Episode 280

Pry With Rails

Pry[1] is an alternative to IRB and like IRB it provides a prompt for executing Ruby code. What makes it different is that it comes with many extra features. In this episode we'll show you how Pry works and how to integrate it into your Rails applications.

Pry comes as a gem and is easy to install. We'll install the `pry-doc` gem along with it and you'll see more about this later.

```
$ gem install pry pry-doc
```

As we're using RVM gemsets we'll install Pry so that it's available globally across all gemsets. We can do this with the following two commands.

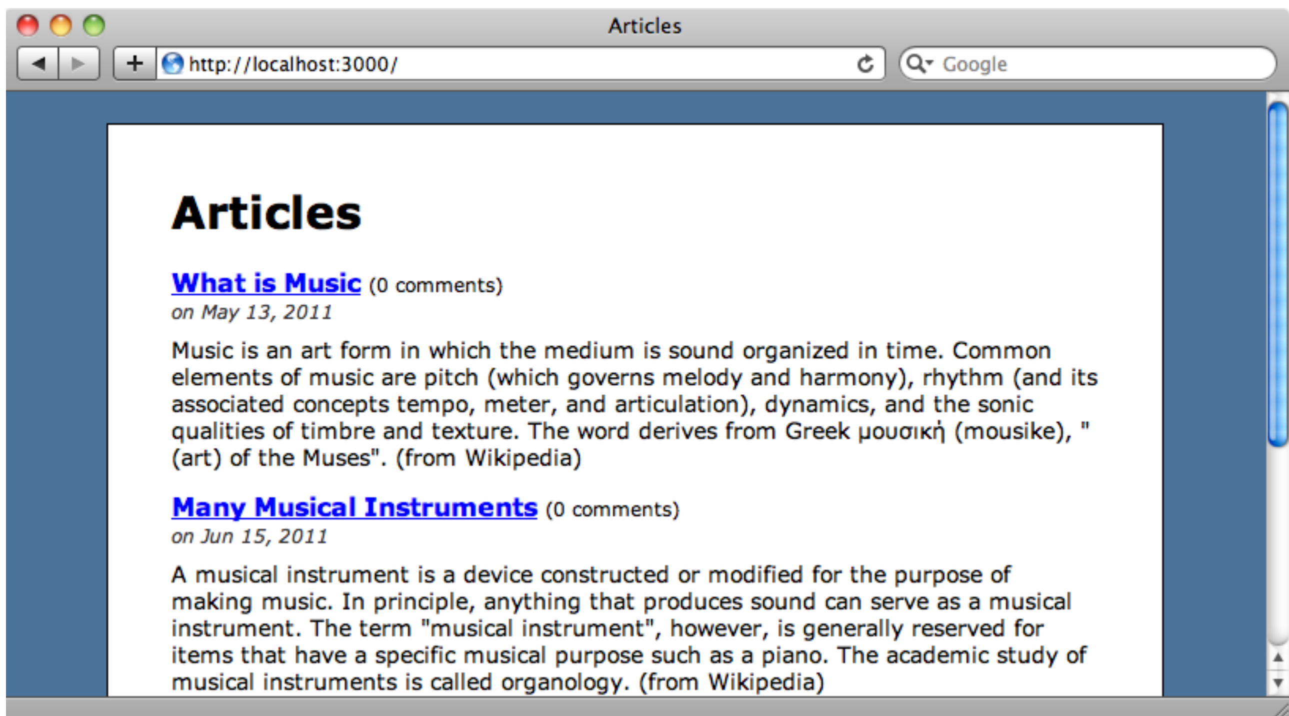
```
$ rvm gemset use global
$ gem install pry pry-doc
```

Once we have Pry installed we can run it with the `pry` command and execute Ruby code just as we would with `irb`.

```
$ pry
pry(main)> 1 + 2
=> 3
```

Pry is much more than a simple calculator but before we go any further with it we'll look at setting it up to work with a Rails application. The application we'll use is the familiar blogging application we've used in several previous episodes.

[1] http://pry.github.com/



If we run `rails c` from the application's directory IRB will start up. To use Pry instead all we need to do is run `pry` and pass in the Rails environment file. When we do we'll have access to our application's models just as we would in the standard Rails console.

```
$ pry -r ./config/environment
pry(main)> Article.count
=> 3
```

Now that we have Pry set up we can take a look at some of its features. If we type `help` we'll get a list of all of the commands that Pry supports. The two we'll use most often are `cd` and `ls` so let's look at those. The `cd` command changes the current scope, so if we type `cd Article` we'll move into the `Article` class and we can check the current scope at any time by running `self`.

```
pry(main)> cd Article
pry(#<Class:0x1022f60e0>):1> self
=> Article(id: integer, name: string, content: text, created_at: 
datetime, updated_at: datetime, published_at: datetime)
```

Now that we're inside the `Article` class we can call any of its methods, such as `first` which will return the first article, just as if we'd typed `Article.first`.

```
pry(#<Class:0x1022f60e0>):1> first
=> #<Article id: 1, name: "What is Music", content: "Music is an art form in which the medium is sound o...", created_at: "2011-08-24 20:35:29", updated_at: "2011-08-24 20:37:22", published_at: "2011-05-13 23:00:00">
```

We can also `cd` into any object so if we run `cd first` while in the `Article` scope we'll change the scope to that article. We can then call methods and properties on it, such as `name`.

```
pry(#<Class:0x1022f60e0>):1> cd first
pry(#<Article:0x102300c98>):2> name
=> "What is Music"
```

We can even `cd` into the article's `name` and call methods on that string.

```
pry(#<Article:0x102300c98>):2> cd name
pry("What is Music"):3> upcase
=> "WHAT IS MUSIC"
```

Pry keeps track of where we've gone and we can see this by calling `nesting`.

```
pry("What is Music"):3> nesting
Nesting status:
--
0. main (Pry top level)
1. #<Class:0x1022f60e0>
2. #<Article:0x102300c98>
3. "What is Music"
```

This command returns the list of objects that we've gone in to. If we type `exit` this will pop us out back into the previous object, in this case the first article. If we exit again we'll be taken back into the `Article` class.

The other most-commonly used command is `ls` and this lists variables and methods. By default it lists all of the variables in the current scope, so if we're currently in the `Article` class we'll see its methods when we run it.

```
pry(#<Class:0x1022f60e0>):1> ls
[:_, :_pry_, :inp, :out, :@_create_callbacks,
:@_defined_class_methods, :@_save_callbacks, :@_update_callbacks,
:@_validate_callbacks, :@arel_engine, :@arel_table,
:@attribute_methods_generated, :@cached_attributes,
:@column_names, :@columns, :@columns_hash,
:@finder_needs_type_condition, :@generated_attribute_methods,
:@inheritable_attributes, :@inheritance_column, :@parent_name,
:@quoted_primary_key, :@quoted_table_name, :@relation]
```

Some Pry commands support flags and we can get a list of a command's flags by running the command with the `-h` option. If we run `ls -h` we'll see all of the options it supports, including `-m` which we can use to display a list of methods in the class and `-M` which returns a list of the instance methods. We can also pass in any object or class to get a list of methods for that rather than for the current scope.

Another useful Pry command is `show-doc`. Let's say that we want to know to the `Array` class's `in_groups_of` method works. We can find out by running `show-doc Array#in_groups_of`.

```
pry(#<Class:0x1022f60e0>):1> show-doc Array#in_groups_of

From: /Users/eifion/.rvm/gems/ruby-1.9.2-p290/gems/
activesupport-3.0.10/lib/active_support/core_ext/array/grouping.rb
@ line 19:
Number of lines: 15

signature: in_groups_of(number, fill_with=?)

Splits or iterates over the array in groups of size number,
padding any remaining slots with fill_with unless it is false.

  %w(1 2 3 4 5 6 7).in_groups_of(3) {|group| p group}
  ["1", "2", "3"]
  ["4", "5", "6"]
  ["7", nil, nil]

  %w(1 2 3).in_groups_of(2, ' ') {|group| p group}
  ["1", "2"]
  ["3", " "]

  %w(1 2 3).in_groups_of(2, false) {|group| p group}
  ["1", "2"]
  ["3"]
```

We can also call `show-doc` directly on objects. Our scope is currently the `Article` class so we can call `all` to return an array of the articles. We can run `show-doc all.in_groups_of` to return the same documentation as above.

Another useful command is `show-method`. This shows us the source code of any method. We can use this to see the source of `in_groups_of`. (It's worth pointing out here that when we're entering commands Pry has autocompletion that we can activate by pressing the TAB key.)

```
pry(#<Class:0x104e63de0>):1> show-method all.in_groups_of

From: /Users/eifion/.rvm/gems/ruby-1.9.2-p290/gems/
activesupport-3.0.10/lib/active_support/core_ext/array/grouping.rb
@ line 19:
Number of lines: 19

def in_groups_of(number, fill_with = nil)
  if fill_with == false
    collection = self
  else
    # size % number gives how many extra we have;
    # subtracting from number gives how many to add;
    # modulo number ensures we don't add group of just fill.
    padding = (number - size % number) % number
    collection = dup.concat([fill_with] * padding)
  end

  if block_given?
    collection.each_slice(number) { |slice| yield(slice) }
  else
    groups = []
    collection.each_slice(number) { |group| groups << group }
    groups
  end
end
```

Similarly we have an `edit-method` command. When we run this against a method it will open up the relevant source code file in a text editor and take us to the appropriate line.

We can also run shell commands by prefixing them with a full stop. If we run `.ls` in Pry this will run the `ls` shell command and list all of the files in the current directory.

Pry is also useful for debugging. In our `Article` model we have a `word_count` method that should return the number of words in an article's `content`. There's a bug in the method as it stands, however, so that it always returns `0`, no matter what the content in the article. We can look at the method by `cd`-ing to the first `Article` and then running `edit-method word_count`. This is what the method looks like:

/app/models/article.rb

```
class Article < ActiveRecord::Base
  attr_accessible :name, :content, :published_at
  has_many :comments

  def word_count
    words = content.scan(/\\w+/)
    words.size
  end
end
```

We can add a breakpoint to the code any point by calling `binding.pry`. If we add that immediately before the `words.size` line above and save the file then when we call the `word_count` method again it will stop running at `binding.pry` and we'll return to the Pry prompt.

```
> word_count

From: /Users/eifion/blog/app/models/article.rb @ line 7 in
Article#word_count:

     2:   attr_accessible :name, :content, :published_at
     3:   has_many :comments
     4:
     5:   def word_count
     6:     words = content.scan(/\\w+/)
 =>  7:     binding.pry
     8:     words.size
     9:   end
    10: end
```

We have access to all of the method's local variables here so we can call `words` to see the contents of the `words` array.

```
pry(#<Article:0x1008c3f38>):3> words
=> []
```

The array is empty so it appears that there's something wrong with the regular expression that scans the content. If we look at it we'll see that there are two backslashes there where there should be one. To fix this we can run `edit-method`

word_count again, fix the regular expression, remove the binding.pry line and save the file.

/app/models/article.rb

```
class Article < ActiveRecord::Base
  attr_accessible :name, :content, :published_at
  has_many :comments

  def word_count
    words = content.scan(/\w+/)
    words.size
  end
end
```

We can test our fix by calling word_count again and this time it works as expected.

```
pry(#<Article:0x1008c3f38>):3> word_count
=> 55
```

Sometimes we'll want to debug something in the controller or view layers of our application and not necessarily through the console. Pry can help with this too. First we'll need to add a reference to Pry in the Gemfile.

/Gemfile

```
source 'http://rubygems.org'

gem 'rails', '3.0.10'
gem 'sqlite3'
gem 'nifty-generators'
gem 'pry', :group => :development
```

When can now run bundle to install the gems then rails s to start up the server. We can now add calls to binding.pry anywhere in our controllers to add a breakpoint.

/app/controllers/articles_controller.rb

```
def index
  @articles = Article.all
  binding.pry
end
```

If we go to that page in a browser now it will "stick" while loading but inside the terminal we'll have a Pry prompt that has stopped at that breakpoint. As we could in the model we can inspect the values of any local or instance variables. Once we've finished we can type `exit-all` to allow the request to complete.

```
From: /Users/eifion/blog/app/controllers/articles_controller.rb @ line 4 in ArticlesController#index:

     1: class ArticlesController < ApplicationController
     2:   def index
     3:     @articles = Article.all
 =>  4:     binding.pry
     5:   end
     6:
     7:   def show
     8:     @article = Article.find(params[:id])
     9:   end
```

That's it for this episode on Pry. It's a really useful gem and does much more than we've covered here. The wiki[2] covers more than we've done here and includes a link to a very useful screencast[3] by Joshua Cheek.

[2] https://github.com/pry/pry/wiki

[3] http://vimeo.com/26391171