Extracted from:

Rails Recipes

Rails 3 Edition

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

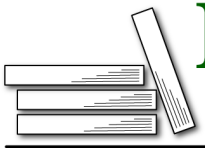
Rails Recipes

Rails 3 Edition

Chad Fowler

Edited by John Osborn

The Facets of Ruby Series



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

John Osborn (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-93435-677-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2012

Problem

Rails ships with a nice collection of model validators. You can use them to validate presence, numericality, format, and several other qualities commonly associated with attributes. Most of the time, these validators are enough to meet our needs. But sometimes they're not, such as when an application requires validation specific to a domain that the built-in validators can't handle and that we'd like to reuse elsewhere.

How do we create a clean, reusable custom validator for Rails?

Solution

The solution is to create and reference a subclass of ActiveModel::Validator.

In Rails 3, all of the fancy declarative validators are built on a single, configurable method called validates_with(). Under the covers, Rails uses this single configurable method to set up all validations instead of validation-specific methods such as validates_uniqueness_of(). As a shortcut to simplify the options, the class-level method validates() is provided to our models.

The validates() method allows us to specify multiple validations for a given attribute in one call. It uses naming conventions to locate the validators. The following code, for example, will ensure that instances of Person include a value for age that is an integer:

```
class Person < ActiveRecord::Base
  validates :age, :presence => true, :numericality => {:only_integer => true}
end
```

The validates() method takes one or more attributes and a Hash of validation options. The keys in these options are not hard-coded into Rails. They work from naming conventions. The name :presence is translated and resolved to the class name PresenceValidator. The name :numericality is resolved to the class name NumericalityValidator. The validates() (and its underlying validates_with()) has no knowledge of these specific validators. Let's look at an example of how we can use this to our advantage.

Imagine we had a Product model for which we wanted to validate the format of a stock-keeping unit (SKU) code. Let's say that in our business the SKUs consist of four uppercase ASCII letters, followed by a dash, followed by an eight-digit numeric code. We could declare this validation "manually" with the built-in validates_format_of() and a regular expression. But we're missing the beauty of Ruby and Rails: the ability it gives us to program close to the domain!

So, instead, we'll make a custom validator. This will give us a more declarative, domain-level representation of our validation as well as giving us the added benefit of being able to reuse the validation in other models or on other fields.

We'll start by defining our Product model and the validation for the sku field:

rr2/custom_validator/app/models/product.rb

```
class Product < ActiveRecord::Base
  validates :identifier, :sku => true
end
```

Let's start the console and take a look at the Product model:

```
>> Product
ArgumentError: Unknown validator: 'sku'
        from .../validations/validates.rb:87:in `rescue in block in validates'
        from .../validations/validates.rb:84:in `block in validates'
        from .../validations/validates.rb:83:in `each'
        from .../validations/validates.rb:83:in `validates'
        ....
```

Oops! We haven't created the validator for sku yet. This gives us some insight into how the validators are resolved. During the call to validates(), the validator is located and put in place. So, we need to define a validator that matches the expected naming convention for sku. Let's name this validator SkuValidator. We can define it anywhere as long as Rails loads it. Let's put it in app/models. If we name it using the usual Rails filenaming convention, sku_validator.rb, Rails will automatically find it without having to explicitly require() it. Here's the validator:

rr2/custom_validator/app/models/sku_validator.rb

```
class SkuValidator < ActiveModel::EachValidator
  def validate_each(record, attribute, value)
    record.errors[attribute] << (
      options[:message] || "is not a valid SKU code"
    ) unless
      value =~ /\A([A-Z]{4})-([0-9]{8})\z/i
  end
end
```

Our validator class inherits from ActiveModel::EachValidator, which is what most of the built-in validators inherit from. EachValidator's job is to iterate through the list of given attributes, calling validate_each() for each one. The validate_each() method takes the object being validated, the attribute name currently being validated, and the value assigned to that attribute. To signal a validation error, we simply add a message for the given attribute to the object's error list.

Now we can reload the console and interact with the model:

```
>> shampoo = Product.new(:name => "Glue Shampoo", :identifier => "shampoo!")
 => #<Product id: nil, name: "Glue Shampoo", identifier: "shampoo!">
>> shampoo.valid?
 => false
>> shampoo.errors.full_messages.to_sentence
 => "Identifier is not a valid SKU code"
>> shampoo.identifier = "ABCD-12345678"
 => "ABCD-12345678"
>> shampoo.valid?
 => true
```

Now that we've created the custom validator, we can use it in any class or future application that may need it. Even if we don't reuse it, we've separated validation logic from the rest of the model, making the code easier to understand and maintain.