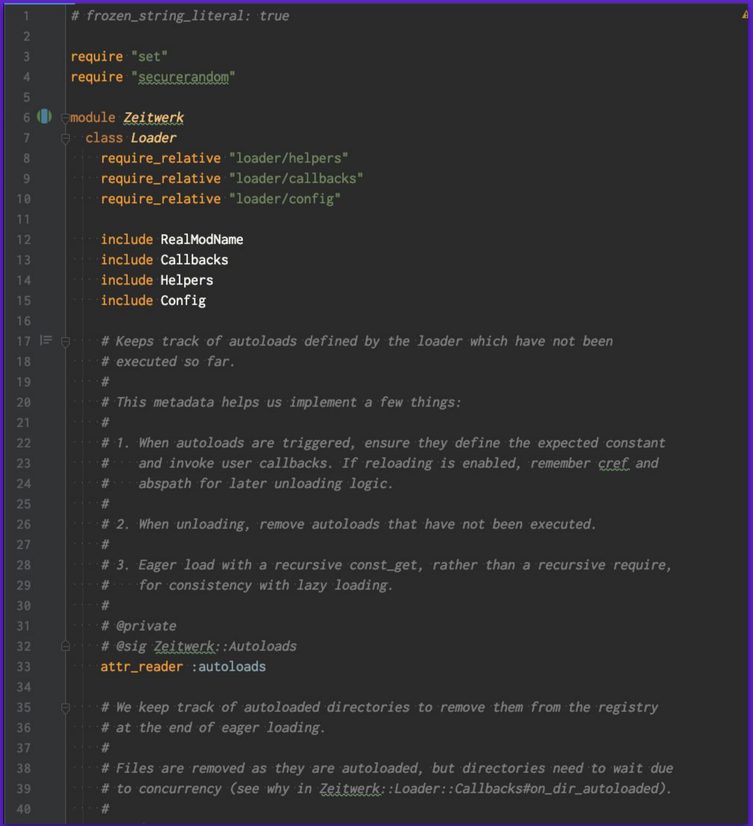
Rails and Zeitwerk

GOCARDLESS

What is Zeitwerk

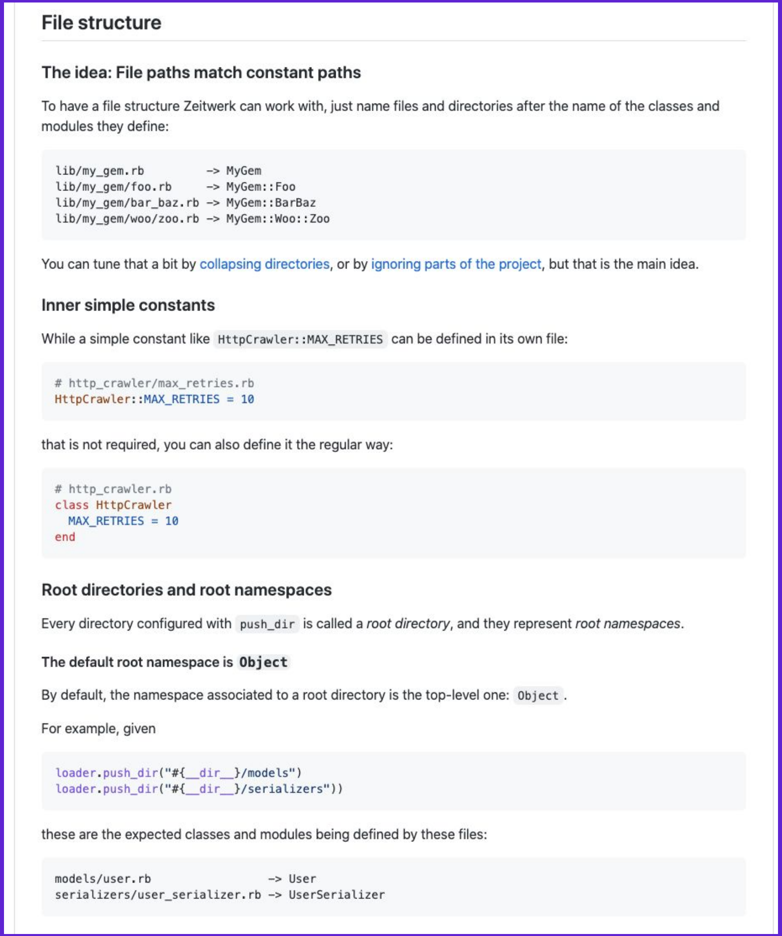
- Zeitwerk is an autoloader for Ruby files
- It replaces the “classic” Rails autoloader
- Lets us choose where to load code from
- Means we don’t have to put “require” statements everywhere
- Makes it easier to relocate classes
- Unlocks the use of Packwerk components



```ruby
# frozen_string_literal: true

require "set"
require "securerandom"

module Zeitwerk
  class Loader
    require_relative "loader/helpers"
    require_relative "loader/callbacks"
    require_relative "loader/config"

    include RealModName
    include Callbacks
    include Helpers
    include Config

    # Keeps track of autoloads defined by the loader which have not been
    # executed so far.
    #
    # This metadata helps us implement a few things:
    #
    # 1. When autoloads are triggered, ensure they define the expected constant
    #    and invoke user callbacks. If reloading is enabled, remember cref and
    #    abspath for later unloading logic.
    #
    # 2. When unloading, remove autoloads that have not been executed.
    #
    # 3. Eager load with a recursive const_get, rather than a recursive require,
    #    for consistency with lazy loading.
    #
    # @private
    # @sig Zeitwerk::Autoloads
    attr_reader :autoloads

    # We keep track of autoloaded directories to remove them from the registry
    # at the end of eager loading.
    #
    # Files are removed as they are autoloaded, but directories need to wait due
    # to concurrency (see why in Zeitwerk::Loader::Callbacks#on_dir_autoloaded).
    #
```

How does it know how to find my code?

- Naming conventions!
  - Similar to PHP's PSR-4
- For each root directory, subdirectories define modules
- Each subdirectory requires a new module
- Names are uniformly formatted as CamelCase
- If files or classes don't match the expected convention, Zeitwerk will refuse to load it
- We tell Zeitwerk which directories to look in, and it does the rest
- By default the root namespace is `Object` but this is also configurable for a root directory



File structure

The idea: File paths match constant paths

To have a file structure Zeitwerk can work with, just name files and directories after the name of the classes and modules they define:

```
lib/my_gem.rb         -> MyGem
lib/my_gem/foo.rb     -> MyGem::Foo
lib/my_gem/bar_baz.rb -> MyGem::BarBaz
lib/my_gem/woo/zoo.rb -> MyGem::Woo::Zoo
```

You can tune that a bit by collapsing directories, or by ignoring parts of the project, but that is the main idea.

Inner simple constants

While a simple constant like `HttpCrawler::MAX_RETRIES` can be defined in its own file:

```
# http_crawler/max_retries.rb
HttpCrawler::MAX_RETRIES = 10
```

that is not required, you can also define it the regular way:

```
# http_crawler.rb
class HttpCrawler
  MAX_RETRIES = 10
end
```

Root directories and root namespaces

Every directory configured with `push_dir` is called a *root directory*, and they represent *root namespaces*.

The default root namespace is `Object`

By default, the namespace associated to a root directory is the top-level one: `Object`.

For example, given

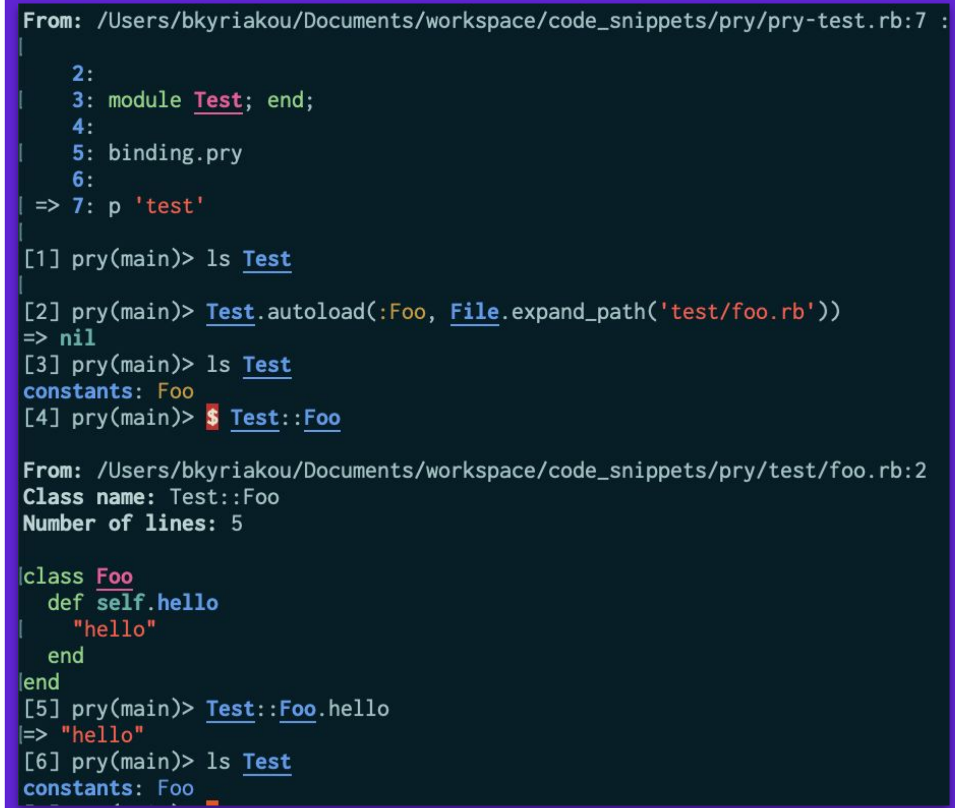
```
loader.push_dir("#{__dir__}/models")
loader.push_dir("#{__dir__}/serializers"))
```

these are the expected classes and modules being defined by these files:

```
models/user.rb                 -> User
serializers/user_serializer.rb -> UserSerializer
```

How does it werk?

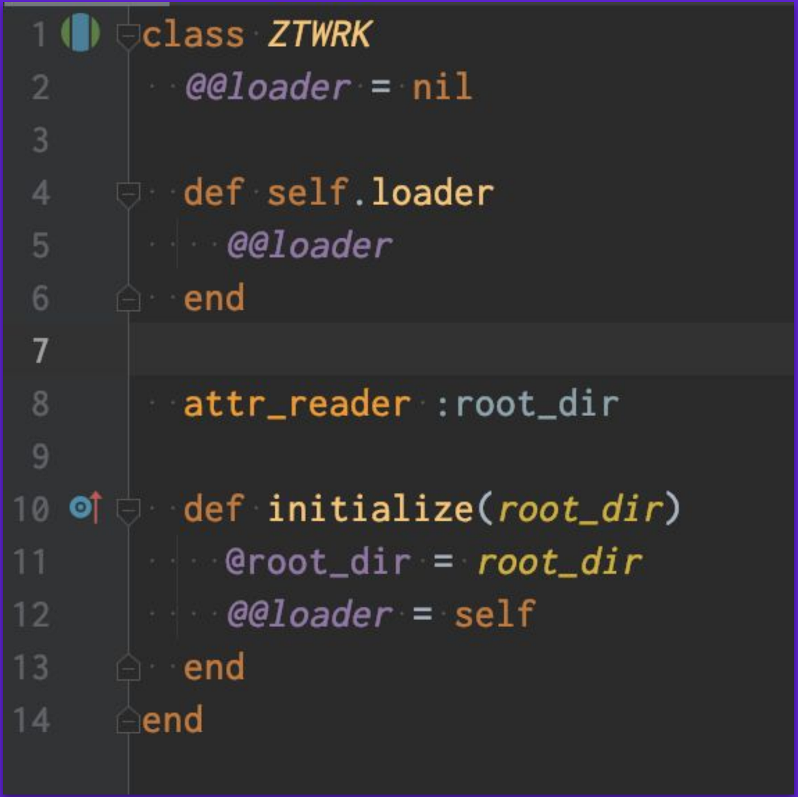
- Makes use of the `.autoload()` method
- This is a core Ruby method on all objects
- Lets us tell Ruby where to load a class from
- The root namespace is `Object`



```
From: /Users/bkyriakou/Documents/workspace/code_snippets/pry/pry-test.rb:7 :

    2:
    3: module Test; end;
    4:
    5: binding.pry
    6:
 => 7: p 'test'

[1] pry(main)> ls Test

[2] pry(main)> Test.autoload(:Foo, File.expand_path('test/foo.rb'))
=> nil
[3] pry(main)> ls Test
constants: Foo
[4] pry(main)> $ Test::Foo

From: /Users/bkyriakou/Documents/workspace/code_snippets/pry/test/foo.rb:2
Class name: Test::Foo
Number of lines: 5

class Foo
  def self.hello
    "hello"
  end
end
[5] pry(main)> Test::Foo.hello
=> "hello"
[6] pry(main)> ls Test
constants: Foo
```

A simple example

- An example of a very simplified autoloader that uses the principles Zeitwerk does
- We create a class that will store an internal reference to the current loader if it exists



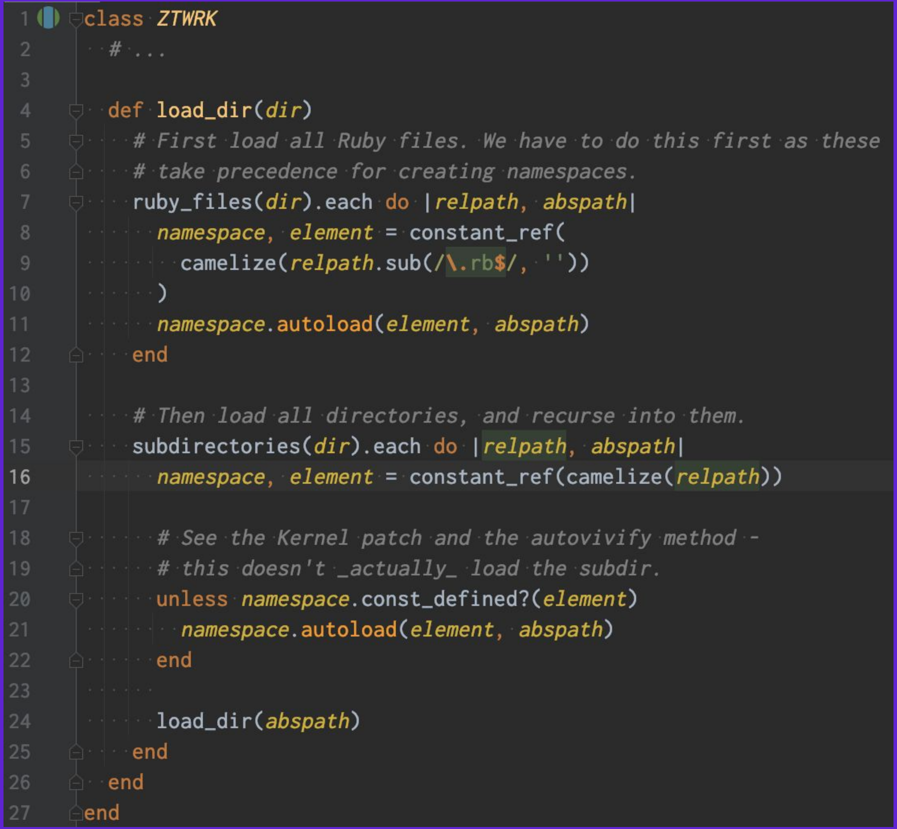
```ruby
class ZTWRK
  @@loader = nil

  def self.loader
    @@loader
  end

  attr_reader :root_dir

  def initialize(root_dir)
    @root_dir = root_dir
    @@loader = self
  end
end
```

https://github.com/benkyriakou/ztwrk

A simple example

- Now we add the main loading method
- This first autoloads namespaces from Ruby files
- Then it autoloads any undefined namespaces from subdirectories
- `constant_ref` is the equivalent of Zeitwerk's `cref` method
    - Takes a camelized relative path like "`Foo::Bar`"
    - Returns the namespace (constant `Foo`) and the element (symbol `:Bar`)

https://github.com/benkyriakou/ztwrk

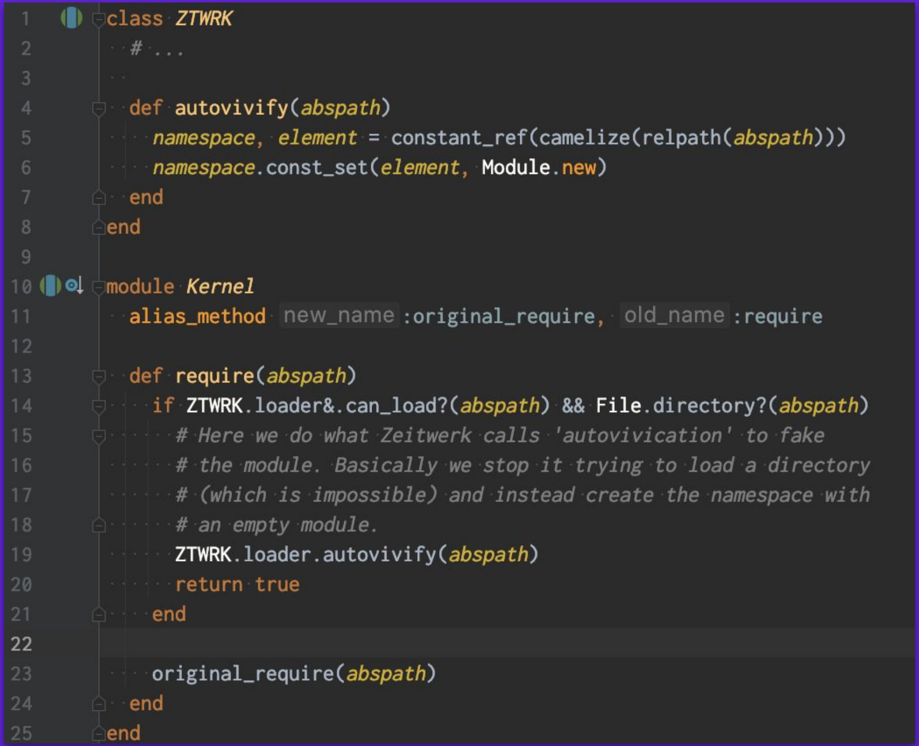


```ruby
class ZTWRK
  # ...

  def load_dir(dir)
    # First load all Ruby files. We have to do this first as these
    # take precedence for creating namespaces.
    ruby_files(dir).each do |relpath, abspath|
      namespace, element = constant_ref(
        camelize(relpath.sub(/\.rb$/, ''))
      )
      namespace.autoload(element, abspath)
    end

    # Then load all directories, and recurse into them.
    subdirectories(dir).each do |relpath, abspath|
      namespace, element = constant_ref(camelize(relpath))

      # See the Kernel patch and the autovivify method -
      # this doesn't _actually_ load the subdir.
      unless namespace.const_defined?(element)
        namespace.autoload(element, abspath)
      end

      load_dir(abspath)
    end
  end
end
```

A simple example

- Finally we add “autovivification”
- This allows the loader to create module namespaces for subdirectories without actually loading anything

https://github.com/benkyriakou/ztwrk



```ruby
class ZTWRK
  # ...
  
  def autovivify(abspath)
    namespace, element = constant_ref(camelize(relpath(abspath)))
    namespace.const_set(element, Module.new)
  end
end

module Kernel
  alias_method new_name :original_require, old_name :require

  def require(abspath)
    if ZTWRK.loader&.can_load?(abspath) && File.directory?(abspath)
      # Here we do what Zeitwerk calls 'autovivication' to fake
      # the module. Basically we stop it trying to load a directory
      # (which is impossible) and instead create the namespace with
      # an empty module.
      ZTWRK.loader.autovivify(abspath)
      return true
    end

    original_require(abspath)
  end
end
```

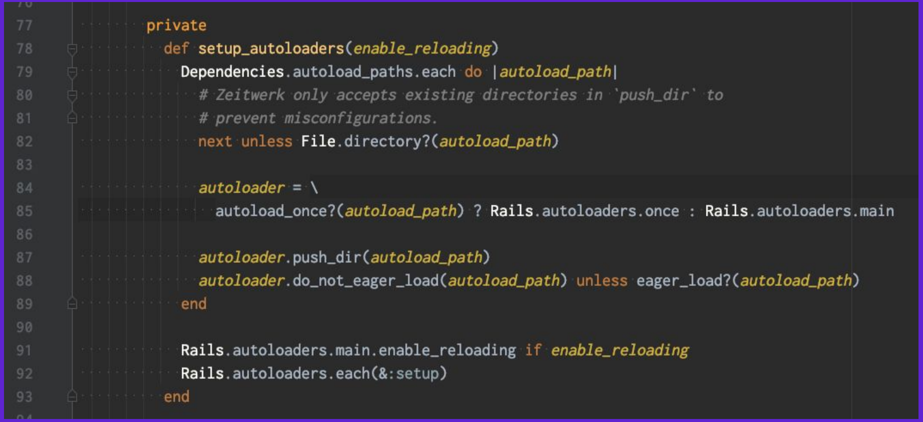
Zeitwerk in Rails

GOCARDLESS

Zeitwerk in Rails

- Set up your autoload configuration in `application.rb` using `config.autoload_paths`
- This is passed to the autoloaders in `zeitwerk_integration.rb`
- This populates the autoloaders from the Rails configuration defined
- There are two autoloaders set up
  - `Rails.autoloaders.main`
  - `Rails.autoloaders.once`

https://guides.rubyonrails.org/autoloading_and_reloading_constants.html

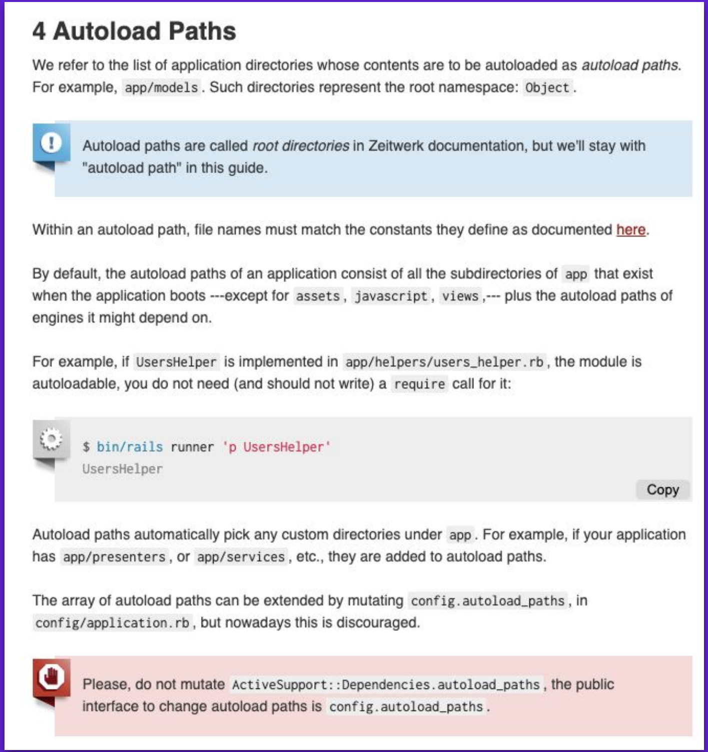


```ruby
    private
      def setup_autoloaders(enable_reloading)
        Dependencies.autoload_paths.each do |autoload_path|
          # Zeitwerk only accepts existing directories in `push_dir` to
          # prevent misconfigurations.
          next unless File.directory?(autoload_path)

          autoloader = \
            autoload_once?(autoload_path) ? Rails.autoloaders.once : Rails.autoloaders.main

          autoloader.push_dir(autoload_path)
          autoloader.do_not_eager_load(autoload_path) unless eager_load?(autoload_path)
        end

        Rails.autoloaders.main.enable_reloading if enable_reloading
        Rails.autoloaders.each(&:setup)
      end
```

Zeitwerk in Rails

- By default it adds all subdirectories of `/app` to `autoload_paths`
- This does not work with most of our namespaces, so we have to do some custom configuration
- We also have to add `/lib` as this has been retired as a standard load path in Rails

https://guides.rubyonrails.org/autoloading_and_reloading_constants.html



4 Autoload Paths

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

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`, `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
```

Copy

Autoload paths automatically pick any custom directories under `app`. For example, if your application has `app/presenters`, or `app/services`, etc., they are added to autoload paths.

The array of autoload paths can be extended by mutating `config.autoload_paths`, in `config/application.rb`, but nowadays this is discouraged.

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

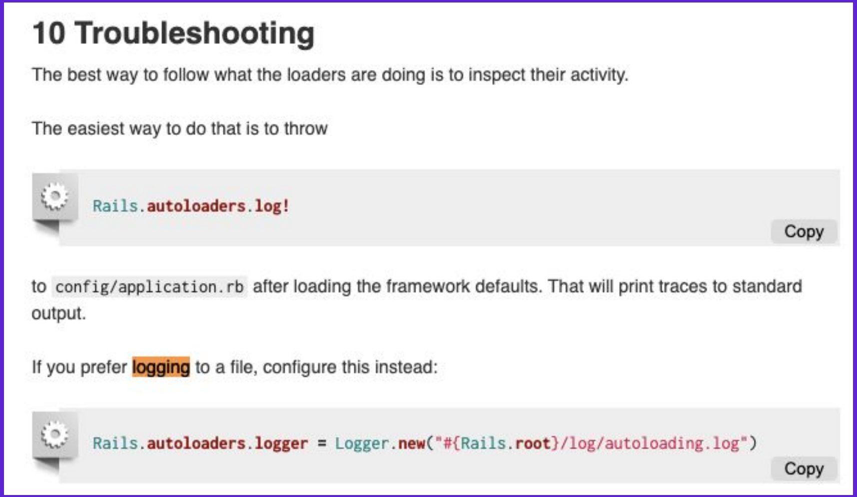
Useful things to know

GOCARDLESS

Useful things to know

- Loading in files in a non-Zeitwerk order (e.g. requiring something during bootstrap) can break the expected autoloading
  - e.g. if `Foo::Bar` is loaded during bootstrap, `Foo` can no longer be autoloaded as the namespace is taken
- You can enable logging in `application.rb`
  - This outputs a lot of debugging information about what is loaded

https://guides.rubyonrails.org/autoloading_and_reloading_constants.html



10 Troubleshooting

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

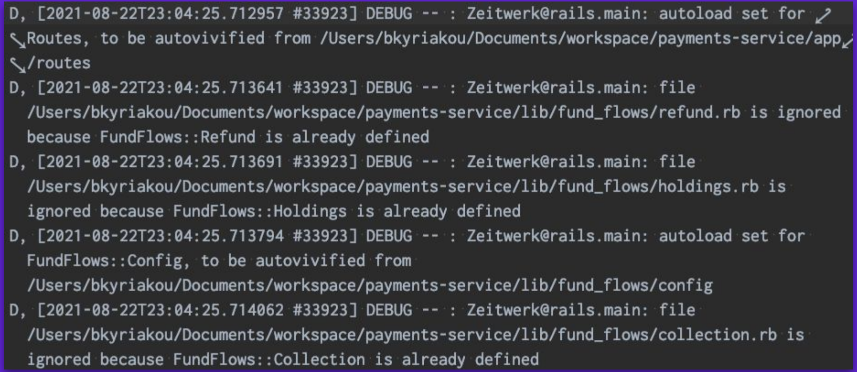
The easiest way to do that is to throw

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

to `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")
```

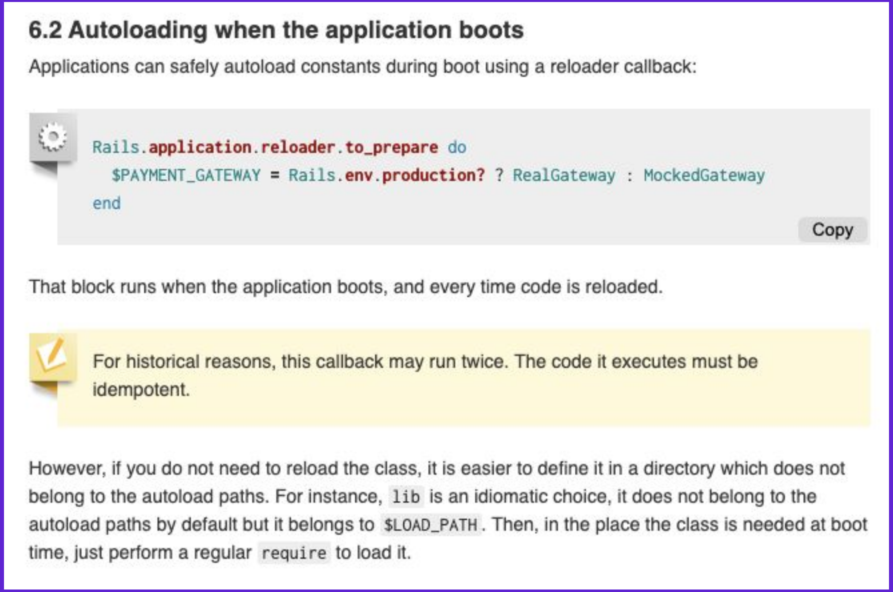


```
D, [2021-08-22T23:04:25.712957 #33923] DEBUG -- : Zeitwerk@rails.main: autoload set for
Routes, to be autovivified from /Users/bkyriakou/Documents/workspace/payments-service/app
/routes
D, [2021-08-22T23:04:25.713641 #33923] DEBUG -- : Zeitwerk@rails.main: file
  /Users/bkyriakou/Documents/workspace/payments-service/lib/fund_flows/refund.rb is ignored
  because FundFlows::Refund is already defined
D, [2021-08-22T23:04:25.713691 #33923] DEBUG -- : Zeitwerk@rails.main: file
  /Users/bkyriakou/Documents/workspace/payments-service/lib/fund_flows/holdings.rb is
  ignored because FundFlows::Holdings is already defined
D, [2021-08-22T23:04:25.713794 #33923] DEBUG -- : Zeitwerk@rails.main: autoload set for
  FundFlows::Config, to be autovivified from
  /Users/bkyriakou/Documents/workspace/payments-service/lib/fund_flows/config
D, [2021-08-22T23:04:25.714062 #33923] DEBUG -- : Zeitwerk@rails.main: file
  /Users/bkyriakou/Documents/workspace/payments-service/lib/fund_flows/collection.rb is
  ignored because FundFlows::Collection is already defined
```

Useful things to know

- If your code needs to reload with the application, or if you want to autoload constants during initialization, you should use the application reloader
- We use this in some of our initializers already
- This has proved more reliable than the Zeitwerk autoloader's `on_load` method

https://guides.rubyonrails.org/autoloading_and_reloading_constants.html



6.2 Autoloading when the application boots

Applications can safely autoload constants during boot using a reloader callback:

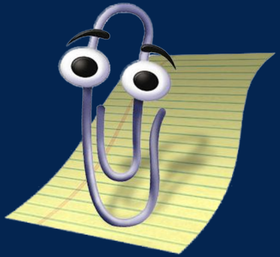
```ruby
Rails.application.reloader.to_prepare do
  $PAYMENT_GATEWAY = Rails.env.production? ? RealGateway : MockedGateway
end
```

That block runs when the application boots, and every time code is reloaded.

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

However, if you do not need to reload the class, it is easier to define it 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 belongs to `$LOAD_PATH`. Then, in the place the class is needed at boot time, just perform a regular `require` to load it.

Questions?



GOCARDLESS