Error Reporting in Rails Applications

This guide introduces ways to manage errors in a Rails application.

After reading this guide, you will know:

- How to use Rails' error reporter to capture and report errors.
- How to create custom subscribers for your error-reporting service.

Chapters

1 Error Reporting

The Rails error reporter provides a standard way to collect errors that occur in your application and report them to your preferred service or location (e.g. you could report the errors to a monitoring service such as Sentry).

It aims to replace boilerplate error-handling code like this:

```
begin
  do_something
rescue SomethingIsBroken => error
  MyErrorReportingService.notify(error)
end
```

with a consistent interface:

```
Rails.error.handle(SomethingIsBroken) do
  do_something
end
```

Rails wraps all executions (such as HTTP requests, jobs, and rails runner invocations) in the error reporter, so any unhandled errors raised in your app will automatically be reported to your error-reporting service via their subscribers.

This means that third-party error-reporting libraries no longer need to insert a Rack middleware or do any *monkey-patching* to capture unhandled errors. Libraries that use Active Support can also use this to non-intrusively report warnings that would previously have been lost in logs.

> Using the Rails error reporter is optional, as other means of capturing errors still work.

1.1 Subscribing to the Reporter

To use the error reporter with an external service, you need a *subscriber*. A subscriber can be any Ruby object with a `report` method. When an error occurs in your application or is manually reported, the Rails error reporter will call this method with the error object and some options.

> Some error-reporting libraries, such as Sentry's and Honeybadger's, automatically register a subscriber for you.

You may also create a custom subscriber. For example:

```
# config/initializers/error_subscriber.rb
class ErrorSubscriber
  def report(error, handled:, severity:, context:, source: nil)
    MyErrorReportingService.report_error(error, context: context, handled:
handled, level: severity)
  end
end
```

After defining the subscriber class, you can register it by calling the `Rails.error.subscribe` method:

```
Rails.error.subscribe(ErrorSubscriber.new)
```

You can register as many subscribers as you wish. Rails will call them in the order in which they were registered.

It is also possible to unregister a subscriber by calling `Rails.error.unsubscribe`. This may be useful if you'd like to replace or remove a subscriber added by one of your dependencies. Both `subscribe` and `unsubscribe` can take either a subscriber or a class as follows:

```
subscriber = ErrorSubscriber.new
Rails.error.unsubscribe(subscriber)
# or
Rails.error.unsubscribe(ErrorSubscriber)
```

> The Rails error reporter will always call registered subscribers, regardless of your environment. However, many error-reporting services only report errors in production by default. You should configure and test your setup across environments as needed.

1.2 Using the Error Reporter

Rails error reporter has four methods that allow you to report methods in different ways:

- `Rails.error.handle`
- `Rails.error.record`
- `Rails.error.report`
- `Rails.error.unexpected`

1.2.1 Reporting and Swallowing Errors

The `Rails.error.handle` method will report any error raised within the block. It will then **swallow** the error, and the rest of your code outside the block will continue as normal.

```
result = Rails.error.handle do
  1 + "1" # raises TypeError
end
result # => nil
1 + 1 # This will be executed
```

If no error is raised in the block, `Rails.error.handle` will return the result of the block, otherwise it will return `nil`. You can override this by providing a `fallback`:

```
user = Rails.error.handle(fallback: -> { User.anonymous }) do
  User.find(params[:id])
end
```

1.2.2 Reporting and Re-raising Errors

The `Rails.error.record` method will report errors to all registered subscribers and then **re-raise** the error, meaning that the rest of your code won't execute.

```
Rails.error.record do
  1 + "1" # raises TypeError
end
1 + 1 # This won't be executed
```

If no error is raised in the block, `Rails.error.record` will return the result of the block.

1.2.3 Manually Reporting Errors

You can also manually report errors by calling `Rails.error.report`:

```
begin
  # code
rescue StandardError => e
  Rails.error.report(e)
end
```

Any options you pass will be passed on to the error subscribers.

1.2.4 Reporting Unexpected Errors

You can report any unexpected error by calling `Rails.error.unexpected`.

When called in production, this method will return nil after the error is reported and the execution of your code will continue.

When called in development, the error will be wrapped in a new error class (to ensure it's not being rescued higher in the stack) and surfaced to the developer for debugging.

For example:

```
def edit
  if published?
    Rails.error.unexpected("[BUG] Attempting to edit a published article, that
shouldn't be possible")
    false
  end
  # ...
end
```

> This method is intended to gracefully handle any errors that may occur in production, but that aren't anticipated to be the result of typical use.

1.3 Error-reporting Options

The reporting APIs `#handle`, `#record`, and `#report` support the following options, which are then passed along to all registered subscribers:

- `handled`: a `Boolean` to indicate if the error was handled. This is set to `true` by default. `#record` sets this to `false`.
- `severity`: a `Symbol` describing the severity of the error. Expected values are: `:error`, `:warning`, and `:info`. `#handle` sets this to `:warning`, while `#record` sets it to `:error`.
- `context`: a `Hash` to provide more context about the error, like request or user details
- `source`: a `String` about the source of the error. The default source is `"application"`. Errors reported by internal libraries may set other sources; the Redis cache library may use `"redis_cache_store.active_support"`, for instance. Your subscriber can use the source to ignore errors you aren't interested in.

```
Rails.error.handle(context: { user_id: user.id }, severity: :info) do
  # ...
end
```

1.4 Setting Context Globally

In addition to setting context through the `context` option, you can use `Rails.error.set_context`. For example:

```
Rails.error.set_context(section: "checkout", user_id: @user.id)
```

Any context set this way will be merged with the `context` option

```
Rails.error.set_context(a: 1)
Rails.error.handle(context: { b: 2 }) { raise }
# The reported context will be: {:a=>1, :b=>2}
Rails.error.handle(context: { b: 3 }) { raise }
# The reported context will be: {:a=>1, :b=>3}
```

1.5 Filtering by Error Classes

With `Rails.error.handle` and `Rails.error.record`, you can also choose to only report errors of certain classes. For example:

```
Rails.error.handle(IOError) do
  1 + "1" # raises TypeError
end
1 + 1 # TypeErrors are not IOErrors, so this will *not* be executed
```

Here, the `TypeError` will not be captured by the Rails error reporter. Only instances of `IOError` and its descendants will be reported. Any other errors will be raised as normal.

1.6 Disabling Notifications

You can prevent a subscriber from being notified of errors for the duration of a block by calling `Rails.error.disable`. Similarly to `subscribe` and `unsubscribe`, you can pass in either the subscriber itself, or its class.

```
Rails.error.disable(ErrorSubscriber) do
  1 + "1" # TypeError will not be reported via the ErrorSubscriber
end
```

> This can also be helpful for third-party error reporting services who may want to manage error handling a different way, or higher in the stack.

2 Error-reporting Libraries

Error-reporting libraries can register their subscribers in a Railtie:

```
module MySdk
  class Railtie < ::Rails::Railtie
    initializer "my_sdk.error_subscribe" do
      Rails.error.subscribe(MyErrorSubscriber.new)
    end
  end
end
```

> If you register an error subscriber, but still have other error mechanisms like a Rack middleware, you may end up with errors reported multiple times. You should either remove your other mechanisms or adjust your report functionality so it skips reporting an error it has seen before.

Feedback

You're encouraged to help improve the quality of this guide.

Please contribute if you see any typos or factual errors. To get started, you can read our documentation contributions section.

You may also find incomplete content or stuff that is not up to date. Please do add any missing documentation for main. Make sure to check Edge Guides first to verify if the issues are already fixed or not on the main branch. Check the Ruby on Rails Guides Guidelines for style and conventions.

If for whatever reason you spot something to fix but cannot patch it yourself, please open an issue.

And last but not least, any kind of discussion regarding Ruby on Rails documentation is very welcome on the official Ruby on Rails Forum.