Resolver

Description

`P4.Resolver` is a class for handling resolves in Helix Core. It is intended to be subclassed, and for subclasses to override the `resolve()` method. When `P4.run_resolve()` is called with a `P4.Resolver` object, it calls the `P4.Resolver.resolve()` method of the object once for each scheduled resolve.

Instance Attributes

None.

Class Methods

None.

Instance Methods

resolver.resolve(self, mergeData) -> string

Returns the resolve decision as a string. The standard Helix Core resolve strings apply:

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

By default, all automatic merges are accepted, and all merges with conflicts are skipped. The `P4.Resolver.resolve()` method is called with a single parameter, which is a reference to a `P4.MergeData` object.

Class P4.Spec

Description

Utility class providing easy access to the attributes of the fields in a Helix server form.

Only valid field names may be set in a `P4.Spec` object. Only the field name is validated, not the content. Attributes provide easy access to the fields.

Instance Attributes

spec._<fieldname> -> string

Contains the value associated with the field named <*fieldname*>.

spec.comment -> dict

Contains an array containing the comments associated with the spec object.

spec.permitted_fields -> dict

Contains 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.

Class Methods

P4.Spec.new(dict) -> P4.Spec

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

Instance Methods

None.

P4PHP

Introduction

P4PHP, the PHP interface to the Helix C/C++ API, enables you to write PHP code that interacts with a Helix server. P4PHP enables your PHP scripts to:

- Get Helix Core data and forms in arrays.
- Edit Helix Core forms by modifying arrays.
- Provide exception-based error handling and optionally ignore warnings.
- Issue multiple commands on a single connection (performs better than spawning single commands and parsing the results).

System Requirements and Release Notes

P4PHP is supported on Windows, Linux, FreeBSD, and OS X.

For system requirements, see the release notes at https://www.perforce.com/perforce/doc.current/user/p4phpnotes.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:

```
anges = p4.run_changes("-uusername", "-m1").shift
```

If you include a space (`"-u username"`), the command fails.

Installing P4PHP

You can download P4PHP from the Perforce web site at https://www.perforce.com/downloads/helix-core-api-php.

You must build the interface from source, as described in the release notes packaged with P4PHP.

Programming with P4PHP

The following example illustrates the basic structure of a P4PHP script. The example establishes a connection, issues a command, and tests for errors resulting from the command.

```
<?php
```

```
$p4 = new P4();
$p4->port = "1666";
$p4->user = "fred";
$p4->client = "fred-ws";

try {
  $p4->connect();
  $info = $p4->run( "info" );
  foreach ( $info[0] as $key => $val ) {
    print "$key = $val\n";
  }
  $p4->run( "edit", "file.txt" );
  $p4->disconnect();
} catch ( P4_Exception $e ) {
  print $e->getMessage() . "\n";
  foreach ( $p4->errors as $error ) {
    print "Error: $error\n";
  }
}
?>
```

This example creates a client workspace from a template and syncs it:

```
<?php

$template = "my-client-template";
$client_root = "/home/user/work/my-root";
$p4 = new P4();

try {
  $p4->connect();

  // Convert client spec into an array

  $client = $p4->fetch_client( "-t", $template );
  $client['Root'] = $client_root;
  $p4->save_client( $client );
  $p4->run_sync();
} catch ( P4_Exception $e ) {
  // If any errors occur, we'll jump in here. Just log them
  // and raise the exception up to the higher level
}
?>
```

Submitting a Changelist

This example creates a changelist, modifies it, and then submits it:.

```
<?php

$p4 = new P4();
$p4->connect();

$change = $p4->fetch_change();

// Files were opened elsewhere and we want to
// submit a subset that we already know about.

$myfiles = array(
  '//depot/some/path/file1.c',
  '//depot/some/path/file1.h'
);

$change['description'] = "My changelist\nSubmitted from P4PHP\n";
$change['files'] = $myfiles;
$p4->run_submit( $change );

?>
```

Logging into Helix Core using ticket-based authentication

On some servers, users might need to log in to Helix Core before issuing commands. The following example illustrates login using Helix Core tickets.

```
<?php

$p4 = new P4();
$p4->user = "bruno";
$p4->connect();
$p4->run_login( 'my_password' );

$opened = $p4->
run_opened();

?>
```

Connecting to Helix Core over SSL

Scripts written with P4PHP 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.

Changing your password

You can use P4PHP to change your password, as shown in the following example:

```
<?php

$p4 = new P4();
$p4->user = "bruno";
$p4->password = "MyOldPassword";
$p4->connect();

$p4->run_password( "MyOldPassword", "MyNewPassword" );

// $p4->password is automatically updated with the encoded password

?>
```

P4PHP Classes

The P4 module consists of several public classes:

- P4
- P4_Exception
- P4_DepotFile
- P4_Revision
- P4_Integration
- P4_Map
- P4_MergeData
- P4_OutputHandlerAbstract
- P4_Resolver

The following tables provide more details about each public class.

P4

Helix Core client class. Handles connection and interaction with the Helix server. There is one instance of each connection.

The following table lists properties of the class **`P4`** in P4PHP. The properties are readable and writable unless indicated otherwise. The properties can be strings, arrays, or integers.

Property	Description
`api_level`	API compatibility level. (Lock server output to a specified server level.)
`charset`	Charset for Unicode servers.
`client`	`P4CLIENT`, the name of the client workspace to use.
`cwd`	Current working directory.
`errors`	A read-only array containing the error messages received during execution of the last command.
`exception_level`	The exception level of the P4 instance. Values can be: - `0` : no exceptions are raised - `1` : only errors are raised as exceptions - `2` : warnings are also raised as exceptions The default value is 2.

Property	Description
`expand_sequences`	Control whether keys with trailing numbers are expanded into arrays; by default, true, for backward-compatibility.
`handler`	An output handler.
`host`	`P4HOST`, the name of the host used.
`input`	Input for the next command. Can be a string, or an array.
`maxlocktime`	MaxLockTime used for all following commands.
`maxresults`	MaxResults used for all following commands.
`maxscanrows`	MaxScanRows used for all following commands.
`p4config_file`	The location of the configuration file used (`P4CONFIG`). This property is read-only.
`password`	`P4PASSWD`, the password used.
`port`	`P4PORT`, the port used for the connection
`prog`	The name of the script.
`server_level`	Returns the current Helix server level. This property is read only.
`streams`	Enable or disable support for streams.
`tagged`	To disable tagged output for the following commands, set the value to 0 or False. By default, tagged output is enabled.
`ticket_file`	`P4TICKETS`, the ticket file location used.
`user`	`P4USER`, the user under which the connection is run.
`version`	The version of the script.
`warnings`	A read-only array containing the warning messages received during execution of the last command.

The following table lists all public methods of the class `P4`.

Method	Description
`connect()`	Connects to the Helix server.
`connected()`	Returns `True` if connected and the connection is alive, otherwise `False`.

Method	Description
`delete_<spectype>()`	Deletes the spec <*spectype*>. Equivalent to the command: `P4::run( "<spectype>", "-d" );`
`disconnect()`	Disconnects from the Helix server.
`env()`	Get the value of a Helix Core environment variable, taking into account `P4CONFIG` files and (on Windows or OS X) the registry or user preferences.
`identify()`	Returns a string identifying the P4PHP module. (This method is static.)
`fetch_<spectype>()`	Fetches the spec <*spectype*>. Equivalent to the command: `P4::run( "<spectype>", "-o" );`
`format_<spectype>()`	Converts the spec <*spectype*> into a string.
`parse_<spectype>()`	Parses a string representation of the spec <*spectype*> and returns an array.
`run()`	Runs a command on the server. Needs to be connected, or an exception is raised.
`run_cmd()`	Runs the command *cmd*. Equivalent to: `P4::run( "cmd" );`
`run_filelog()`	This command returns an array of `P4_DepotFile` objects. Specialization for the `run()` command.
`run_login()`	Logs in using the specified password or ticket.
`run_password()`	Convenience method: updates the password. Takes two arguments: *oldpassword*, *newpassword*.
`run_resolve()`	Interface to `p4 resolve`.
`run_submit()`	Convenience method for submitting changelists. When invoked with a change spec, it submits the spec. Equivalent to: `p4::input = myspec;` `p4::run( "submit", "-i" );`

Method	Description
`save_<spectype>()`	Saves the spec <*spectype*>. Equivalent to the command: `P4::run( "<spectype>", "-i" );`

P4_Exception

Exception class. Instances of this class are raised when errors and/or (depending on the **`exception_level`** setting) warnings are returned by the server. The exception contains the errors in the form of a string. **`P4_Exception`** extends the standard PHP **`Exception`** class.

P4_DepotFile

Container class returned by **`P4::run_filelog()`**. Contains the name of the depot file and an array of **`P4_Revision`** objects.

Property	Description
`depotFile`	Name of the depot file
`revisions`	Array of Revision objects.

P4_Revision

Container class containing one revision of a **`P4_DepotFile`** object.

Property	Description
`action`	Action that created the revision.
`change`	Changelist number.
`client`	Client workspace used to create this revision.
`desc`	Short changelist description.
`depotFile`	The name of the file in the depot.
`digest`	MD5 digest of the revision.
`fileSize`	File size of this revision.
`integrations`	Array of **`P4_Integration`** objects.
`rev`	Revision.
`time`	Timestamp.

Property	Description
`type`	File type.
`user`	User that created this revision.

P4_Integration

Container class containing one integration for a `P4_Revision` object.

Property	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 Helix Core mappings without requiring a connection to the Helix server.

Method	Description
__construct()	Construct a new Map object.
join()	Joins two maps to create a third (static method).
clear()	Empties a map.
count()	Returns the number of entries in a map.
is_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.
as_array()	Returns the map as an array.

P4_MergeData

Class encapsulating the context of an individual merge during execution of a `p4 resolve` command. Passed to `P4::run_resolve()`.

Property	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.

P4_OutputHandlerAbstract

Handler class that provides access to streaming output from the server; set `$p4->handler` to an instance of a subclass of `P4_OutputHandlerAbstract` 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_Resolver

Abstract class for handling resolves in Perforce. This class must be subclassed in order to be used.

Method	Description
`resolve()`	Perform a resolve and return the resolve decision as a string.

Class P4

Description

Main interface to the PHP client API.

This module provides an object-oriented interface to Helix Core, the Perforce version control system. Data is returned in arrays and input can also be supplied in these formats.

Each **P4** object represents a connection to the Helix server, and multiple commands may be executed (serially) over a single connection (which of itself can result in substantially improved performance if executing long sequences of Helix Core commands).

1. Instantiate your **P4** object.
2. Specify your Helix Core 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 Core protocol is not designed to support multiple concurrent queries over the same connection. Multithreaded applications that use the C++ API or derived APIs (including P4PHP) 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 Core commands.
6. Disconnect from the Perforce service.

Properties

P4::api_level -> int

Contains the API compatibility level desired. This is useful when writing scripts using Helix Core 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 Core release and facilitate seamless upgrades. Must be called before calling `P4::connect()`.

```
<?php

$p4 = new P4();
$p4->api_level = 57; // Lock to 2005.1 format
$p4->connect();

...

$p4->disconnect();

?>
```

For the API integer levels that correspond to each Helix Core release, see:

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

P4::charset -> string

Contains 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 `P4_Exception`.

```
<?php

$p4 = new P4();
$p4->client = "www";
$p4->charset = "iso8859-1";

$p4->connect();
$p4->run_sync();
$p4->disconnect();

?>
```

P4::client -> string

Contains the name of your client workspace. By default, this is the value of the `P4CLIENT` taken from any `P4CONFIG` file present, or from the environment according to the normal Helix Core conventions.

P4::cwd -> string

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

```
<?php

$p4 = new P4();
$p4->cwd = "/home/bruno"

?>
```

P4::errors -> array (read-only)

Returns an array containing the error messages received during execution of the last command.

```
<?php

$p4 = new P4();
$p4->connect();
$p4->exception_level = 1;
$p4->connect();  // P4_Exception on failure
$p4->run_sync(); // File(s) up-to-date is a warning; no exception raised

$err = $p4->errors;
print_r( $err );

$p4->disconnect();

?>
```

P4::exception_level -> int

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

- `0` : disables all exception handling and makes the interface completely procedural; you are responsible for checking the `P4::errors` and `P4::warnings` arrays.
- `1` : causes exceptions to be raised only when errors are encountered.
- `2` : causes exceptions to be raised for both errors and warnings. This is the default.

For example:

```
<?php

$p4 = new P4();
$p4->exception_level = 1;
$p4->connect();  // P4_Exception on failure
$p4->run_sync(); // File(s) up-to-date is a warning; no exception raised
$p4->disconnect();

?>
```

P4::expand_sequences -> bool

Controls whether keys with trailing numbers are expanded into arrays when using tagged output. By default, `expand_sequences` is `true` to maintain backwards compatibility. Expansion can be enabled and disabled on a per-command basis.

For example:

```
<?php

$p4 = new P4();
$p4->connect();
$p4->expand_sequences = false; // disables sequence expansion.
$result = $p4->run( 'fstat', '-Oa', '//depot/path/...' );
var_dump( $result );

?>
```

P4::handler -> handler

Contains the output handler.

P4::host -> string

Contains the name of the current host. It defaults to the value of `P4HOST` taken from any `P4CONFIG` file present, or from the environment as per the usual Helix Core convention. Must be called before connecting to the Helix server.

```
<?php

$p4 = new P4();
$p4->host = "workstation123.perforce.com";
$p4->connect();

?>
```

P4::input -> string | array

Contains input for the next command.

Set this property prior to running a command that requires input from the user. When the command requests input, the specified data is supplied to the command. Typically, commands of the form **p4 *cmd* -i** are invoked using the `P4::save_<spectype>()` methods, which retrieve the value from **P4::input** internally; there is no need to set **P4::input** when using the `P4::save_<spectype>()` shortcuts.

You may pass a string, an array, or (for commands that take multiple inputs from the user) an array of strings or arrays. If you pass an array, note that the first element of the array will be popped each time Helix Core asks the user for input.

For example, the following code supplies a description for the default changelist and then submits it to the depot:

```
<?php

$p4 = new P4();
$p4->connect();

$change = $p4->run_change( "-o" )[0];
$change[ 'Description' ] = "Autosubmitted changelist";
$p4->input = $change;
$p4->run_submit( "-i" );
$p4->disconnect();

?>
```

P4::maxlocktime -> int

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::maxresults -> int

Limit the number of results Helix Core 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::maxscanrows -> int

Limit the number of database records Helix Core scans 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::p4config_file -> string (read-only)

Contains the name of the current `P4CONFIG` file, if any. This property cannot be set.

P4::password -> string

Contains your Helix Core password or login ticket. If not used, takes the value of `P4PASSWD` from any `P4CONFIG` file in effect, or from the environment according to the normal Helix Core conventions.

This password is also used if you later call `P4::run_login()` to log in using the 2003.2 and later ticket system. After running `P4::run_login()`, the property contains the ticket the allocated by the server.

```
<?php

$p4 = new P4();
$p4->password = "mypass";
$p4->connect();
$p4->run_login();

...

$p4->disconnect();

?>
```

P4::port -> string

Contains the host and port of the Helix server to which you want to connect. It defaults to the value of `P4PORT` in any `P4CONFIG` file in effect, and then to the value of `P4PORT` taken from the environment.

```
<?php

$p4 = new P4();
$p4->port = "localhost:1666";
$p4->connect();

...

$p4->disconnect();

?>
```

P4::prog -> string

Contains the name of the program, as reported to Helix Core system administrators running **`p4 monitor show -e`**. The default is **`unnamed p4-php script`**

```
<?php

$p4 = new P4();
$p4->prog = "sync-script";
print $p4->prog;
$p4->connect();

...

$p4->disconnect();

?>
```

P4::server_level -> int (read-only)

Returns the current Helix serverlevel. 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.

This property is 0 before the first command is run, and is set automatically after the first communication with the server.

For the API integer levels that correspond to each Helix Core release, see:

http://kb.perforce.com/article/571

P4::streams -> bool

If `true`, `P4::streams` enables support for streams. By default, streams support is enabled at 2011.1 or higher (`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.

```
<?php

$p4 = new P4();
$p4->streams = false;
print $p4->streams;

?>
```

P4::tagged -> bool

If `true`, `P4::tagged` enables tagged output. By default, tagged output is on.

```
<?php

$p4 = new P4();
$p4->tagged = false;
print $p4->tagged;

?>
```

P4::ticket_file -> string

Contains the location of the `P4TICKETS` file.

P4::user -> string

Contains the Helix Core username. It defaults to the value of `P4USER` taken from any `P4CONFIG` file present, or from the environment as per the usual Helix Core convention.

```
<?php
```

```
$p4 = new P4();
$p4->user = "bruno";
$p4->connect();
...
P4::disconnect();

?>
```

P4::version -> string

Contains the version of the program, as reported to Helix Core system administrators in the server log.

```
<?php

$p4 = new P4();
$p4->version = "123";
print $p4->version;
$p4->connect();
...
$p4->disconnect();

?>
```

P4::warnings -> array (read-only)

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

```
<?php

$p4 = new P4();
$p4->connect();  // P4_Exception on failure
$p4->exception_level = 2;

$files = $p4->run_sync();
$warn = $p4->warnings;
print_r( $warn );

$p4->disconnect();
```

```
?>
```

Constructor

P4::__construct

Construct a new **P4** object. For example:

```
<?php

$p4 = new P4();

?>
```

Static Methods

P4::identify() -> string

Return the version of P4PHP that you are using, and, if applicable, the version of the OpenSSL library used for building the underlying Helix C/C++ API with which P4PHP was built).

```
<?php

print P4::identify();

?>
```

produces output similar to the following:

```
Perforce - The Fast Software Configuration Management System.
Copyright 1995-2013 Perforce Software.  All rights reserved.
Rev. P4PHP/LINUX26X86/2013.1/644389 (2013.1 API) (2013/05/21).
```

Instance Methods

P4::connect() -> bool

Initializes the Helix Core client and connects to the server.

If the connection is successfully established, returns `None`. If the connection fails and `exception_level` is `0`, returns False, otherwise raises a `P4_Exception`. If already connected, prints a message.

```
<?php

$p4 = new P4();
$p4->connect();
...
$p4->disconnect();

?>
```

P4::connected() -> bool

Returns `true` if connected to the Helix server and the connection is alive, otherwise `false`.

```
<?php

$p4 = new P4();
if ( !$p4->connected() ) {
  print "Not Connected\n";
}

$p4->connect();
if ( $p4->connected() ) {
  print "Connected\n";
}

$p4->disconnect();

?>
```

P4::delete_<spectype>([options], name) -> array

The `delete_<spectype>()` methods are shortcut methods that allow you to delete the definitions of clients, labels, branches, etc. These methods are equivalent to:

```
P4::run( "<spectype>", '-d', [options], "spec name" );
```

The following code uses `P4::delete_client()` to delete client workspaces that have not been accessed in more than 365 days:

```
<?php

```

```
$p4 = new P4();
try {
  $p4->connect();
  foreach ( $p4->run_clients() as $client) {
    $atime = int( $client['Access'] );
    // If the client has not been accessed for a year, delete it
    if ( (time() - $atime) > 31536000 ) { // seconds in 365 days
      $p4->delete_client( "-f", $client["Client"] );
    }
  }
} catch ( P4_Exception $e ) {
  print $e->getMessage() . "\n";
  foreach ( $p4->errors as $error ) {
    print "Error: $error\n";
  }
}
?>
```

P4::disconnect() -> void

Disconnect from the Helix server. Call this method before exiting your script.

```
<?php

$p4 = new P4();
$p4->connect();
...
$p4->disconnect();

?>
```

P4::env(var) -> string

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

```
<?php

$p4 = new P4();
```

```
print $p4->env( "P4PORT" );

?>
```

P4::fetch_<spectype>() -> array

The `fetch_<spectype>()` methods are shortcuts for running `$p4->run( "<spectype>", "-o")` and returning the first element of the array. For example:

```
$label      = $p4->fetch_label( "labelname" );
$change     = $p4->fetch_change( changeno);
$clientspec = $p4->fetch_client( "clientname" );
```

are equivalent to:

```
$label      = $p4->run( "label", "-o", "labelname" );
$change     = $p4->run( "change", "-o", changeno );
$clientspec = $p4->run( "client", "-o", clientname );
```

P4::format_spec("<spectype>", array) -> string

Converts the fields in the array 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 `$p4->format_<spectype>( array)` instead of `$p4->format_spec( "<spectype>", array )`, where <*spectype*> is the name of a Helix server spec, such as client, label, etc.

P4::format_<spectype>(array) -> string

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>", array );
```

P4::parse_spec("<spectype>", string) -> array

Parses a Helix server form (spec) in text form into an array 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.

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::parse_<spectype>(string) -> array

This is equivalent to:

```
$p4->parse_spec( "<spectype>", string )
```

For example:

```
$p4->parse_job( myJob );
```

converts the String representation of a job spec into an array.

To parse a spec, **P4** needs to have the spec available. When not connected to the Helix server, **P4** assumes the default format for the spec, which is hardcoded. This assumption can fail for jobs if the server's jobspec has been modified. In this case, your script can load a job from the server first with the command `fetch_job( "somename")`, andP4 will cache and use the spec format in subsequent `P4::parse_job()` calls.

P4::run(<cmd>, [arg, ...]) -> mixed

Base interface to all the run methods in this API. Runs the specified Helix Core command with the arguments supplied. Arguments may be in any form as long as they can be converted to strings. 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 arrays depends on:

1. server support for tagged output for the command, and
2. 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()` raises a `P4_Exception`. 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.

```
<?php

$p4 = new P4();
print $p4->env( "P4PORT" );

$p4->connect();
```

```
$spec = $p4->run( "client", "-o" )[0];
$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. For example, this:

```
<?php

$p4 = new P4();
$p4->connect();

$clientspec = array_pop( $p4->run_client( "-o" ));
$clientspec["Description"] = "Build Client";

$p4->input = $clientspec;
$p4->run_client( "-i" );

$p4->disconnect();

?>
```

may be shortened to:

```
<?php

$p4 = new P4();
$p4->connect();

$clientspec = $p4->fetch_spec();
$clientspec["Description"] = "Build client";

$p4->save_client( $clientspec );
```

```
$p4->disconnect();

?>
```

The following are equivalent:

Shortcut	Equivalent to
`$p4->delete_<spectype>();`	`$p4->run( "<spectype>", "-d " );`
`$p4->fetch_<spectype>();`	`array_shift( $p4->run( "<spectype>", "-o " ) );`
`$p4->save_<spectype>( spec );`	`$p4->input = $spec; $p4->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:

```
<?php

$p4 = new P4();
$p4->connect();

$spec = $p4->fetch_change();
$spec["Description"] = "Automated change";
$p4->run_submit( $spec );

$p4->disconnect();

?>
```

P4::run_<cmd>() -> mixed

Shorthand for:

```
P4::run( "cmd", arguments... );
```

P4::run_filelog(<fileSpec>) -> array

Runs a **`p4 filelog`** on the *fileSpec* provided and returns an array of **`P4_DepotFile`** results (when executed in tagged mode), or an array of strings when executed in nontagged 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.

For example:

```
<?php

$p4 = new P4();
try {
  $p4->connect();
  $filelog = $p4->run_filelog( "index.html" );
  foreach ( $filelog->revisions as $revision ) {
    // do something
  }
} catch ( P4_Exception $e ) {
  print $e->getMessage() . "\n";
  foreach ( $p4->errors as $error ) {
    print "Error: $error\n";
  }
}
?>
```

P4::run_login(arg...) -> array

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

P4::run_password(oldpass, newpass) -> array

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

```
<?php

$p4->input = array( $oldpass, $newpass, $newpass );
$p4->run( "password" );

?>
```

For example:

```
<?php

$p4 = new P4();
$p4->password = "myoldpass";

try {
  $p4->connect();
  $p4->run_password( "myoldpass", "mynewpass" );
  $p4->disconnect();
} catch ( P4_Exception $e ) {
  print $e->getMessage() . "\n";
  foreach ( $p4->errors as $error ) {
    print "Error: $error\n";
  }
}
?>
```

P4::run_resolve([<resolver>], [arg...]) -> array

Run a `p4 resolve` command. Interactive resolves require the <*resolver*> parameter to be an object of a class derived from `P4_Resolver`. In these cases, the `P4::Resolver::resolve()` method is called to handle the resolve. For example:

```
<?php

$p4->run_resolve( new MyResolver() );

?>
```

To perform an automated merge that skips whenever conflicts are detected:

```
<?php

class MyResolver extends P4_Resolver {
  public function resolve( $merge_data ) {
    if ( $merge_data->merge_hint != 'e' ) {
      return $merge_data->merge_hint;
    } else {
      return "s"; // skip, there's a conflict
    }
  }
}
?>
```

In non-interactive resolves, no `P4_Resolver` object is required. For example:

```
$p4->run_resolve ( "-at" );
```

P4::run_submit([array], [arg…]) -> array

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

```
$p4->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::save_<spectype>()>

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 = $arrayOrString;
$p4->run( "<spectype> ", "-i" );
```

For example:

```
<?php

$p4 = new P4();
try {
  $p4->connect();
  $client = $p4->fetch_client();
  $client["Owner"] = $p4->user;
  $p4->save_client( $client );
  $p4->disconnect();
} catch ( P4_Exception $e ) {
  print $e->getMessage() . "\n";
  foreach ( $p4->errors as $error ) {
    print "Error: $error\n";
  }
}
?>
```

Class P4_Exception

Description

Instances of this class are raised when **`P4`** encounters an error or a warning from the server. The exception contains the errors in the form of a string. **`P4_Exception`** is an extension of the standard Exception class.

Class Attributes

None.

Static Methods

None.

Class P4_DepotFile

Description

Utility class providing easy access to the attributes of a file in a Helix Core depot. Each **`P4_DepotFile`** object contains summary information about the file and an array of revisions (**`P4_Revision`** objects) of that file. Currently, only the `P4::run_filelog()` method returns an array of **`P4_DepotFile`** objects.

Properties

$df->depotFile -> string

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

$df->revisions -> array

Returns an array of **`P4_Revision`** objects, one for each revision of the depot file.

Static Methods

None.

Instance Methods

None.

Class P4_Revision

Description

Utility class providing easy access to the revisions of `P4_DepotFile` objects. Created by `P4::run_filelog()`.

Properties

$rev->action -> string

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

$rev->change -> long

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

$rev->client -> string

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

$rev->depotFile -> string

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

$rev->desc -> string

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 -> string

Returns the MD5 digest of this revision.

$rev->fileSize -> long

Returns this revision’s size in bytes.

$rev->integrations -> array

Returns the array of `P4_Integration` objects for this revision.

$rev->rev -> long

Returns the number of this revision of the file.

$rev->time -> string

Returns the date/time that this revision was created.

$rev->type -> string

Returns this revision's Helix Core filetype.

$rev->user -> string

Returns the name of the user who created this revision.

Static Methods

None.

Instance Methods

None.

Class P4_Integration

Description

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

Properties

$integ->how -> string

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

$integ->file -> string

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

$integ->srev -> int

Returns the start revision number used for this integration.

$integ->erev -> int

Returns the end revision number used for this integration.

Static Methods

None.

Instance Methods

None.

Class P4_Map

Description

The **P4_Map** class allows users to create and work with Helix Core mappings, without requiring a connection to a Helix server.

Properties

None.

Constructor

P4_Map::__construct([array]) -> P4_Map

Constructs a new **P4_Map** object.

Static Methods

P4_Map::join (map1, map2) -> P4_Map

Join two **P4_Map** objects and create a third **P4_Map**. 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 = new P4_Map();
$client_map->insert( "//depot/main/...", "//client/..." );

// Map client syntax to local syntax
$client_root = new P4_Map();
$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() -> void

Empty a map.

$map->count() -> int

Return the number of entries in a map.

$map->is_empty() -> bool

Test whether a map object is empty.

$map->insert(string ...) -> void

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 (string, [bool])-> string

Translate a string through a map, and return the result. If the optional second argument is `1`, translate forward, and if it is `0`, translate in the reverse direction. By default, translation is in the forward direction.

$map->includes(string) -> bool

Tests whether a path is mapped or not.

```
if $map->includes( "//depot/main/..." ) {
  ...
}
```

$map->reverse() -> P4_Map

Return a new **`P4_Map`** object with the left and right sides of the mapping swapped. The original object is unchanged.

$map->lhs() -> array

Returns the left side of a mapping as an array.

$map->rhs() -> array

Returns the right side of a mapping as an array.

$map->as_array() -> array

Returns the map as an array.

Class P4_MergeData

Description

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

Properties

$md->your_name -> string

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

$md->their_name -> string

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

$md->base_name -> string

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

$md->your_path -> string

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

$md->their_path -> string

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 `their_name` have been loaded.

$md->base_path -> string

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 `base_name` have been loaded.

$md->result_path -> string

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.

$md->merge_hint -> string

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

Class P4_OutputHandlerAbstract

Description

The `P4_OutputHandlerAbstract` 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_OutputHandlerAbstract`.

By default, `P4_OutputHandlerAbstract` returns `HANDLER_REPORT` for all output methods. The different return options are:

Value	Meaning
`HANDLER_REPORT`	Messages added to output (don’t handle, don’t cancel).
`HANDLER_HANDLED`	Output is handled by class (don’t add message to output).
`HANDLER_CANCEL`	Operation is marked for cancel, message is added to output.

Class Methods

class MyHandler extends P4_OutputHandlerAbstract

Constructs a new subclass of `P4_OutputHandlerAbstract`.

Instance Methods

$handler->outputBinary -> int

Process binary data.

$handler->outputInfo -> int

Process tabular data.

$handler->outputMessage -> int

Process informational or error messages.

$handler->outputStat -> int

Process tagged data.

$handler->outputText -> int

Process text data.

Class P4_Resolver

Description

`P4_Resolver` is a class for handling resolves in Helix Core. It must be subclassed, to be used; subclasses can override the `P4::resolve()` method. When `P4::run_resolve()` is called with a `P4_Resolver` object, it calls the `P4_Resolver::resolve()` method of the object once for each scheduled resolve.

Properties

None.

Static Methods

None.

Instance Methods

$resolver->resolve(self, mergeData) -> string

Returns the resolve decision as a string. The standard Helix Core resolve strings apply:

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

By default, all automatic merges are accepted, and all merges with conflicts are skipped. The `P4_Resolver::resolve()` method is called with a single parameter, which is a reference to a **`P4_MergeData`** object.

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.

apple file type

Helix server file type assigned to files that are stored using AppleSingle format, permitting the data fork and resource fork to be stored as a single file.

archive

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

atomic change transaction

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

B

base

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

binary file type

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

branch

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

branch form

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

branch mapping

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

branch view

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

broker

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

C

change review

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

changelist

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

changelist form

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

changelist number

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

check in

To submit a file to the Helix Core depot.

check out

To designate one or more files for edit.

checkpoint

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

classic depot

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

client form

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

client name

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

client root

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

client side

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

client workspace

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

code review

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

codeline

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

comment

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

commit server

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

conflict

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

copy up

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

counter

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

D

default changelist

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

deleted file

In Helix server, a file with its head revision marked as deleted. Older revisions of the file are still available. in Helix Core, 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 versionsed files (depot files or source files) within a Helix Versioning Engine. It contains all versions of all files ever submitted to the depot. There can be multiple depots on a single installation.

depot root

The topmost (root) directory for a depot.

depot side

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

depot syntax

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

diff

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

donor file

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

E

edge server

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

exclusionary access

A permission that denies access to the specified files.

exclusionary mapping

A view mapping that excludes specific files or directories.

F

file conflict

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

file pattern

Helix Core 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 Core, 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 Core 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 Core commands. For example, you use the change form to enter comments about a particular changelist to verify the affected files.

forwarding replica

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

G

Git Fusion

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

graph depot

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

group

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

H

have list

The list of file revisions currently in the client workspace.

head revision

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

Helix server

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

Helix TeamHub

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

Helix4Git

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

I

integrate

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

J

job

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

job specification

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

job view

A syntax used for searching Helix server jobs.

journal

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

journal rotation

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

journaling

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

L

label

A named list of user-specified file revisions.

label view

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

lazy copy

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

license file

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

list access

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

local depot

Any depot located on the currently specified Helix server.

local syntax

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

lock

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

log

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

M

mapping

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

MDS checksum

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

merge

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

merge file

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

metadata

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

modification time or modtime

The time a file was last changed.

N

nonexistent revision

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

numbered changelist

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

O

opened file

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

owner

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

P

p4

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

p4d

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

pending changelist

A changelist that has not been submitted.

project

In Helix Swarm, a group of Helix Core 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 Core 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.

reresolve

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

resolve

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

reverse delta storage

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

revert

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

review access

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

revision number

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

revision range

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

revision specification

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

S

server data

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

server root

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

service

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

shelve

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

status

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

stream

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

stream depot

A depot used with streams and stream clients.

submit

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

super access

An access level that gives the user permission to run every Helix Core 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 Core 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 Core file type assigned to a file that contains only ASCII text, including Unicode text. See also binary file type.

theirs

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

three-way merge

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

trigger

A script automatically invoked Helix Core when various conditions are met.

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

view

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

W

wildcard

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

workspace

See client workspace.

workspace view

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

write access

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

Y

yours

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

License Statements