Overview

- Why?
- Getting started
- Steps
  - What’s a step?
  - step
  - model
  - policy
  - params
  - transaction
  - try
  - lock
  - options
  - only_if
  - each
  - Steps arguments
- The context object
- Handling service results
- Testing
  - Available matchers
    - fail_a_policy(name)
    - fail_a_contract
    - fail_to_find_a_model(name)
    - fail_with_an_invalid_model(name)
    - fail_with_exception
    - fail_a_step(name)
    - run_successfully
- API
  - Steps

    - params(name = :default, default_values_from: nil, base_class: Service::ContractBase, from_resource:
    - options(&block)
    - model(name = :model, step_name = :"fetch_#{name}", optional: false)
    - policy(name = :default, class_name: nil)
    - transaction(&block)
    - try(*exceptions, &block)
    - lock(*keys, &block)
    - only_if(condition_name, &block)
    - each(collection_name, as: nil, persist: nil, &block)
    - step(name)
- Helper available inside a step
    - fail!(message)
- The context object
    - success?
    - failure?
    - fail!(context = {})
    - fail(context = {})
- Calling a service with a block
    - .call(context = {}, &actions)
    - on_success
    - on_failure
    - on_failed_step(name)
    - on_failed_policy(name = "default")
    - on_failed_contract(name = "default")
    - on_model_not_found(name = "model")
    - on_model_errors(name = "model")
    - on_exceptions(*exceptions)
    - on_lock_not_acquired(*keys)
- Contracts
    - Nested attributes
    - Reusing contracts
- Parameters
- Policy objects

    - Actions
- Best practices and guidelines
- Debugging
    - Steps Inspector
    - Live debugging

A service is a small object that encompasses business logic for a given action. From outside, it should be seen as a sort of black box. You provide it with parameters, it runs (including all the side effects it can trigger), and then it returns a result object describing the outcome of the action.

You can think of a service as a conductor of an orchestra, it organizes how things are done and not necessarily does them by itself.

A service object has a functional flow control using steps and while it holds no state between executions, during an execution all state between steps is maintained in its context object. Everything that happens for a business action will be done in the service, let it be by using specialized steps, custom methods or delegating things to other objects.

Our service concepts are heavily inspired by Trailblazer 2.0, dry-transaction and to a lesser extent interactor.

Why?

The most common place where to use a service object is in a controller action. A controller action is usually a point of entry for dedicated business logic, but it's not always clear where to put that logic. Of course, it can stay in the controller action, that's what a lot of Rails apps do, but then it tends to grow and repeat the same processes over and over. Sometimes part of that logic can be extracted into models, and that's fine, but it can lead to what's called fat models where a model starts handling way more things than it should.

That's a typical case where a service object comes in handy: you describe what the business logic is step by step (fetching models, checking permissions, and so on) and then you can take action depending on the outcome in a very simple way. All the error handling is done for you inside the service, and you can match its outcome in a very deterministic way. Input parameters are validated using a dedicated object called a contract and when a step fails, the service stops. That means that if a step runs, every previous step was successful.

It doesn't mean every situation should be handled by a service. A service encapsulates business logic for a given action, and is encouraged to rely on specialized objects. So it won't replace models or libraries, for example. As said before, it really shines when used in a controller action, but it can be used anywhere. A background job is another good example.

Another benefit immediately available with services, is that since all the logic related to a business action is encapsulated in it, you can call it from anywhere (console, SDK, controllers, specs, etc.) and you'll always get the same outcome.

It's also quite easy to understand what's happening in a service at a glance, since all the steps are listed sequentially. There's also a matching system to handle the possible outcomes in a deterministic way, here again you can understand at a glance what a controller will do, for example.

Getting started

Here's a simplified service to update a user's username which demonstrates most available steps:

```
class User::UpdateUsername
  include Service::Base

  params do
    attribute :id, :integer
    attribute :username, :string

    validates :id, presence: true
    validates :username, presence: true, format: { with: /\A[a-zA-Z0-9]+\z/ }
  end

  model :user
  policy :can_update_username

  transaction do
    step :update
    step :log
  end

  private

  def fetch_user(params:)
    User.find_by(id: params.id)
  end

  def can_update_username(guardian:, user:)
```

```
    guardian.can_edit_username?(user)
  end

  def update(params:, user:)
    user.update!(username: params.username)
  end

  def log(guardian:, user:)
    StaffActionLogger.new(guardian.user).log_username_change(
      user,
      user.username_before_last_save,
      user.username,
    )
  end
end
```

And here is an example on how you could call such a service:

```
User::UpdateUsername.call(params: {id: 1, username: "username"}, guardian:
Discourse.system_user.guardian)
```

Without knowing how services work, you can probably guess what's happening here. Let's dive in.

Steps

What's a step?

This is the basic unit of a service. There is a generic one (`step`) and specialized ones (`params`, `model`, etc.), and they're all steps.

Steps are defined *in the order they will be called*. Each step will call a corresponding method and, depending on its return value, will continue or halt the execution of the service. Most steps rely on returning a value, and *not raising an exception* (otherwise, you'll break the execution flow).

The immediate benefit is that error handling is done for you, and you don't have to implement any specific logic in the service itself. We'll see later how to handle errors.

> ⚠ As said above, a step shouldn't raise an exception, as this will not be handled automatically for you by the service. If a service raises an exception, it should be treated as a bug. If you need to call a piece of code that might raise an exception under expected usage, then you should use the `try` step to wrap the steps that could raise.

Let's see what steps are available and how to use them.

`step`

This is the generic step, you provide a name, and it will run the defined method of the same name. The return value of this step doesn't impact the execution flow. To mark the service as failed in a generic step, you have to call `#fail!` explicitly.

`model`

This specialized step helps to remove some boilerplate when dealing with models. By default, it will execute the method named `fetch_<name>`. In the above example, you can see we name our model `:user` and the corresponding method is named `fetch_user`.

Here, you can fetch (or instantiate) a model as you see fit. If the step returns a falsy value, or an exception is raised, then the execution flow will stop here. If an `ActiveRecord` model with unsaved changes is returned, the step also fails when the model contains errors or its validations fail. If not, the execution flow will stop.

This step is also compatible with collections: if a collection is fetched but empty, the execution flow will stop.

You can also provide an `ActiveRecord` relation. The step won't load records but will determine if the relation will return any records. If there are none, the execution flow will stop.

Sometimes, you need to fetch a model (or a collection of models), but it's ok if it's empty. For those cases, you can use the `optional: true` option allowing the execution flow to continue even if the model returns a falsy value.

`policy`

This step will execute the method of the same name. You can put arbitrary code here, and the execution flow will stop if the return value is falsy.

Usually, a policy is related to some state on one of the service models and/or to the current user (if any).

`params`

This is one of the most powerful steps. Its main purpose is to validate the incoming data before feeding it to the models and to the service at a more global level. This is actually `ActiveModel` validations but applied to the incoming parameters.

This step will run coercions and validations defined in the provided block. If the underlying contract isn't valid (at least one validation failed), then the execution flow will stop.

`transaction`

This step is a bit special, as it will wrap any other steps defined in its block inside a SQL transaction.

Any step that fails inside the transaction block will cause a rollback of the transaction.

`try`

This step will catch exceptions raised by the steps defined in its block. Specific exception classes can be provided if you don't want to automatically catch all exceptions.

`lock`

This step uses `DistributedMutex` under the hood to create a lock around the steps defined in its block. This is useful when race conditions can happen.

`options`

This step is another special one, as it's similar to a contract (without the validations part), but for options your service can take. This is useful if you need to change your service behavior depending on certain conditions. Also, that step can't fail.

`only_if`

This step will only run the steps defined in its block if the provided condition is true. That's another step that can't fail by itself.

`each`

This step iterates over a collection from the context and runs the steps defined in its block for each item. The item is provided under its singularized name (e.g., `each :users` provides `user:`) along with `index:`. If any nested step fails, iteration stops immediately and the service fails with the failing item and index available in context.

An empty collection is silently skipped.

A `persist:` option allows values to accumulate across iterations and survive the loop's variable isolation. This is useful for collecting results during bulk operations.

Steps arguments

Each step is called with the service context. To access a value in it, just provide its key as a keyword argument.

The context object

The only state a service maintains is its context object. This is where each step can put data to be used by other steps. Most of the time, you don't need to access the context directly, as specialized steps (such as `params`, `model` or `options`) will store the proper data for you. But sometimes it's necessary (even if uncommon). In those cases, it's just a matter of using the context like you would with a hash:

```
def first_step
  context[:my_special_key] = "My special value"
end


…


# Then in a later step, you can use it like any other key from the context
def another_step(my_special_key:)
  # do something with `my_special_key`
end
```

Handling service results

Once a service has been called, it will return a result object, to know whether the call was a success. In the case of a failure, the result object can be inspected to know what failed and why.

Each step will store its outcome in the result object, accessible through special keys (like `result.model.user` for example). While this is nice, this would be tedious to manually check the result object. That's why there's a built-in feature allowing to run the service, match step outcomes and act upon results using a custom DSL.

This feature makes the use of a service a breeze. Continuing with our `User::UpdateUsername` service, this is how we could use it inside a controller (but it can work anywhere, not just in controllers):

```
def update
  User::UpdateUsername.call(service_params) do |result|
    on_success { |user:| render(json: success_json.merge(new_username: user.username)) }
    on_failure { render(json: failed_json, status: 422) }
    on_failed_contract { |contract| render(json: failed_json.merge(errors:
contract.errors.full_messages), status: 400) }
    on_model_not_found(:user) { raise Discourse::NotFound }
    on_failed_policy(:can_update_username) { raise Discourse::InvalidAccess.new }
  end
end
```

Passing a block to `.call` allows to "match" an outcome, a bit like if we were using pattern matching. There are two generic matchers (`on_success` and `on_failure`) and each specialized step has at least one dedicated matcher. The complete detailed list is available in the API section.

> `on_failure` is like a catch-all rule, it will match only if the service fails *and* no other more specialized matcher matches.

This declarative way helps decoupling what is handled by the caller (here a controller) from what is handled by the service.

Testing

To simplify testing, custom RSpec matchers have been added. It's also considered a best practice to always follow the same structure. Remember to test the caller class too. If your service is called from a controller, for example, that controller should be tested with a request spec. Following the various outcome blocks will help to know what to test. Here is how we could test our `User::UpdateUsername` service:

```
RSpec.describe User::UpdateUsername do
  describe described_class::Contract, type: :model do
```

```
    it { is_expected.to validate_presence_of(:id) }
    it { is_expected.to validate_presence_of(:username) }
    it do
      is_expected.to allow_values("0userName", "USERNAME", "username",
"21421341").for(:username)
    end
    it { is_expected.not_to allow_values("invalid-username").for(:username) }
  end

  describe ".call" do
    subject(:result) { described_class.call(params:, **dependencies) }

    fab!(:user)
    let(:params) { { username:, id: user_id } }
    let(:dependencies) { { guardian: } }
    let(:guardian) { user.guardian }
    let(:username) { "NewUsername" }
    let(:user_id) { user.id }

    context "when contract isn’t valid" do
      let(:username) { "----" }

      it { is_expected.to fail_a_contract }
    end

    context "when model is not found" do
      let(:user_id) { 0 }

      it { is_expected.to fail_to_find_a_model(:user) }
    end

    context "when current user cannot update user's username" do
      let(:guardian) { Guardian.new }

      it { is_expected.to fail_a_policy(:can_update_username) }
    end

    context "when everything’s ok" do
      it { is_expected.to run_successfully }
```

```
      it "updates user's username" do
        expect { result }.to change { user.reload.username }.to(username)
      end

      it "logs the action" do
        expect { result }.to change { UserHistory.count }.by(1)
      end
    end
  end
end
```

First, and because a contract is present, we test it using a dedicated `describe` block. Since a contract uses `ActiveModel` under the hood, the simplest way to test it is to use Shoulda Matchers.

Then, we use a `describe` block for the `.call` method, which is how the service is run. We're using a `context` for each possible branching. It's quite easy as we just have to follow the steps we defined in the service. You can see we're not testing all the possible values to have the contract fail: that's because it's tested extensively above, so here we're just ensuring the `params` step is properly called and if a bad value is provided, then it will stop the execution of the service. For the other steps, if they can fail, then they should have a context using a dedicated matcher.

The `run_successfully` matcher ensures the service succeeded (`result.success?` is `true`) and will provide some debugging information if that's not the case.

In the event of a matcher failing, it will output details about the result object to help debugging things:

```
Failures:

  1) User::UpdateUsername.call when current user cannot update user's username is expected to
fail a policy named 'can_update_username'
     Failure/Error: it { is_expected.to fail_a_policy(:can_update_username) }

       Expected policy 'can_update_username' (key: 'result.policy.can_update_username') to
fail but it succeeded.

       Inspecting User::UpdateUsername result object:

       [1/6] [params] default (0.1152 ms) □
       [2/6] [model] user (1.7147 ms) □
       [3/6] [policy] can_update_username (0.0093 ms) □ ⚠□  <= expected to return false but
```

```
got true instead
       [4/6] [transaction] (2.5952 ms)
       [5/6]   [step] update (2.4038 ms) □
       [6/6]   [step] log (0.0054 ms) □
     # ./spec/services/update_username_spec.rb:39:in `block (6 levels) in <main>'
     # ./spec/rails_helper.rb:497:in `block (2 levels) in <top (required)>'
     # /home/discourse/.bundle/gems/ruby/3.3.0/gems/webmock-3.23.1/lib/webmock/rspec.rb:39:in
`block (2 levels) in <top (required)>'
```

Available matchers

`fail_a_policy(name)`

This matcher expects a policy named `name` to fail.

`fail_a_contract`

This matcher expects the service contract to be invalid.

`fail_to_find_a_model(name)`

This matcher expects a model step named `name` to not find its model.

`fail_with_an_invalid_model(name)`

This matcher expects a model step named `name` to find its model, but that model should be invalid.

`fail_with_exception`

This matcher expects the `try` step to have caught an exception. A specific exception class can be provided.

`fail_a_step(name)`

This matcher expects a step named `name` to fail.

`run_successfully`

This matcher expects the service to succeed.

API

Steps

`params(name = :default, default_values_from: nil, base_class: Service::ContractBase, from_resource: nil, only: nil, except: nil, &block)`

Arguments

- *name*: the name of the contract, in the case there is more than one. Defaults to `default`.
- *default_values_from*: name of a model to use to pre-fill the contract values. This is useful when you want some values of a model to be updated through a contract while applying other default values. A real-world example is available in the `Chat::UpdateChannel` service.
- *base_class*: the base class to use for the contract. This is particularly useful for reusing an existing contract (for example, you could have very similar contracts between a `Create` and a `Update` service). Defaults to `Service::ContractBase`.
- *from_resource*: a resource whose writable attribute names and ActiveModel types are imported into the contract.
- *only*: writable attribute names to import. Accepts strings or symbols. An empty array imports no attributes.
- *except*: writable attribute names to exclude from the import. Accepts strings or symbols.
- *block*: the block containing all the validations, attribute definitions, etc.

This step declares the use of a contract to validate input parameters. Parameters provided to the service will be passed to the contract if their name matches the attributes defined in the contract.

With `from_resource`, all writable attributes are imported by default. Use `only:` or `except:` to select the attributes for an action:

```
params(from_resource: TopicResource, only: %i[title]) do
  attribute :notify, :boolean, default: false
  validates :title, presence: true
end
```

Imports run before the block. Explicit attribute declarations override imported types through normal ActiveModel behavior. A declaration that omits the type preserves the imported type. Selection applies only to imports, so the block can declare additional parameters or declare an excluded attribute explicitly.

Selection options require `from_resource:` and cannot be combined. Unknown or read-only selection names raise an error when the service is defined. Imports include only names and types.

With `default_values_from`, nested hashes are merged recursively. Supplied values override model values, including `nil`, `false`, and `0`. An empty hash keeps the defaults at that level. A supplied array replaces the whole model array; an empty array clears it.

For example, model values `{ dimensions: { width: 640, height: 480 }, tags: ["draft"] }` combined with params `{ dimensions: { width: 800 }, tags: [] }` produce `{ dimensions: { width: 800, height: 480 }, tags: [] }` before contract validation.

String and symbol keys are treated alike at every level. Combining defaults and inputs does not modify their hashes or arrays. Coercion and validation still run on the resulting values.

Under the hood, a class for the contract will be automatically created, allowing easy testing. The default contract will result in `Contract`, otherwise it will prepend the name used for the contract (for `params(:user_avatar)`, this will give `UserAvatarContract`).

If the contract is invalid, it will stop the execution of the service. Its result object can be inspected by accessing the `result.contract.<name>` key of the main result object. The contract result object exposes two keys:

- *errors*: the errors returned by the contract.
- *parameters*: the raw parameters provided to the contract before any coercion happens.

`options(&block)`

Arguments

- *block*: the block containing the option definitions.

This steps is used to define options that can be provided to the service to change its behavior. The syntax to define an option is the same as the one used for contracts, but it has no validations. A good example can be found in the Chat::CreateMessage service.

This step cannot fail.

`model(name = :model, step_name = :"fetch_#{name}", optional: false)`

Arguments

- *name*: the name of the model. Defaults to `model`.

- *step_name*: the name of the method to call for this step. For example, when instantiating a new model, we could use `instantiate_model`. Defaults to `fetch_<model_name>`.
- *optional*: if the model is marked as optional, the step won't fail if the model isn't found. Defaults to `false`.

This step helps to remove some boilerplate when fetching/instantiating models or a collection of models. A model can be pretty much anything (not only `ActiveRecord` models), being a single object or a collection. The result of the step will be stored in the context as `name` (so, by default, it would be `context[:model]`).

The step will fail if the model is `nil` or `empty?`, or if an exception is raised. In the case of an `ActiveRecord` object with unsaved changes, the step also fails if the model already contains errors (a record whose save failed, for example) — those errors are preserved for `on_model_errors` — or otherwise if running its validations marks it as invalid. Models without pending changes are not revalidated, and non-`ActiveRecord` objects are never validated.

Its result object can be inspected by accessing the `result.model.<name>` key of the main result object. The model result object exposes one or two keys:

- *invalid*: will be `true` if the model has been found but is invalid.
- *not_found*: will be `true` if the model was not found.
- *exception*: the exception that caused the step to fail.

`policy(name = :default, class_name: nil)`

Arguments

- *name*: the name of the policy. Defaults to `default`.
- *class_name*: a policy class to implement the logic instead of defining the step in the service. Defaults to `nil`.

This step declares the use of a policy. A policy is just arbitrary code, and the step will fail if the policy result is falsy. If you have a rather complex policy, it's better to use a policy class. It needs to inherit from `Service::PolicyBase` and implement `#call` and `#reason` as using a policy class allows explaining in more details why the policy failed through the use of the `#reason` method. A complete example can be found in the `Chat::DirectMessageChannel::Policy::MaxUsersExcess` class used by the `Chat::CreateDirectMessageChannel` service.

The step will fail if the policy returns a falsy value. Its result object can be inspected by accessing the `result.policy.<name>` key of the main result object. The policy result object exposes one key:

- *reason*: the reason why the policy failed if a policy class was used.

`transaction(&block)`

This step is a bit special as its only purpose is to wrap other steps inside a SQL transaction. It cannot fail by itself but if a step fails inside the transaction block, the transaction will rollback.

`try(*exceptions, &block)`

Arguments

- *exceptions*: one or more exception classes to catch. Not providing any class is equivalent to provide `StandardError`.
- *block*: a block containing other steps.

This step wraps other steps. If any of the wrapped steps raises an exception, the `try` step will catch it and fail, which will halt the execution flow.

`lock(*keys, &block)`

Arguments

- *keys*: one or more keys to provide the lock with an unique ID.
- *block*: a block containing other steps.

This step wraps other steps. If the lock can't be acquired for any reason, the execution flow will be halted.

The `keys` provided are first matched against the `params` object of the service and, if not found, against the service context. This allows locking on values derived from earlier steps, such as a model fetched from the database. When a resolved value responds to `id` (e.g. an `ActiveRecord` model), its `id` is used automatically to build the lock name.

For example, if the provided key is `user_id` and its value is `1`, then the resulting lock name will be `user_id:1`, so if the same service is called with `2` as the `user_id` value, it will not wait on the first lock. Similarly, `lock(:topic)` after a `model :topic` step will use the topic's `id` in the lock name.

`only_if(condition_name, &block)`

Arguments

- *condition_name*: the name of the condition to check.
- *block*: a block containing other steps.

This step wraps other steps. If the condition is not met (meaning the related method returns a falsy value), the steps inside its block will be skipped but the execution flow won’t be halted.

That step is very useful when you have several steps depending on the same condition to be run, as it allows to keep all the error handling provided by steps like `model`, `policy`, etc.

`each(collection_name, as: nil, persist: nil, &block)`

Arguments

- *collection_name*: the name of the context key holding the collection to iterate over.
- *as*: override the singularized item name. For example, `each :users, as: :member` provides `member:` instead of `user:`.
- *persist*: keys that should accumulate across iterations and survive the loop's variable isolation. Accepts a hash where values are lambdas or method symbols used to initialize the key, or an array of symbols for keys without initialization.
- *block*: a block containing other steps.

This step iterates over a collection and runs the steps defined in its block for each item. Each iteration receives the singularized item name (e.g., `user:` for `each :users`) and `index:` as keyword arguments.

By default, variables set inside the loop are isolated: they don't leak to the outer context. The `persist:` option allows specific keys to survive the isolation. This is particularly useful for accumulating results during bulk operations.

If any nested step fails (a policy returns falsy, a model is not found, or `fail!` is called), iteration stops immediately and the service fails. The failing item and index remain available in context for error reporting, and existing matchers (`on_failed_policy`, `on_model_not_found`, etc.) work as expected.

An empty collection is silently skipped.

```
# Fail-fast iteration
each :users do
  policy :can_delete
  step :destroy
end

# Accumulating results across iterations
each :tag_names, persist: { results: -> { { created: [], failed: [] } } } do
  step :create_tag
end
```

```
# Custom item name
each :users, as: :target_user do
  step :process
end
```

`step(name)`

Arguments

- *name*: the name of the step.

This is a generic step, to execute arbitrary code. A generic step won't ever fail by itself, no matter what its return value is. If you need to mark a step as failed, you should use the `#fail!` method.

A generic step has a result object, even if by default it exposes nothing. It can be accessed with the `result.step.<name>` key of the main result object.

Helper available inside a step

`fail!(message)`

Arguments

- *message*: the error message to set on the result object.

This method can be used to mark a generic step as failed. The result object is accessible at `result.step.<step_name>` and exposes an `error` key.

The context object

This context object is available inside a step as `context` or as the return value of a service.

`success?`

Returns `true` if the context is set as successful (this is the default).

`failure?`

Returns `true` if the context is set as failed.

`fail!(context = {})`

Arguments

- *context*: the context to merge into the current one.

Marks the context as failed and raises a `Service::Base::Failure` exception.

`fail(context = {})`

Arguments

- *context*: the context to merge into the current one.

Marks the context as failed without raising an exception.

Calling a service with a block

The block form of `.call` can be used anywhere (a controller action, a job, another class, etc.). The provided actions will be evaluated in the order they appear, and the execution will stop at the first responder. The only exception to this is `on_failure` as it will always be executed last.

`.call(context = {}, &actions)`

Arguments

- *context*: the initial context to provide to the service. If the service is called from a controller, you can use the `service_params` helper which will return `params` and the `guardian` object.
- *actions*: the block containing the steps to match on.

Example

```
MyService.call(**service_params, extra_dependency: my_dependency) do |result|
  on_success { |my_model:| do_something(my_model) }
  on_failure { handle_generic_failure }
end
```

If you need to access the result object, it's available as the first object passed to the main block (see example above). Each outcome block can match keys from the context (exactly as you do when writing step definitions) independently from what object is passed to the block. For example, it means that with the `on_failed_contract` matcher, you could access a previously fetched model while using the provided contract as the first argument. It would be done like this:

```
on_failed_contract { |contract, my_model:| do_something(contract, my_model) }
```

You could do all this by only using the result object, but it's a bit nicer this way (and will ensure the key you're trying to access actually exists).

A required keyword argument (`my_model:`) will raise if the service didn't set that key, which is what you want most of the time. For a key that's only set on some paths (a model fetched by a step that didn't run, or steps wrapped in an `only_if` block), give the argument a default value and it will be used as a fallback:

```
on_failure { |my_model: nil| do_something(my_model) }
```

`on_success`

Will execute the provided block if the service succeeds.

`on_failure`

Will execute the provided block if the service fails.

`on_failed_step(name)`

Arguments

- *name*: the name of the step to match.

Will execute the provided block if the step named `name` fails. It also provides the step result object as the first argument of the block.

`on_failed_policy(name = "default")`

Arguments

- *name*: the name of the policy to match. Defaults to `default`.

Will execute the provided block if the policy named `name` fails. It also provides the policy result object as the first argument of the block.

`on_failed_contract(name = "default")`

Arguments

- *name*: the name of the contract to match. Defaults to `default`.

Will execute the provided block if the contract named `name` is invalid. It also provides the contract result object as the first argument of the block.

`on_model_not_found(name = "model")`

Arguments

- *name*: the name of the model to match. Defaults to `model`.

Will execute the provided block if the model named `name` is not present. It also provides the model result object as the first argument of the block.

`on_model_errors(name = "model")`

Arguments

- *name*: the name of the model to match. Defaults to `model`.

Will execute the provided block if the model named `name` contains errors. It also provides the actual model as the first argument of the block.

`on_exceptions(*exceptions)`

Arguments

- *exceptions*: zero or more exception classes that can be caught by a `try` step.

Will execute the provided block if a `try` step failed by catching one of the provided exception classes. If no class is provided, then the block will be executed if a `try` step caught any exception. It also provides the actual exception as the first argument of the block.

`on_lock_not_acquired(*keys)`

Arguments

- *keys*: the keys used by the lock to match.

Will execute the provided block if the `lock` step failed.

Contracts

The main purpose of a contract is to validate the incoming data before feeding it to the models and to the service at a more global level. The important part being validating **user input** (typically coming from `params` in a controller, and services expect to access those parameters through the `params` key of their context).

A contract is actually an `ActiveModel` object, so all the API of the latter is available. Anyway, let's see how to define and use a contract inside a service.

To define a service contract, just call `params` and open a block:

```
params do
  attribute :id, :integer
  attribute :username, :string

  validates :id, presence: true
  validates :username, presence: true, format: { with: /\A[a-zA-Z0-9]+\z/ }
end
```

Here, all the API from `ActiveModel` is available. In this example, we define we want to validate two attributes, `id` and `username` with their respective cast type (`integer` and `string`).

> Use cast types extensively as they'll provide you with proper objects before any validation happens.
>
> Rails ships with cast types for `big_integer`, `binary`, `boolean`, `date`, `datetime`, `decimal`, `float`, `immutable_string`, `integer`, `string` and `time`. Custom cast types can be defined, we ship two: `array` and `symbol`.

> Don't define attributes if you don't transform them or validate them. The primary purpose of a contract is to validate data, it can also be used to cast or

> massage data before using it (usually a contract does both).

Then, we define validations, exactly like you would in an `ActiveRecord` model. Here, we're checking for `id` and `username` not being blank and that `username` respects an expected format.

Another thing that is available in a contract, since it's an `ActiveModel` object, are validation callbacks. If you need to manipulate the attribute values, you can do so by calling `before_validation` or `after_validation`. There are examples in the codebase, like in the `Chat::CreateCategoryChannel` service.

> ⚠ Once run by the service, a contract is frozen and you can't modify its attributes. If you need to do some processing on its values, you can do it directly inside the contract itself.

Some methods have been added to the contract object to make your life a bit easier when dealing with model updates and things like that:

- `#slice` and `#merge` are available.
- `#to_hash` has been implemented, so the contract object will be automatically cast as a hash by Ruby depending on the context. For example, with an ActiveRecord model, you can do this: `user.update(**params)`.

Nested attributes

It's fairly common to get nested parameters from an endpoint. This usually helps keeping things organized. Contracts support nested hashes and nested arrays. Let's imagine we have a service that needs to respond to that kind of input:

```
{
  "user": {
    "id": 2,
    "username": "new username"
  },
  "page": {
    "sort": "asc",
    "number": 3
  }
}
```

We can easily define a contract to validate such input:

```
params do
  attribute :user, :hash do
    attribute :id, :integer
    attribute :username, :string

    validates :id, :username, presence: true
  end

  attribute :page, :hash, default: -> { {} } do
    attribute :sort, :symbol, default: :asc
    attribute :number, :integer, default: 1

    validates :sort, inclusion: { in: %i[asc desc] }
    validates :number, numericality: { only_integer: true, greater_than: 0 }
  end

  validates :user, presence: true
end
```

For each nested attribute we want to validate, we just have to open a block and inside that block, we can define attributes and validations as usual. Inside that block, we have access to another contract object, so everything we can do in the parent contract is available in the nested one.

There are some things to notice:

- Here, before opening the block, we're using `:hash` to tell our contract to expect another hash. If we had an array of hashes, we would have used `:array` instead.
- A nested attribute is just an attribute, so if you want to validate its presence, you have to add a validation for it (as done for the `user` attribute). Otherwise, it will be considered as an optional attribute.
- When using default values, like we do for the `page` attribute, make sure to provide a default value for the whole attribute (here, an empty hash) to avoid `nil` errors when accessing nested attributes.

After validation, errors will be defined directly on the main contract, as usual. We're using the same approach Rails does for nested attributes in models. For example, if the `username` attribute is blank, the error will be available at `errors[:"user.username"]`. The full message will be `User username can't be blank`.

Now, let's imagine our input has arrays of hashes, like this:

```
{
  "tags": [{ "name": "tag1" }, { "name": "tag2" }]
```

```
}
```

A contract to validate that input could look like this:

```
params do
  attribute :tags, :array do
    attribute :name, :string

    validates :name, presence: true
  end

  validates :tags, presence: true
end
```

If there are errors, it will use the index of the array to define them. For example, if the second tag has no name, the error will be available at `errors[:"tags[1].name"]`. The full message will be `Tags[1] name can't be blank`.

And of course, you can combine nested hashes and arrays as you see fit.

Reusing contracts

Sometimes, you may want to reuse a contract between different services. For example, you could have a `Create` and an `Update` service for the same concept, and both would share most of their contract logic. First, let's create a (naive) `User::Create` service:

```
class User::Create
  include Service::Base

  params do
    attribute :username, :string
    attribute :email, :string
    attribute :date_of_birth, :date

    validates :username, presence: true, format: { with: /\A[a-zA-Z0-9]+\z/ }
    validates :email, :date_of_birth, presence: true
  end

  policy :can_create_user
  model :user, :create_user
```

```
  …
end
```

The important part here is the contract checking for a username, an email and a date of birth to allow creating a user. Now, let’s say we have another service to update a user, and you can update the same attributes. If we were to write a dedicated contract, that could be a bit tedious, as we’d want to be able to only update the username without having to update the email, for example. We can combine two options of the `params` step to help us, `default_values_from` and `base_class`:

```
class User::Update
  include Service::Base

  model :user
  params default_values_from: :user, base_class: User::Create::Contract

  private

  def fetch_user(params:)
    User.find_by(id: params.id)
  end
end
```

First, we fetch the user model to update. If `params.id` wasn’t provided or is invalid, the service will stop here. Then, we define the `params` step. Here, we tell the service to use the `User::Create::Contract` as the base class for the contract. This means all the attributes and validations defined in that contract will be available in this service too. Using the `default_values_from: :user` option tells the service to pre-fill the contract attributes with the values from the fetched user model. This way, if only the `username` attribute is provided, the `email` and `date_of_birth` attributes will be pre-filled with the current values from the user, thus passing the presence validations.

Parameters

Parameters provided to the service through the `params` key are accessible like `params.my_param` (not `params[:my_param]`). Parameters are available even if they're not processed by a contract.

Policy objects

When a policy starts becoming complex or when you'd like to provide more context on why it can fail, then it's time to use a policy object instead of a simple policy.

It's quite easy to create a new policy object, let's take the `can_update_username` policy we have in the `User::UpdateUsername` service and convert it:

```
class User::Policy::CanUpdateUsername < Service::PolicyBase
  delegate :user, to: :context, private: true

  def call
    guardian.can_edit_username?(user)
  end

  def reason
    # Here we can put more complex logic to dynamically output a reason, this is just an
example
    I18n.t("cannot_edit_username", username: user.username)
  end
end
```

There are some rules to keep in mind when writing a policy object:

- It must inherit from `Service::PolicyBase`.
- It must define two methods: `#call` and `#reason`.
- The context object is automatically injected in the policy, and is available by calling `#context` (like in a service).
- The guardian object is also automatically available as `#guardian`.
- By convention, it should be namespaced under its concept followed by the `Policy` namespace: for our current example, it means `User::Policy::` which maps to `app/services/user/policy/` on the filesystem.

> To keep things short and clear, feel free to use `delegate` extensively.

Then, when you want to use it in a service, just write your step like this:

```
policy :can_update_username, class_name: User::Policy::CanUpdateUsername
```

Actions

When a step starts becoming too complex, like it has too many branching statements for example, then it's time to extract all that logic to a dedicated class. That logic could live in a model for instance, but when in doubt, just create a new action.

An action is just a small class that responds to a `.call` method by convention. What happens inside is up to you. The idea, however, is to execute an action (hence the name) with minimal overhead. It means an action should not validate data, for example. It should be called with valid objects only, thus being able to work with them right away. It also means that an action should not fail. You can think of an action as a bare-bones service without all the bells and whistles. Also, an action can be reused by different services.

Here again, it's relatively simple to create a new action. Let's take as an example our `log` step:

```
class User::Action::LogUsernameChange < Service::ActionBase
  option :actor
  option :user

  def call
    StaffActionLogger.new(actor).log_username_change(
      user,
      user.username_before_last_save,
      user.username,
    )
  end
end
```

Of course, this is a very basic example, you can do more complex things in an action. A real-world example can be found in `User::Action::TriggerPostAction`. `Service::ActionBase` comes with `Dry::Initializer` which provides a nice mini-DSL:

- Use `option :my_arg` to declare a required keyword argument named `my_arg`.
- Use `optional: true` to declare the argument optional (for instance, `option :my_arg, optional: true`).

You should not need anything more than that to work with an action, but if you want to use some advanced features of `dry-initializer` (like coercion), just take a look at their docs.

Some rules to keep in mind when writing an action:

- It must inherit from `Service::ActionBase`.
- It must define one method: `#call`.
- You should prefer `option` over `param` to define arguments, as it's a bit more self-documenting on the caller side. It also allows you to use the hash shorthand syntax.

- By convention, it should be namespaced under its concept followed by the `Action` namespace: for our current example, it means `User::Action::` which maps to `app/services/user/action/` on the filesystem.

> 💡 To keep things short and clear, feel free to use `delegate` extensively.

Then, when you want to use it in a service, just write your step like this:

```
def log(guardian:, user:)
  User::Action::LogUsernameChange.call(actor: guardian.user, user:)
end
```

Best practices and guidelines

- Use namespaces for concepts. Most of the time, a model name is a business concept.
- Name services using a verb, describing the action (`CreateUser` not `UserCreator`).
- Don't repeat the concept name in the service name (`User::Create` is easily understandable, no need for `User::CreateUser`).
- If your service receives parameters, they should be validated through the `params` step.
- Don't put too much logic in a step. If logic becomes complex, prefer to use an action instead. It's better to offload complex logic to an action, as it will simplify the reasoning and the testing.
- Don't inject models into services. Only dependencies (like a guardian, or input parameters) should be injected. Models are expected to be fetched inside the service and have their error handled by the `model` step.
- Use a policy object when a policy logic becomes relatively complex and/or if you need to expose a custom reason why that policy failed. It could be a dynamic reason or simply because building the reason needs some dedicated logic.
- A good rule of thumb is to extract any logic that becomes relatively complex into dedicated objects. They can be models, PORO, actions, etc. Just don't try to always pack everything into the service itself.
- Likewise, avoid utility methods in a service. A service should only have step definitions. If some processing is needed, then it can probably be done in a contract, an action or extracted somewhere else.
- If an action is pretty complex (it has a lot of edge cases or several branching statements, for example), test it in isolation instead of testing it directly in the service specs. Then in the service specs, just ensure that action is properly called.

- When defining a method for a step, don’t provide default parameters, the framework won’t allow it. If you need a default value for something, it’s probably best to declare it in a contract.
- Try to follow the steps order when writing your methods or your tests.

Debugging

The main tool to help debugging a service is the steps inspector. However, it’s not a live debugging tool, as it inspects the result object once the service has run.

Steps Inspector

This small tool is very useful to debug the outcome of a service. The `Service::StepsInspector` class is not meant to be used directly, as there’s a shortcut available directly on any result object. Call `#inspect_steps` on a result object, and it will output all the steps of the service with their current state. This is how it looks like for the `User::UpdateUsername` service we’re using in our examples:

```
Inspecting User::UpdateUsername result object:

[1/6] [params] default (0.3581 ms) □
[2/6] [model] user (68.9291 ms) □
[3/6] [policy] can_update_username □

(3 more steps not shown as the execution flow was stopped before reaching them)
```

Here we can see each step is numbered to track the execution order easily. Then the type of the step is outputted, followed by its name and how much time the step took to run. Finally, there’s either a checkmark or a cross, depending on the step outcome.

So, we can see the `can_update_username` policy failed, and since it’s a simple policy it’s easy to see that the problem lies with the user not having enough permissions (through `guardian`). The policy is defined as:

```
def can_update_username(guardian:, user:)
  guardian.can_edit_username?(user)
end
```

In the case of a more complex result object, like with a contract, the steps inspector provides the error from the failing step. Let’s say we call our `User::UpdateUsername` service without providing any

parameters. It would then fail at the `params` step. The inspector now outputs this:

```
Inspecting User::UpdateUsername result object:

[1/6] [params] default □

(5 more steps not shown as the execution flow was stopped before reaching them)

Why it failed:

#<ActiveModel::Errors [#<ActiveModel::Error attribute=id, type=blank, options={}>,
#<ActiveModel::Error attribute=username, type=blank, options={}>, #<ActiveModel::Error
attribute=username, type=invalid, options={:value=>nil}>]>

Provided parameters: {"id"=>nil, "username"=>nil}
```

Here we can see `ActiveModel` errors, telling us `id` and `username` were blank. The provided parameters are also outputted to help debugging.

Here's a recap of what errors will be outputted for the different steps:

- *model*: when the model is an `ActiveRecord` one, it outputs its validation errors. Otherwise, it outputs the reason why it failed, probably a `Model not found` error.
- *params*: outputs the validation errors followed by the provided parameters.
- *policy*: doesn't output anything for a simple policy. When a policy object is used, then it outputs its `reason`.
- *try*: outputs the exception caught by `try`.
- *lock*: outputs the lock wasn't acquired and provides the lock name.
- *step*: outputs the message provided to `fail!`.

Live debugging

We don't have a live debugging tool (yet), but it's not that hard to make sense of what's happening in a service.

The simplest thing to do is to put a `binding.pry` statement inside any step you want to inspect. It's just a method, so you'll have access to its parameters and to the `context` object. To inspect it, you can call `#to_h` on it and see what keys and values it holds. Remember, if your pry session doesn't open, it means a step before the one you're trying to inspect failed.

□□□ #2
□ □□□ □□□ 2026-10-31 13:28:20 UTC
□ □□□ □□□ 2026-10-31 13:28:22 UTC