

ROBERT PANKOWECKI

April 30, 2014

Mastering Rails Validations: Contexts



Many times Rails programmers ask *How can I skip one (or more) validations in Rails*. The common usecase for it is that users with higher permissions are granted less strict validation rules. After all, what's the point of being admin if admin cannot do more than normal user, right? *With great power comes great responsibility and all of that yada yada yada*. But back to the topic. Let's start with something simple and refactor it a little bit to show Rails feature that I rerly see in use in the real world.

This is our starting point

Where the fun begins


```
class User < ActiveRecord::Base
  validates_length_of :slug, minimum: 3
end
```

Our users can change the **slug** (/u/slug) under which their profiles will appear. However the most valuable short slugs are not available for them. Our business model dictates that we are going to sell them to **earn a lot of money** [disclaimer: polish joke, I could not resist]

So, we need to add conditional validation that will be different for admins and different for users. Nothing simpler, right?

Where the fun ends

```
class User < ActiveRecord::Base
  attr_accessor :edited_by_admin
  validates_length_of :slug, minimum: 3, unless: Proc.new{|u| u.edited_by_admin? }
  validates_length_of :slug, minimum: 1, if:      Proc.new{|u| u.edited_by_admin? }
end
```



```
class Admin::UsersController
  def edit
    @user = User.find(params[:id])
    @user.edited_by_admin = true
    if @user.save
      redirect # ...
    else
      render # ...
    end
  end
end
```

Now this would work, however it is not code I would be proud about.

But wait, you already know a way to mark validations to trigger only sometimes. Do you remember it?

```
class Meeting < ActiveRecord::Base
  validate :starts_in_future, on: :create
end
```

We've got `on: :create` option which makes a validation run only when saving new record (`#new_record?`).

I wonder whether we could use it...

Where it's fun again

```
class User < ActiveRecord::Base
  validates_length_of :slug, minimum: 3, on: :user
  validates_length_of :slug, minimum: 1, on: :admin
end
```

```
class Admin::UsersController
  def edit
    @user = User.find(params[:id])
    if @user.save(context: :admin)
      redirect # ...
    else
      render # ...
    end
  end
end
```

Wow, now look at that. Isn't it cute?

And if you want to only check validation without saving the object you can use:

```
u = User.new
u.valid?(:admin)
# or
u.valid?(:user)
```

This feature is actually even documented `ActiveModel::Validations#valid?(context=nil)`

Now it is a good moment to remind ourselves of a nice API that can make it less redundant in case of multiple rules: `Object#with_options`

```

class User < ActiveRecord::Base
  with_options({on: :user}) do |for_user|
    for_user.validates_length_of :slug, minimum: 3
    for_user.validates_acceptance_of :terms_of_service
  end

  with_options({on: :admin}) do |for_admin|
    for_admin.validates_length_of :slug, minimum: 1
  end
end

```

When it's miserable again

The problem with this approach is that you cannot supply multiple contexts.

If you would like to have some validations on: :admin and some on: :create then it is probably not gonna work the way you would want.

```

class User < ActiveRecord::Base
  validates_length_of :slug, minimum: 3, on: :user
  validates_length_of :slug, minimum: 1, on: :admin
  validate :something, on: :create
end

```

When you run `user.valid?(:admin)` or `user.save(context: admin)` for new record, it's not gonna trigger the last validation because we substituted the default :create context with our own :admin context.

You can [see it for yourself in rails code](#):

```

# Runs all the validations within the specified context. Returns +true+ if
# no errors are found, +false+ otherwise.
#
# If the argument is +false+ (default is +nil+), the context is set to <tt>:create</tt>
# <tt>new_record?</tt> is +true+, and to <tt>:update</tt> if it is not.
#
# Validations with no <tt>:on</tt> option will run no matter the context. Validation
# some <tt>:on</tt> option will only run in the specified context.
def valid?(context = nil)
  context ||= (new_record? ? :create : :update)
  output = super(context)

```

```
errors.empty? && output
end
```

The trick with `on: :create` and `on: :update` works because Rails by default does the job of providing the most suitable context. But that does not mean you are only limited in your code to those two cases which work out of box.

We could go with manual check for both contexts in our controllers but we would have to take database transaction into consideration, if our validations are doing SQL queries.

```
class Admin::UsersController
  def edit
    User.transaction do
      @user = User.find(params[:id])
      if @user.valid?(:admin) && @user.valid?(:create)
        @user.save!(validate: false)
        redirect # ...
      else
        render # ...
      end
    end
  end
end
```

I doubt that the end result is 100% awesome.

When it might come useful

I once used this technique to introduce new context `on: :destroy` which was doing something similar to:

```
class User < ActiveRecord::Base
  has_many :invoices
  validate :does_not_have_any_invoice, on: :destroy

  def destroy
    transaction do
      valid?(:destroy) or raise RecordInvalid.new(self)
      super()
    end
  end
end
```

```
private

def does_not_have_any_invoice
  errors.add(:invoices, :present) if invoices.exists?
end
end
```

The idea was, that it should not be possible to delete user who already took part of some important business activity.

Nowdays we have `has_many(dependent: :restrict_with_exception)` but you might still find this technique beneficial in other cases where you would like to run some validations before destroying an object.

What next?

That was quick introduction to custom validation contexts in Rails. In the next episode we are going to talk about other, perhaps better, ways to solve our initial dilemma that started with validations being context dependent. Subscribe to our newsletter below if you don't want to miss it.

You might also want to read some of our other popular blogposts ActiveRecord-related:

- [3 ways to do eager loading \(preloading\) in Rails 3 & 4](#)
- [Single Table Inheritance - problems and solutions](#)

Update

The next part is out:

- [Mastering Rails Validations: Objectify](#)

Would you like to continue learning more?

If you enjoyed the article, [subscribe to our newsletter](#) so that you are always the first one to get the knowledge that you might find useful in your everyday Rails programmer job.

Content is mostly focused on (but not limited to) Ruby, Rails, Web-development and refactoring Rails applications.

Also, make sure to check out our latest book [Domain-Driven Rails](#). Especially if you work with big, complex Rails apps.