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

Classic to Zeitwerk HOWTO

This guide documents how to migrate Rails applications from `classic` to `zeitwerk` mode.

After reading this guide, you will know:

- What are `classic` and `zeitwerk` modes
- Why switch from `classic` to `zeitwerk`
- How to activate `zeitwerk` mode
- How to verify your application runs in `zeitwerk` mode
- How to verify your project loads OK in the command line
- How to verify your project loads OK in the test suite
- How to address possible edge cases
- New features in Zeitwerk you can leverage

What are `classic` and `zeitwerk` Modes?

From the very beginning, and up to Rails 5, Rails used an autoloader implemented in Active Support. This autoloader is known as `classic` and is still available in Rails 6.x. Rails 7 does not include this autoloader anymore.

Starting with Rails 6, Rails ships with a new and better way to autoload, which delegates to the Zeitwerk gem. This is `zeitwerk` mode. By default, applications loading the 6.0 and 6.1 framework defaults run in `zeitwerk` mode, and this is the only mode available in Rails 7.

Why Switch from `classic` to `zeitwerk`?

The `classic` autoloader has been extremely useful, but had a number of issues that made autoloading a bit tricky and confusing at times. Zeitwerk was developed to address this, among other motivations.

When upgrading to Rails 6.x, it is highly encouraged to switch to `zeitwerk` mode because it is a better autoloader, `classic` mode is deprecated.

Rails 7 ends the transition period and does not include `classic` mode.

I am Scared

Don't be :).

Zeitwerk was designed to be as compatible with the classic autoloader as possible. If you have a working application autoloading correctly today, chances are the switch will be easy. Many projects, big and small, have reported really smooth switches.

This guide will help you change the autoloader with confidence.

If for whatever reason you find a situation you don't know how to resolve, don't hesitate to open an issue in `rails/rails` and tag `@fxn`.

How to Activate `zeitwerk` Mode

Applications running Rails 5.x or Less

In applications running a Rails version previous to 6.0, `zeitwerk` mode is not available. You need to be at least in Rails 6.0.

Applications running Rails 6.x

In applications running Rails 6.x there are two scenarios.

If the application is loading the framework defaults of Rails 6.0 or 6.1 and it is running in `classic` mode, it must be opting out by hand. You have to have something similar to this:

```
# config/application.rb
config.load_defaults 6.0
config.autoloader = :classic # DELETE THIS LINE
```

As noted, just delete the override, `zeitwerk` mode is the default.

On the other hand, if the application is loading old framework defaults you need to enable `zeitwerk` mode explicitly:

```
# config/application.rb
config.load_defaults 5.2
config.autoloader = :zeitwerk
```

Applications Running Rails 7

In Rails 7 there is only `zeitwerk` mode, you do not need to do anything to enable it.

Indeed, in Rails 7 the setter `config.autoloader=` does not even exist. If `config/application.rb` uses it, please delete the line.

How to Verify The Application Runs in `zeitwerk` Mode?

To verify the application is running in `zeitwerk` mode, execute

```
bin/rails runner 'p Rails.autoloaders.zeitwerk_enabled?'
```

If that prints `true`, `zeitwerk` mode is enabled.

Does my Application Comply with Zeitwerk Conventions?

config.eager_load_paths

Compliance test runs only for eager loaded files. Therefore, in order to verify Zeitwerk compliance, it is recommended to have all autoload paths in the eager load paths.

This is already the case by default, but if the project has custom autoload paths configured just like this:

```
config.autoload_paths << "#{Rails.root}/extras"
```

those are not eager loaded and won't be verified. Adding them to the eager load paths is easy:

```
config.autoload_paths << "#{Rails.root}/extras"
config.eager_load_paths << "#{Rails.root}/extras"
```

zeitwerk:check

Once `zeitwerk` mode is enabled and the configuration of eager load paths double-checked, please run:

```
bin/rails zeitwerk:check
```

A successful check looks like this:

```
% 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.

If the double-check explained in the previous section determined actually there have to be some custom autoload paths outside the eager load paths, the task will detect and warn about them. However, if the test suite loads those files successfully, you're good.

Now, if there's any file that does not define the expected constant, the task will tell you. It does so one file at a time, because if it moved on, the failure loading one file could cascade into other failures unrelated to the check we want to run and the error report would be confusing.

If there's one constant reported, fix that particular one and run the task again. Repeat until you get "All is good!".

Take for example:

```
% bin/rails zeitwerk:check
Hold on, I am eager loading the application.
expected file app/models/vat.rb to define constant Vat
```

VAT is an European tax. The file `app/models/vat.rb` defines `VAT` but the autoloader expects `Vat`, why?

Acronyms

This is the most common kind of discrepancy you may find, it has to do with acronyms. Let's understand why do we get that error message.

The classic autoloader is able to autoload `VAT` because its input is the name of the missing constant, `VAT`, invokes `underscore` on it, which yields `vat`, and looks for a file called `vat.rb`. It works.

The input of the new autoloader is the file system. Give the file `vat.rb`, Zeitwerk invokes `camelize` on `vat`, which yields `Vat`, and expects the file to define the constant `Vat`. That is what the error message says.

Fixing this is easy, you only need to tell the inflector about this acronym:

```
# config/initializers/inflections.rb
ActiveSupport::Inflector.inflections(:en) do |inflect|
```

```
  inflect.acronym "VAT"
end
```

Doing so affects how Active Support inflects globally. That may be fine, but if you prefer you can also pass overrides to the inflectors used by the autoloaders:

```
# config/initializers/zeitwerk.rb
Rails.autoloaders.main.inflector.inflect("vat" => "VAT")
```

With this option you have more control, because only files called exactly `vat.rb` or directories exactly called `vat` will be inflected as `VAT`. A file called `vat_rules.rb` is not affected by that and can define `VatRules` just fine. This may be handy if the project has this kind of naming inconsistencies.

With that in place, the check passes!

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

Once all is good, it is recommended to keep validating the project in the test suite. The section *Check Zeitwerk Compliance in the Test Suite* explains how to do this.

Concerns

You can autoload and eager load from a standard structure with `concerns` subdirectories like

```
app/models
app/models/concerns
```

By default, `app/models/concerns` belongs to the autoload paths and therefore it is assumed to be a root directory. So, by default, `app/models/concerns/foo.rb` should define `Foo`, not `Concerns::Foo`.

If your application uses `Concerns` as namespace, you have two options:

1. Remove the `Concerns` namespace from those classes and modules and update client code.
2. Leave things as they are by removing `app/models/concerns` from the autoload paths:

```
# config/initializers/zeitwerk.rb
ActiveSupport::Dependencies.
  autoload_paths.
  delete("#{Rails.root}/app/models/concerns")
```

Having `app` in the autoload paths

Some projects want something like `app/api/base.rb` to define `API::Base`, and add `app` to the autoload paths to accomplish that.

Since Rails adds all subdirectories of `app` to the autoload paths automatically (with a few exceptions), we have another situation in which there are nested root directories, similar to what happens with `app/models/concerns`. That setup no longer works as is.

However, you can keep that structure, just delete `app/api` from the autoload paths in an initializer:

```
# config/initializers/zeitwerk.rb
ActiveSupport::Dependencies.
  autoload_paths.
  delete("#{Rails.root}/app/api")
```

Beware of subdirectories that do not have files to be autoloaded/eager loaded. For example, if the application has `app/admin` with resources for ActiveAdmin, you need to ignore them. Same for `assets` and friends:

```
# config/initializers/zeitwerk.rb
Rails.autoloaders.main.ignore(
  "app/admin",
  "app/assets",
  "app/javascripts",
  "app/views"
)
```

Without that configuration, the application would eager load those trees. Would err on `app/admin` because its files do not define constants, and would define a `Views` module, for example, as an unwanted side-effect.

As you see, having `app` in the autoload paths is technically possible, but a bit tricky.

Autoloaded Constants and Explicit Namespaces

If a namespace is defined in a file, as `Hotel` is here:

```
app/models/hotel.rb         # Defines Hotel.
app/models/hotel/pricing.rb # Defines Hotel::Pricing.
```

the `Hotel` constant has to be set using the `class` or `module` keywords. For example:

```
class Hotel
end
```

is good.

Alternatives like

```
Hotel = Class.new
```

or

```
Hotel = Struct.new
```

won't work, child objects like `Hotel::Pricing` won't be found.

This restriction only applies to explicit namespaces. Classes and modules not defining a namespace can be defined using those idioms.

One file, one constant (at the same top-level)

In `classic` mode you could technically define several constants at the same top-level and have them all reloaded. For example, given

```
# app/models/foo.rb

class Foo
end

class Bar
end
```

while `Bar` could not be autoloaded, autoloading `Foo` would mark `Bar` as autoloaded too.

This is not the case in `zeitwerk` mode, you need to move `Bar` to its own file `bar.rb`. One file, one top-level constant.

This affects only to constants at the same top-level as in the example above. Inner classes and modules are fine. For example, consider

```
# app/models/foo.rb

class Foo
  class InnerClass
  end
end
```

If the application reloads `Foo`, it will reload `Foo::InnerClass` too.

Globs in `config.autoload_paths`

Beware of configurations that use wildcards like

```
config.autoload_paths += Dir["#{config.root}/extras/**/"]
```

Every element of `config.autoload_paths` should represent the top-level namespace (`Object`). That won't work.

To fix this, just remove the wildcards:

```
config.autoload_paths << "#{config.root}/extras"
```

Decorating Classes and Modules from Engines

If your application decorates classes or modules from an engine, chances are it is doing something like this somewhere:

```
config.to_prepare do
  Dir.glob("#{Rails.root}/app/overrides/**/*_override.rb").each do |override|
```

```
    require_dependency override
  end
end
```

That has to be updated: You need to tell the `main` autoloader to ignore the directory with the overrides, and you need to load them with `load` instead. Something like this:

```
overrides = "#{Rails.root}/app/overrides"
Rails.autoloaders.main.ignore(overrides)
config.to_prepare do
  Dir.glob("#{overrides}/**/*_override.rb").each do |override|
    load override
  end
end
```

`before_remove_const`

Rails 3.1 added support for a callback called `before_remove_const` that was invoked if a class or module responded to this method and was about to be reloaded. This callback has remained otherwise undocumented and it is unlikely that your code uses it.

However, in case it does, you can rewrite something like

```
class Country < ActiveRecord::Base
  def self.before_remove_const
    expire_redis_cache
  end
end
```

as

```
# config/initializers/country.rb
unless Rails.application.config.cache_classes
  Rails.autoloaders.main.on_unload("Country") do |klass, _abspath|
    klass.expire_redis_cache
  end
end
```

Spring and the `test` Environment

Spring reloads the application code if something changes. In the `test` environment you need to enable reloading for that to work:

```
# config/environments/test.rb
config.cache_classes = false
```

Otherwise you'll get this error:

```
reloading is disabled because config.cache_classes is true
```

This has no performance penalty.

Bootsnap

Please make sure to depend on at least Bootsnap 1.4.4.

Check Zeitwerk Compliance in the Test Suite

The task `zeitwerk:check` is handy while migrating. Once the project is compliant, it is recommended to automate this check. In order to do so, it is enough to eager load the application, which is all `zeitwerk:check` does, indeed.

Continuous Integration

If your project has continuous integration in place, it is a good idea to eager load the application when the suite runs there. If the application cannot be eager loaded for whatever reason, you want to know in CI, better than in production, right?

CIs typically set some environment variable to indicate the test suite is running there. For example, it could be `CI`:

```
# config/environments/test.rb
config.eager_load = ENV["CI"].present?
```

Starting with Rails 7, newly generated applications are configured that way by default.

Bare Test Suites

If your project does not have continuous integration, you can still eager load in the test suite by calling `Rails.application.eager_load!`:

minitest

```
require "test_helper"

class ZeitwerkComplianceTest < ActiveSupport::TestCase
  test "eager loads all files without errors" do
    assert_nothing_raised { Rails.application.eager_load! }
  end
end
```

RSpec

```
require "rails_helper"

RSpec.describe "Zeitwerk compliance" do
  it "eager loads all files without errors" do
    expect { Rails.application.eager_load! }.not_to raise_error
  end
end
```

Delete any `require` calls

In my experience, projects generally do not do this. But I've seen a couple, and have heard of a few others.

In Rails application you use `require` exclusively to load code from `lib` or from 3rd party like gem dependencies or the standard library. **Never load autoloadable application code with `require`**. See why this was a bad idea already in `classic` here.

```
require "nokogiri" # GOOD
require "net/http" # GOOD
require "user"     # BAD, DELETE THIS (assuming app/models/user.rb)
```

Please delete any `require` calls of that type.

New Features You Can Leverage

Delete `require_dependency` calls

All known use cases of `require_dependency` have been eliminated with Zeitwerk. You should grep the project and delete them.

If your application uses Single Table Inheritance, please see the Single Table Inheritance section of the Autoloading and Reloading Constants (Zeitwerk Mode) guide.

Qualified Names in Class and Module Definitions Are Now Possible

You can now robustly use constant paths in class and module definitions:

```
# Autoloading in this class body matches Ruby semantics now.
class Admin::UsersController < ApplicationController
  # ...
end
```

A gotcha to be aware of is that, depending on the order of execution, the classic autoloader could sometimes be able to autoload `Foo::Wadus` in

```
class Foo::Bar
  Wadus
end
```

That does not match Ruby semantics because `Foo` is not in the nesting, and won't work at all in `zeitwerk` mode. If you find such corner case you can use the qualified name `Foo::Wadus`:

```
class Foo::Bar
  Foo::Wadus
end
```

or add `Foo` to the nesting:

```
module Foo
  class Bar
    Wadus
  end
end
```

Thread-safety Everywhere

In `classic` mode, constant autoloading is not thread-safe, though Rails has locks in place for example to make web requests thread-safe.

Constant autoloading is thread-safe in `zeitwerk` mode. For example, you can now autoload in multi-threaded scripts executed by the `runner` command.

Eager Loading and Autoloading are Consistent

In `classic` mode, if `app/models/foo.rb` defines `Bar`, you won't be able to autoload that file, but eager loading will work because it loads files recursively blindly. This can be a source of errors if you test things first eager loading, execution may fail later autoloading.

In `zeitwerk` mode both loading modes are consistent, they fail and err in the same files.