
Mutations



Compose your business logic into commands that sanitize and validate input. Write safe, reusable, and maintainable code for Ruby and Rails apps.

Installation

```
1 gem install mutations
```

Or add it to your Gemfile:

```
1 gem 'mutations'
```

Example

```
1 # Define a command that signs up a user.
2 class UserSignup < Mutations::Command
3
4   # These inputs are required
5   required do
6     string :email, matches: EMAIL_REGEX
7     string :name
8   end
9
10  # These inputs are optional
11  optional do
12    boolean :newsletter_subscribe
13  end
14
15  # The execute method is called only if the inputs validate. It does
16  # your business action.
17  def execute
18    user = User.create!(inputs)
19    NewsletterSubscriptions.create(email: email, user_id: user.id) if
20    newsletter_subscribe
21    UserMailer.async(:deliver_welcome, user.id)
22    user
23  end
24 end
25
26 # In a controller action (for instance), you can run it:
27 def create
28   outcome = UserSignup.run(params[:user])
29 end
```

```
28 # Then check to see if it worked:
29 if outcome.success?
30   render json: {message: "Great success, #{outcome.result.name}!"}
31 else
32   render json: outcome.errors.symbolic, status: 422
33 end
34 end
```

Some things to note about the example:

- We don't need `attr_accessible` or `strong_attributes` to protect against mass assignment attacks
- We're guaranteed that within `execute`, the inputs will be the correct data types, even if they needed some coercion (all strings are stripped by default, and strings like "1" / "0" are converted to true/false for `newsletter_subscribe`)
- We don't need ActiveRecord validations
- We don't need callbacks on our models – everything is in the `execute` method (helper methods are also encouraged)
- We don't use `accepts_nested_attributes_for`, even though multiple ActiveRecord models are created
- This code is completely re-usable in other contexts (need an API?)
- The inputs to this 'function' are documented by default – the bare minimum to use it (name and email) are documented, as are 'extras' (`newsletter_subscribe`)

Why is it called 'mutations'?

Imagine you had a folder in your Rails project:

```
1 app/mutations
```

And inside, you had a library of business operations that you can do against your datastore:

```
1 app/mutations/users/signup.rb
2 app/mutations/users/login.rb
3 app/mutations/users/update_profile.rb
4 app/mutations/users/change_password.rb
5 ...
6 app/mutations/articles/create.rb
7 app/mutations/articles/update.rb
8 app/mutations/articles/publish.rb
9 app/mutations/articles/comment.rb
10 ...
11 app/mutations/ideas/upsert.rb
12 ...
```

Each of these *mutations* takes your application from one state to the next.

That being said, you can create commands for things that don't mutate your database.

How do I call mutations?

You have two choices. Given a mutation `UserSignup`, you can do this:

```
1 outcome = UserSignup.run(params)
2 if outcome.success?
3   user = outcome.result
4 else
5   render outcome.errors
6 end
```

Or, you can do this:

```
1 user = UserSignup.run!(params) # returns the result of execute, or
    raises Mutations::ValidationException
```

What can I pass to mutations?

Mutations only accept hashes as arguments to `#run` and `#run!`

That being said, you can pass multiple hashes to `run`, and they are merged together. Later hashes take precedence. This give you safety in situations where you want to pass unsafe user inputs and safe server inputs into a single mutation. For instance:

```
1 # A user comments on an article
2 class CreateComment < Mutations::Command
3   required do
4     model :user
5     model :article
6     string :comment, max_length: 500
7   end
8
9   def execute; ...; end
10 end
11
12 def somewhere
13   outcome = CreateComment.run(params[:comment],
14     user: current_user,
15     article: Article.find(params[:article_id])
16   )
17 end
```

Here, we pass two hashes to `CreateComment`. Even if the `params[:comment]` hash has a `user` or `article` field, they're overwritten by the second hash. (Also note: even if they weren't, they couldn't be of the

correct data type in this particular case.)

How do I define mutations?

1. Subclass Mutations::Command

```
1 class YourMutation < Mutations::Command
2   # ...
3 end
```

2. Define your required inputs and their validations:

```
1 required do
2   string :name, max_length: 10
3   symbol :state, in: %i(AL AK AR ... WY)
4   integer :age
5   boolean :is_special, default: true
6   model :account
7 end
```

3. Define your optional inputs and their validations:

```
1 optional do
2   array :tags, class: String
3   hash :prefs do
4     boolean :smoking
5     boolean :view
6   end
7 end
```

4. Define your execute method. It can return a value:

```
1 def execute
2   record = do_thing(inputs)
3   # ...
4   record
5 end
```

See a full list of options [here](#).

How do I write an execute method?

Your execute method has access to the inputs passed into it:

```
1 self.inputs # white-listed hash of all inputs passed to run. Hash has
               indifferent access.
```

If you define an input called *email*, then you'll have these three methods:

```
1 self.email          # Email value passed in
2 self.email=(val)     # You can set the email value in execute. Rare,
                       # but useful at times.
3 self.email_present? # Was an email value passed in? Useful for
                       # optional inputs.
```

You can do extra validation inside of execute:

```
1 if email =~ /aol.com/
2   add_error(:email, :old_school, "Wow, you still use AOL?")
3   return
4 end
```

You can return a value as the result of the command:

```
1 def execute
2   # ...
3   "WIN!"
4 end
5
6 # Get result:
7 outcome = YourMutation.run(...)
8 outcome.result # => "WIN!"
```

What about validation errors?

If things don't pan out, you'll get back an `Mutations::ErrorHash` object that maps invalid inputs to either symbols or messages. Example:

```
1 # Didn't pass required field 'email', and newsletter_subscribe is the
   # wrong format:
2 outcome = UserSignup.run(name: "Bob", newsletter_subscribe: "Wat")
3
4 unless outcome.success?
5   outcome.errors.symbolic # => {email: :required, newsletter_subscribe:
   # :boolean}
6   outcome.errors.message # => {email: "Email is required",
   # newsletter_subscribe: "Newsletter Subscription isn't a boolean"}
7   outcome.errors.message_list # => ["Email is required", "Newsletter
   # Subscription isn't a boolean"]
8 end
```

You can add errors in a `validate` method if the default validations are insufficient. Errors added by `validate` will prevent the `execute` method from running.

```
1 #...
```

```
2 def validate
3   if password != password_confirmation
4     add_error(:password_confirmation, :doesnt_match, "Your passwords
      dont't match")
5   end
6 end
7 # ...
8
9 # That error would show up in the errors hash:
10 outcome.errors.symbolic # => {password_confirmation: :doesnt_match}
11 outcome.errors.message # => {password_confirmation: "Your passwords don
    't match"}
```

Alternatively you can also add these validations in the execute method:

```
1 #...
2 def execute
3   if password != password_confirmation
4     add_error(:password_confirmation, :doesnt_match, "Your passwords
      dont't match")
5     return
6   end
7 end
8 # ...
9
10 # That error would show up in the errors hash:
11 outcome.errors.symbolic # => {password_confirmation: :doesnt_match}
12 outcome.errors.message # => {password_confirmation: "Your passwords don
    't match"}
```

If you want to tie the validation messages into your I18n system, you'll need to write a custom error message generator.

FAQs

Is this better than the 'Rails Way'?

Rails comes with an awesome default stack, and a lot of standard practices that folks use are very reasonable (eg, thin controllers, fat models).

That being said, there's a whole slew of patterns that are available to experienced developers. As your Rails app grows in size and complexity, my experience has been that some of these patterns can help your app immensely.

How do I share code between mutations?

Write some modules that you include into multiple mutations.

Can I subclass my mutations?

Yes, but I don't think it's a very good idea. Better to compose.

Can I use this with Rails forms helpers?

Somewhat. Any form can submit to your server, and mutations will happily accept that input. However, if there are errors, there's no built-in way to bake the errors into the HTML with Rails form tag helpers. Right now this is really designed to support a JSON API. You'd probably have to write an adapter of some kind.