Scripting with Perforce

Using the Perl and Ruby interfaces

LAS VEGAS
PERFORCE
USER CONFERENCE
2003

Introduction

- What are P4Perl and P4Ruby
  - Perl & Ruby bindings for the Perforce C++ API
  - Object-Oriented
  - Interface designed to suit the language

Why Bother?

- Reasons to be cheerful…
  - Data returned as native objects: arrays and hashes
  - Support for both tagged mode and non-tagged mode
  - Smart form processing
  - Run many commands over a single connection.
  - Exception based error handling (P4Ruby)

Getting Started

- Before running commands
  - Load the module
  - Create an instance of the P4 class
  - Set options
  - Connect

Getting Started (Examples)

- Perl

```
use P4;
my $p4 = new P4;
$p4->Init() or die( “Can’t connect to Perforce” );
```

- Ruby

```
require “P4”
p4 = P4.new
p4.connect
```

Simple Usage

- Simple things are simple:
  - Running “p4 sync”

```
p4->Run( “sync”);          (Perl)
p4.run( “sync” )           (Ruby)
```

Command Shorthand

- Both P4Perl and P4Ruby have a shorthand syntax for executing Perforce commands
- They differ slightly due to the desire to make the interfaces feel natural to each language
- Makes P4Perl and P4Ruby less dependent on server version

Command Shorthand (Perl)

- All unknown methods are assumed to be Perforce commands
  - $p4->NewCommand();
  - Runs “p4 newcommand”
  - Implemented using Perl’s AUTOLOADER
- Fetch* and Save* commands are special
  - $p4->FetchXXX is equivalent to $p4->XXX(“-o”)
  - $p4->SaveXXX is equivalent to “$p4->XXX(”-I”)

Command Shorthand (Ruby)

- All unknown methods starting with “run_” are assumed to be Perforce commands
  - p4.run_newcommand
  - Runs “p4 newcommand”
  - Implemented by “P4#method_missing”
- fetch_* and save_* are special
  - fetch_xxx is equivalent to p4.run_xxx(“-o”).shift
  - save_xxx is equivalent to p4.run_xxx(“-i”).shift

Error Handling Introduction

- Not all errors are errors
  - Some are warnings
  - API users can test the severity of errors directly
  - P4Perl and P4Ruby distinguish between errors and warnings
- Commands may partially succeed/fail
  - May succeed with some files but not with others
  - Requires careful handling

Error Handling

- Perl
  - Requires explicit call to check for errors
  - Use P4::ErrorCount() to see how many errors occurred.
  - Use P4::Errors() to get errors as an array
- Ruby
  - Exceptions raised on errors and (optionally) warnings
  - Use P4#errors() to get errors as an array

Error Handling (Perl)

- Example

```
$p4->Sync();
  if ( $p4->ErrorCount() ) {
   foreach my $e ( $p4->Errors() ) {
          print( STDERR, $e, “\n” );
   }
  }
```

Error Handling (Ruby)

- Exception Levels
  - 0 = no exceptions raised at all
  - 1 = no exceptions on warnings
  - 2 = exceptions on both warnings and errors (default)
- Exceptions raised at command completion
  - Meaning that at least one error occurred

Error Handling (Ruby)

- Using Exception Level 1

```
p4.exception_level = 1
begin
    p4.run_sync
    p4.run_edit( "index.html"  )
    …
rescue P4Exception
    p4.errors.each { |e| $stderr.puts( e ) }
    raise
end
```

Overriding methods

- Shorthand methods can be easily overridden with custom implementations
  - Just define the method
  - Call the [Rr]un() method to execute the base command
  - Process the results as normal

Overriding Methods (Perl)

- Custom implementation of “p4 filelog”

```
use P4;
package P4;
sub Filelog {
	my $self = shift;
	my @results = $self->Run( “filelog”, @_ );
	# Post process @results
	return @results;
}
package main;
```

Overriding Methods (Ruby)

- Custom implementation of “p4 filelog”

```
require “P4”
class P4
    def filelog( *args )
        results = self.run( “filelog”, args )
        # Post process results
        return results
    end
end
```

Tagged Mode

- Tagged data from server is returned as a hash
- Allows direct access to the data you are interested in without having to parse the output

Tagged Mode (Example1)

- Perl

```
my $fs     = $p4->Fstat( “file.c” );
my $head = $fs->{ “headRev” };
```

- Ruby

```
fs       = p4.run_fstat( “file.c” )
head = fs[ “headRev” ]
```

Tagged Mode (Example2)

- Perl

```
my $fs      = $p4->Fstat( “file.c” );
foreach my( $key, $value) ( @$fs ) {
  print( $key, “ → “ $value );
}
```

Tagged Mode (Example2)

- Ruby

```
p4.run_fstat( “file.c” ).each do
  |key,value|
  puts( key + “ → “ + value )
end
```

Form Handling

- Both P4Perl and P4Ruby can convert Perforce forms into hashes
- Both can also convert hashes back into Perforce forms
- Editing a clientspec or a changelist is as simple as updating a hash and saving your changes

Form Handing (Perl)

- Example

```
my $c = $p4->FetchChange();
$c->{ “Description” } = “some text…”;
$p4->SaveChange( $c );
```

Form Handling (Ruby)

- Example 1

```
c = p4.fetch_change
c[ “Description” ] = “some text…”
p4.save_change( c )
```

- Example 2

```
c = p4.fetch_client
c[ “Root” ] = ‘d:\work’
c[ “Options” ].sub!( “normdir”, “rmdir”)
p4.save_client( c )
```

Language Wars

P4Perl vs. P4Ruby

- Functionally equivalent
- Big difference is Exception base error handling in P4Ruby
  - Smaller, more reliable code
  - Handles warnings
    - (e.g. “File(s) up-to-date”)
- Some extra support for handling “p4 filelog” output in P4Ruby
- Ruby is much nicer than Perl. Try it!

P4Perl/P4Ruby vs. p4 -G

- Mostly personal preference
  - Multiple commands per connection
  - Separation of output and error streams
  - Not Python! ☺

Questions?

- Neither P4Perl nor P4Ruby is supported by Perforce Software.
- Both are supported by me personally
- Questions, comments etc. to me directly at either
  - tony@perforce.com or
  - tony@smee.org