DO NOT READ THIS FILE ON GITHUB, GUIDES ARE PUBLISHED ON https://guides.rubyonrails.org.

Autoloading and Reloading Constants

This guide documents how autoloading and reloading works in `zeitwerk` mode.

After reading this guide, you will know:

- Related Rails configuration
- Project structure
- Autoloading, reloading, and eager loading
- Single Table Inheritance
- And more

Introduction

INFO. This guide documents autoloading, reloading, and eager loading in Rails applications.

In a normal Ruby program, dependencies need to be loaded by hand. For example, the following controller uses classes `ApplicationController` and `Post`, and normally you'd need to put `require` calls for them:

```
# DO NOT DO THIS.
require "application_controller"
require "post"
# DO NOT DO THIS.

class PostsController < ApplicationController
  def index
    @posts = Post.all
  end
end
```

This is not the case in Rails applications, where application classes and modules are just available everywhere:

```
class PostsController < ApplicationController
  def index
    @posts = Post.all
  end
end
```

Idiomatic Rails applications only issue `require` calls to load stuff from their `lib` directory, the Ruby standard library, Ruby gems, etc. That is, anything that does not belong to their autoload paths, explained below.

To provide this feature, Rails manages a couple of Zeitwerk loaders on your behalf.

Project Structure

In a Rails application file names have to match the constants they define, with directories acting as namespaces.

For example, the file `app/helpers/users_helper.rb` should define `UsersHelper` and the file `app/controllers/admin/payments_controller.rb` should define `Admin::PaymentsController`.

By default, Rails configures Zeitwerk to inflect file names with `String#camelize`. For example, it expects that `app/controllers/users_controller.rb` defines the constant `UsersController` because that is what `"users_controller".camelize` returns.

The section *Customizing Inflections* below documents ways to override this default.

Please, check the Zeitwerk documentation for further details.

config.autoload_paths

We refer to the list of application directories whose contents are to be autoloaded and (optionally) reloaded as *autoload paths*. For example, `app/models`. Such directories represent the root namespace: `Object`.

INFO. Autoload paths are called *root directories* in Zeitwerk documentation, but we'll stay with "autoload path" in this guide.

Within an autoload path, file names must match the constants they define as documented here.

By default, the autoload paths of an application consist of all the subdirectories of `app` that exist when the application boots —except for `assets`, `javascript`, and `views`— plus the autoload paths of engines it might depend on.

For example, if `UsersHelper` is implemented in `app/helpers/users_helper.rb`, the module is autoloadable, you do not need (and should not write) a `require` call for it:

```
$ bin/rails runner 'p UsersHelper'
UsersHelper
```

Rails adds custom directories under `app` to the autoload paths automatically. For example, if your application has `app/presenters`, you don't need to configure anything in order to autoload presenters, it works out of the box.

The array of default autoload paths can be extended by pushing to `config.autoload_paths`, in `config/application.rb` or `config/environments/*.rb`.

For example:

```
module MyApplication
  class Application < Rails::Application
    config.autoload_paths << "#{root}/extras"
  end
end
```

Also, engines can push in body of the engine class and in their own `config/environments/*.rb`.

WARNING. Please do not mutate `ActiveSupport::Dependencies.autoload_paths`; the public interface to change autoload paths is `config.autoload_paths`.

WARNING: You cannot autoload code in the autoload paths while the application boots. In particular, directly in `config/initializers/*.rb`. Please check *Autoloading when the application boots* down below for valid ways to do that.

The autoload paths are managed by the `Rails.autoloaders.main` autoloader.

config.autoload_once_paths

You may want to be able to autoload classes and modules without reloading them. The `autoload_once_paths` configuration stores code that can be autoloaded, but won't be reloaded.

By default, this collection is empty, but you can extend it pushing to `config.autoload_once_paths`. You can do so in `config/application.rb` or `config/environments/*.rb`. For example:

```
module MyApplication
  class Application < Rails::Application
    config.autoload_once_paths << "#{root}/app/serializers"
  end
end
```

Also, engines can push in body of the engine class and in their own `config/environments/*.rb`.

INFO. If `app/serializers` is pushed to `config.autoload_once_paths`, Rails no longer considers this an autoload path, despite being a custom directory under `app`. This setting overrides that rule.

This is key for classes and modules that are cached in places that survive reloads, like the Rails framework itself.

For example, Active Job serializers are stored inside Active Job:

```
# config/initializers/custom_serializers.rb
Rails.application.config.active_job.custom_serializers << MoneySerializer
```

and Active Job itself is not reloaded when there's a reload, only application and engines code in the autoload paths is.

Making `MoneySerializer` reloadable would be confusing, because reloading an edited version would have no effect on that class object stored in Active Job. Indeed, if `MoneySerializer` was reloadable, starting with Rails 7 such initializer would raise a `NameError`.

Another use case is when engines decorate framework classes:

```
initializer "decorate ActionController::Base" do
  ActiveSupport.on_load(:action_controller_base) do
    include MyDecoration
  end
end
```

There, the module object stored in `MyDecoration` by the time the initializer runs becomes an ancestor of `ActionController::Base`, and reloading `MyDecoration` is pointless, it won't affect that ancestor chain.

Classes and modules from the autoload once paths can be autoloaded in `config/initializers`. So, with that configuration this works:

```
# config/initializers/custom_serializers.rb
Rails.application.config.active_job.custom_serializers << MoneySerializer
```

INFO: Technically, you can autoload classes and modules managed by the `once` autoloader in any initializer that runs after `:bootstrap_hook`.

The autoload once paths are managed by `Rails.autoloaders.once`.

$LOAD_PATH

Autoload paths are added to `$LOAD_PATH` by default. However, Zeitwerk uses absolute file names internally, and your application should not issue `require` calls for autoloadable files, so those directories are actually not needed there. You can opt out with this flag:

```
config.add_autoload_paths_to_load_path = false
```

That may speed up legitimate `require` calls a bit since there are fewer lookups. Also, if your application uses Bootsnap, that saves the library from building unnecessary indexes, leading to lower memory usage.

Reloading

Rails automatically reloads classes and modules if application files in the autoload paths change.

More precisely, if the web server is running and application files have been modified, Rails unloads all autoloaded constants managed by the `main` autoloader

just before the next request is processed. That way, application classes or modules used during that request will be autoloaded again, thus picking up their current implementation in the file system.

Reloading can be enabled or disabled. The setting that controls this behavior is `config.cache_classes`, which is false by default in `development` mode (reloading enabled), and true by default in `production` mode (reloading disabled).

Rails uses an evented file monitor to detect files changes by default. It can be configured instead to detect file changes by walking the autoload paths. This is controlled by the `config.file_watcher` setting.

In a Rails console there is no file watcher active regardless of the value of `config.cache_classes`. This is because, normally, it would be confusing to have code reloaded in the middle of a console session. Similar to an individual request, you generally want a console session to be served by a consistent, non-changing set of application classes and modules.

However, you can force a reload in the console by executing `reload!`:

```
irb(main):001:0> User.object_id
=> 70136277390120
irb(main):002:0> reload!
Reloading...
=> true
irb(main):003:0> User.object_id
=> 70136284426020
```

As you can see, the class object stored in the `User` constant is different after reloading.

Reloading and Stale Objects

It is very important to understand that Ruby does not have a way to truly reload classes and modules in memory, and have that reflected everywhere they are already used. Technically, "unloading" the `User` class means removing the `User` constant via `Object.send(:remove_const, "User")`.

For example, check out this Rails console session:

```
irb> joe = User.new
irb> reload!
irb> alice = User.new
irb> joe.class == alice.class
=> false
```

`joe` is an instance of the original `User` class. When there is a reload, the `User` constant then evaluates to a different, reloaded class. `alice` is an instance of the newly loaded `User`, but `joe` is not — his class is stale. You may define `joe`

again, start an IRB subsession, or just launch a new console instead of calling `reload!`.

Another situation in which you may find this gotcha is subclassing reloadable classes in a place that is not reloaded:

```
# lib/vip_user.rb
class VipUser < User
end
```

if `User` is reloaded, since `VipUser` is not, the superclass of `VipUser` is the original stale class object.

Bottom line: **do not cache reloadable classes or modules**.

Autoloading When the Application Boots

While booting, applications can autoload from the autoload once paths, which are managed by the `once` autoloader. Please check the section `config.autoload_once_paths` above.

However, you cannot autoload from the autoload paths, which are managed by the `main` autoloader. This applies to code in `config/initializers` as well as application or engines initializers.

Why? Initializers only run once, when the application boots. If you reboot the server, they run again in a new process, but reloading does not reboot the server, and initializers don't run again. Let's see the two main use cases.

Use case 1: During boot, load reloadable code

Autoload on boot and on each reload Let's imagine `ApiGateway` is a reloadable class from `app/services` managed by the `main` autoloader and you need to configure its endpoint while the application boots:

```
# config/initializers/api_gateway_setup.rb
ApiGateway.endpoint = "https://example.com" # DO NOT DO THIS
```

a reloaded `ApiGateway` would have a `nil` endpoint, because the code above does not run again.

You can still set things up during boot, but you need to wrap them in a `to_prepare` block, which runs on boot, and after each reload:

```
# config/initializers/api_gateway_setup.rb
Rails.application.config.to_prepare do
  ApiGateway.endpoint = "https://example.com" # CORRECT
end
```

NOTE: For historical reasons, this callback may run twice. The code it executes must be idempotent.

Autoload on boot only Reloadable classes and modules can be autoloaded in `after_initialize` blocks too. These run on boot, but do not run again on reload. In some exceptional cases this may be what you want.

Preflight checks are a use case for this:

```
# config/initializers/check_admin_presence.rb
Rails.application.config.after_initialize do
  unless Role.where(name: "admin").exists?
    abort "The admin role is not present, please seed the database."
  end
end
```

Use case 2: During boot, load code that remains cached

Some configurations take a class or module object, and they store it in a place that is not reloaded.

One example is middleware:

```
config.middleware.use MyApp::Middleware::Foo
```

When you reload, the middleware stack is not affected, so, whatever object was stored in `MyApp::Middleware::Foo` at boot time remains there stale.

Another example is Active Job serializers:

```
# config/initializers/custom_serializers.rb
Rails.application.config.active_job.custom_serializers << MoneySerializer
```

Whatever `MoneySerializer` evaluates to during initialization gets pushed to the custom serializers. If that was reloadable, the initial object would be still within Active Job, not reflecting your changes.

Yet another example are railties or engines decorating framework classes by including modules. For instance, `turbo-rails` decorates `ActiveRecord::Base` this way:

```
initializer "turbo.broadcastable" do
  ActiveSupport.on_load(:active_record) do
    include Turbo::Broadcastable
  end
end
```

That adds a module object to the ancestor chain of `ActiveRecord::Base`. Changes in `Turbo::Broadcastable` would have no effect if reloaded, the ancestor chain would still have the original one.

Corollary: Those classes or modules **cannot be reloadable**.

The easiest way to refer to those classes or modules during boot is to have them defined in a directory which does not belong to the autoload paths. For instance,

`lib` is an idiomatic choice. It does not belong to the autoload paths by default, but it does belong to `$LOAD_PATH`. Just perform a regular `require` to load it.

As noted above, another option is to have the directory that defines them in the autoload once paths and autoload. Please check the section about config.autoload_once_paths for details.

Eager Loading

In production-like environments it is generally better to load all the application code when the application boots. Eager loading puts everything in memory ready to serve requests right away, and it is also CoW-friendly.

Eager loading is controlled by the flag `config.eager_load`, which is enabled by default in `production` mode.

The order in which files are eager-loaded is undefined.

During eager loading, Rails invokes `Zeitwerk::Loader.eager_load_all`. That ensures all gem dependencies managed by Zeitwerk are eager-loaded too.

Single Table Inheritance

Single Table Inheritance is a feature that doesn't play well with lazy loading. The reason is that its API generally needs to be able to enumerate the STI hierarchy to work correctly, whereas lazy loading defers loading classes until they are referenced. You can't enumerate what you haven't referenced yet.

In a sense, applications need to eager load STI hierarchies regardless of the loading mode.

Of course, if the application eager loads on boot, that is already accomplished. When it does not, it is in practice enough to instantiate the existing types in the database, which in development or test modes is usually fine. One way to do that is to include an STI preloading module in your `lib` directory:

```
module StiPreload
  unless Rails.application.config.eager_load
    extend ActiveSupport::Concern

    included do
      cattr_accessor :preloaded, instance_accessor: false
    end

    class_methods do
      def descendants
        preload_sti unless preloaded
        super
      end
```

```
      # Constantizes all types present in the database. There might be more on
      # disk, but that does not matter in practice as far as the STI API is
      # concerned.
      #
      # Assumes store_full_sti_class is true, the default.
      def preload_sti
        types_in_db = \
          base_class.
            unscoped.
            select(inheritance_column).
            distinct.
            pluck(inheritance_column).
            compact

        types_in_db.each do |type|
          logger.debug("Preloading STI type #{type}")
          type.constantize
        end

        self.preloaded = true
      end
    end
  end
end
```

and then include it in the STI root classes of your project:

```
# app/models/shape.rb
require "sti_preload"

class Shape < ApplicationRecord
  include StiPreload # Only in the root class.
end

# app/models/polygon.rb
class Polygon < Shape
end

# app/models/triangle.rb
class Triangle < Polygon
end
```

Customizing Inflections

By default, Rails uses `String#camelize` to know which constant a given file or directory name should define. For example, `posts_controller.rb` should

define `PostsController` because that is what `"posts_controller".camelize` returns.

It could be the case that some particular file or directory name does not get inflected as you want. For instance, `html_parser.rb` is expected to define `HtmlParser` by default. What if you prefer the class to be `HTMLParser`? There are a few ways to customize this.

The easiest way is to define acronyms in `config/initializers/inflections.rb`:

```
ActiveSupport::Inflector.inflections(:en) do |inflect|
  inflect.acronym "HTML"
  inflect.acronym "SSL"
end
```

Doing so affects how Active Support inflects globally. That may be fine in some applications, but you can also customize how to camelize individual basenames independently from Active Support by passing a collection of overrides to the default inflectors:

```
# config/initializers/zeitwerk.rb
Rails.autoloaders.each do |autoloader|
  autoloader.inflector.inflect(
    "html_parser" => "HTMLParser",
    "ssl_error"   => "SSLError"
  )
end
```

That technique still depends on `String#camelize`, though, because that is what the default inflectors use as fallback. If you instead prefer not to depend on Active Support inflections at all and have absolute control over inflections, configure the inflectors to be instances of `Zeitwerk::Inflector`:

```
# config/initializers/zeitwerk.rb
Rails.autoloaders.each do |autoloader|
  autoloader.inflector = Zeitwerk::Inflector.new
  autoloader.inflector.inflect(
    "html_parser" => "HTMLParser",
    "ssl_error"   => "SSLError"
  )
end
```

There is no global configuration that can affect said instances; they are deterministic.

You can even define a custom inflector for full flexibility. Please check the Zeitwerk documentation for further details.

Autoloading and Engines

Engines run in the context of a parent application, and their code is autoloaded, reloaded, and eager loaded by the parent application. If the application runs in `zeitwerk` mode, the engine code is loaded by `zeitwerk` mode. If the application runs in `classic` mode, the engine code is loaded by `classic` mode.

When Rails boots, engine directories are added to the autoload paths, and from the point of view of the autoloader, there's no difference. Autoloaders' main inputs are the autoload paths, and whether they belong to the application source tree or to some engine source tree is irrelevant.

For example, this application uses Devise:

```
% bin/rails runner 'pp ActiveSupport::Dependencies.autoload_paths'
[".../app/controllers",
 ".../app/controllers/concerns",
 ".../app/helpers",
 ".../app/models",
 ".../app/models/concerns",
 ".../gems/devise-4.8.0/app/controllers",
 ".../gems/devise-4.8.0/app/helpers",
 ".../gems/devise-4.8.0/app/mailers"]
```

If the engine controls the autoloading mode of its parent application, the engine can be written as usual.

However, if an engine supports Rails 6 or Rails 6.1 and does not control its parent applications, it has to be ready to run under either `classic` or `zeitwerk` mode. Things to take into account:

1. If `classic` mode would need a `require_dependency` call to ensure some constant is loaded at some point, write it. While `zeitwerk` would not need it, it won't hurt, it will work in `zeitwerk` mode too.

2. `classic` mode underscores constant names ("User" -> "user.rb"), and `zeitwerk` mode camelizes file names ("user.rb" -> "User"). They coincide in most cases, but they don't if there are series of consecutive uppercase letters as in "HTMLParser". The easiest way to be compatible is to avoid such names. In this case, pick "HtmlParser".

3. In `classic` mode, the file `app/model/concerns/foo.rb` is allowed to define both `Foo` and `Concerns::Foo`. In `zeitwerk` mode, there's only one option: it has to define `Foo`. In order to be compatible, define `Foo`.

Testing

Manual Testing

The task `zeitwerk:check` checks if the project tree follows the expected naming conventions and it is handy for manual checks. For example, if you're migrating

from `classic` to `zeitwerk` mode, or if you're fixing something:

```
% bin/rails zeitwerk:check
Hold on, I am eager loading the application.
All is good!
```

There can be additional output depending on the application configuration, but the last "All is good!" is what you are looking for.

Automated Testing

It is a good practice to verify in the test suite that the project eager loads correctly.

That covers Zeitwerk naming compliance and other possible error conditions. Please check the section about testing eager loading in the *Testing Rails Applications* guide.

Troubleshooting

The best way to follow what the loaders are doing is to inspect their activity.

The easiest way to do that is to include

```
Rails.autoloaders.log!
```

in `config/application.rb` after loading the framework defaults. That will print traces to standard output.

If you prefer logging to a file, configure this instead:

```
Rails.autoloaders.logger = Logger.new("#{Rails.root}/log/autoloading.log")
```

The Rails logger is not yet available when `config/application.rb` executes. If you prefer to use the Rails logger, configure this setting in an initializer instead:

```
# config/initializers/log_autoloaders.rb
Rails.autoloaders.logger = Rails.logger
```

Rails.autoloaders

The Zeitwerk instances managing your application are available at

```
Rails.autoloaders.main
Rails.autoloaders.once
```

The predicate

```
Rails.autoloaders.zeitwerk_enabled?
```

is still available in Rails 7 applications, and returns `true`.