Constant Autoloading and Reloading

January 13, 2015

This guide documents how constant autoloading and reloading works.
After reading this guide, you will know:

- Key aspects of Ruby constants
- What is `autoload_paths`
- How constant autoloading works
- What is `require_dependency`
- How constant reloading works
- Solutions to common autoloading gotchas

1 Introduction

Ruby on Rails allows applications to be written as if their code was preloaded.

In a normal Ruby program classes need to load their dependencies:

```
require 'application_controller'
require 'post'

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

Our Rubyist instinct quickly sees some redundancy in there: If classes were defined in files matching their name, couldn't their loading be automated somehow? We could save scanning the file for dependencies, which is brittle.

Moreover, `Kernel#require` loads files once, but development is much more smooth if code gets refreshed when it changes without restarting the server. It would be nice to be able to use `Kernel#load` in development, and `Kernel#require` in production.

Indeed, those features are provided by Ruby on Rails, where we just write

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

```
  end
end
```

This guide documents how that works.

2 Constants Refresher

While constants are trivial in most programming languages, they are a rich topic in Ruby.

It is beyond the scope of this guide to document Ruby constants, but we are nevertheless going to highlight a few key topics. Truly grasping the following sections is instrumental to understanding constant autoloading and reloading.

2.1 Nesting

Class and module definitions can be nested to create namespaces:

```
module XML
  class SAXParser
    # (1)
  end
end
```

The *nesting* at any given place is the collection of enclosing nested class and module objects outwards. For example, in the previous example, the nesting at (1) is

```
[XML::SAXParser, XML]
```

It is important to understand that the nesting is composed of class and module *objects*, it has nothing to do with the constants used to access them, and is also unrelated to their names.

For instance, while this definition is similar to the previous one:

```
class XML::SAXParser
  # (2)
end
```

the nesting in (2) is different:

```
[XML::SAXParser]
```

`XML` does not belong to it.

We can see in this example that the name of a class or module that belongs to a certain nesting does not necessarily correlate with the namespaces at the spot.

Even more, they are totally independent, take for instance

```
module X::Y
  module A::B
    # (3)
  end
end
```

The nesting in (3) consists of two module objects:

```
[A::B, X::Y]
```

So, it not only doesn't end in `A`, which does not even belong to the nesting, but it also contains `X::Y`, which is independent from `A::B`.

The nesting is an internal stack maintained by the interpreter, and it gets modified according to these rules:

- The class object following a `class` keyword gets pushed when its body is executed, and popped after it.
- The module object following a `module` keyword gets pushed when its body is executed, and popped after it.
- A singleton class opened with `class << object` gets pushed, and popped later.
- When any of the `*_eval` family of methods is called using a string argument, the singleton class of the receiver is pushed to the nesting of the eval'ed code.
- The nesting at the top-level of code interpreted by `Kernel#load` is empty unless the `load` call receives a true value as second argument, in which case a newly created anonymous module is pushed by Ruby.

It is interesting to observe that blocks do not modify the stack. In particular the blocks that may be passed to `Class.new` and `Module.new` do not get the class or module being defined pushed to their nesting. That's one of the differences between defining classes and modules in one way or another.

The nesting at any given place can be inspected with `Module.nesting`.

2.2 Class and Module Definitions are Constant Assignments

Let's suppose the following snippet creates a class (rather than reopening it):

```
class C
end
```

Ruby creates a constant `C` in `Object` and stores in that constant a class object. The name of the class instance is "C", a string, named after the constant.

That is,

```
class Project < ActiveRecord::Base
end
```

performs a constant assignment equivalent to

```
Project = Class.new(ActiveRecord::Base)
```

including setting the name of the class as a side-effect:

```
Project.name # => "Project"
```

Constant assignment has a special rule to make that happen: if the object being assigned is an anonymous class or module, Ruby sets its name to be the one the constant.

From then on, what happens to the constant and the instance does not matter. For example, the constant could be deleted, the class object could be assigned to a different constant, be stored in no constant anymore, etc. Once the name is set, it doesn't change.

Similarly, module creation using the `module` keyword as in

```
module Admin
end
```

performs a constant assignment equivalent to

```
Admin = Module.new
```

including setting the name as a side-effect:

```
Admin.name # => "Admin"
```

The execution context of a block passed to `Class.new` or `Module.new` is not entirely equivalent to the one of the body of the definitions using the `class` and `module` keywords. But both idioms result in the same constant assignment.

Thus, when one informally says "the `String` class", that really means: the class object stored in the constant called"String" in the class object stored in the `Object` constant. `String` is otherwise an ordinary Ruby constant and everything related to constants applies to it, resolution algorithms, etc.

Likewise, in the controller

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

`Post` is not syntax for a class. Rather, `Post` is a regular Ruby constant. If all is good, the constant evaluates to an object that responds to `all`.

That is why we talk about *constant* autoloading, Rails has the ability to load constants on the fly.

2.3 Constants are Stored in Modules

Constants belong to modules in a very literal sense. Classes and modules have a constant table; think of it as a hash table.

Let's analyze an example to really understand what that means. While common abuses of language like "the `String` class" are convenient, the exposition is going to be precise here for didactic purposes.

Let's consider the following module definition:

```
module Colors
  RED = '0xff0000'
end
```

First, when the `module` keyword is processed the interpreter creates a new entry in the constant table of the class object stored in the `Object` constant. Said entry associates the name "Colors" to a newly created module object. Furthermore, the interpreter sets the name of the new module object to be the string "Colors".

Later, when the body of the module definition is interpreted, a new entry is created in the constant table of the module object stored in the `Colors` constant. That entry maps the name "RED" to the string "0xff0000".

In particular, `Colors::RED` is totally unrelated to any other `RED` constant that may live in any other class or module object. If there were any, they would have separate entries in their respective constant tables.

Put special attention in the previous paragraphs to the distinction between class and module objects, constant names, and value objects associated to them in constant tables.

2.4 Resolution Algorithms

2.4.1 Resolution Algorithm for Relative Constants At any given place in the code, let's define *cref* to be the first element of the nesting if it is not empty, or `Object` otherwise.

Without getting too much into the details, the resolution algorithm for relative constant references goes like this:

1. If the nesting is not empty the constant is looked up in its elements and in order. The ancestors of those elements are ignored.
2. If not found, then the algorithm walks up the ancestor chain of the cref.
3. If not found, `const_missing` is invoked on the cref. The default implementation of `const_missing` raises `NameError`, but it can be overridden.

Rails autoloading **does not emulate this algorithm**, but its starting point is the name of the constant to be autoloaded, and the cref. See more in Relative References.

2.4.2 Resolution Algorithm for Qualified Constants Qualified constants look like this:

```
Billing::Invoice
```

`Billing::Invoice` is composed of two constants: `Billing` is relative and is resolved using the algorithm of the previous section.

Leading colons would make the first segment absolute rather than relative: `::Billing::Invoice`. That would force `Billing` to be looked up only as a top-level constant.

`Invoice` on the other hand is qualified by `Billing` and we are going to see its resolution next. Let's call *parent* to that qualifying class or module object, that is, `Billing` in the example above. The algorithm for qualified constants goes like this:

1. The constant is looked up in the parent and its ancestors.
2. If the lookup fails, `const_missing` is invoked in the parent. The default implementation of `const_missing` raises `NameError`, but it can be overridden.

As you see, this algorithm is simpler than the one for relative constants. In particular, the nesting plays no role here, and modules are not special-cased, if neither they nor their ancestors have the constants, `Object` is **not** checked.

Rails autoloading **does not emulate this algorithm**, but its starting point is the name of the constant to be autoloaded, and the parent. See more in Qualified References.

3 Vocabulary

3.1 Parent Namespaces

Given a string with a constant path we define its *parent namespace* to be the string that results from removing its rightmost segment.

For example, the parent namespace of the string "A::B::C" is the string "A::B", the parent namespace of "A::B" is "A", and the parent namespace of "A" is "".

The interpretation of a parent namespace when thinking about classes and modules is tricky though. Let's consider a module M named "A::B":

- The parent namespace, "A", may not reflect nesting at a given spot.
- The constant `A` may no longer exist, some code could have removed it from `Object`.
- If `A` exists, the class or module that was originally in `A` may not be there anymore. For example, if after a constant removal there was another constant assignment there would generally be a different object in there.
- In such case, it could even happen that the reassigned `A` held a new class or module called also "A"!
- In the previous scenarios M would no longer be reachable through `A::B` but the module object itself could still be alive somewhere and its name would still be "A::B".

The idea of a parent namespace is at the core of the autoloading algorithms and helps explain and understand their motivation intuitively, but as you see that metaphor leaks easily. Given an edge case to reason about, take always into account that by "parent namespace" the guide means exactly that specific string derivation.

3.2 Loading Mechanism

Rails autoloads files with `Kernel#load` when `config.cache_classes` is false, the default in development mode, and with `Kernel#require` otherwise, the default in production mode.

`Kernel#load` allows Rails to execute files more than once if constant reloading is enabled.

This guide uses the word "load" freely to mean a given file is interpreted, but the actual mechanism can be `Kernel#load` or `Kernel#require` depending on that flag.

4 Autoloading Availability

Rails is always able to autoload provided its environment is in place. For example the `runner` command autoloads:

```
$ bin/rails runner 'p User.column_names'
["id", "email", "created_at", "updated_at"]
```

The console autoloads, the test suite autoloads, and of course the application autoloads.

By default, Rails eager loads the application files when it boots in production mode, so most of the autoloading going on in development does not happen. But autoloading may still be triggered during eager loading.

For example, given

```
class BeachHouse < House
end
```

if `House` is still unknown when `app/models/beach_house.rb` is being eager loaded, Rails autoloads it.

5 autoload_paths

As you probably know, when `require` gets a relative file name:

```
require 'erb'
```

Ruby looks for the file in the directories listed in `$LOAD_PATH`. That is, Ruby iterates over all its directories and for each one of them checks whether they have a file called "erb.rb", or "erb.so", or "erb.o", or "erb.dll". If it finds any of them, the interpreter loads it and ends the search. Otherwise, it tries again in the next directory of the list. If the list gets exhausted, `LoadError` is raised.

We are going to cover how constant autoloading works in more detail later, but the idea is that when a constant like `Post` is hit and missing, if there's a `post.rb` file for example in `app/models` Rails is going to find it, evaluate it, and have `Post` defined as a side-effect.

Alright, Rails has a collection of directories similar to `$LOAD_PATH` in which to look up `post.rb`. That collection is called `autoload_paths` and by default it contains:

- All subdirectories of `app` in the application and engines. For example, `app/controllers`. They do not need to be the default ones, any custom directories like `app/workers` belong automatically to `autoload_paths`.
- Any existing second level directories called `app/*/concerns` in the application and engines.
- The directory `test/mailers/previews`.

Also, this collection is configurable via `config.autoload_paths`. For example, `lib` was in the list years ago, but no longer is. An application can opt-in throwing this to `config/application.rb`:

```
config.autoload_paths += "#{Rails.root}/lib"
```

The value of `autoload_paths` can be inspected. In a just generated application it is (edited):

```
$ bin/rails r 'puts ActiveSupport::Dependencies.autoload_paths'
.../app/assets
.../app/controllers
.../app/helpers
.../app/mailers
.../app/models
.../app/controllers/concerns
.../app/models/concerns
.../test/mailers/previews
```

`autoload_paths` is computed and cached during the initialization process. The application needs to be restarted to reflect any changes in the directory structure.

6 Autoloading Algorithms

6.1 Relative References

A relative constant reference may appear in several places, for example, in

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

all three constant references are relative.

6.1.1 Constants after the `class` and `module` Keywords Ruby performs a lookup for the constant that follows a `class` or `module` keyword because it needs to know if the class or module is going to be created or reopened.

If the constant is not defined at that point it is not considered to be a missing constant, autoloading is **not** triggered.

So, in the previous example, if `PostsController` is not defined when the file is interpreted Rails autoloading is not going to be triggered, Ruby will just define the controller.

6.1.2 Top-Level Constants On the contrary, if `ApplicationController` is unknown, the constant is considered missing and an autoload is going to be attempted by Rails.

In order to load `ApplicationController`, Rails iterates over `autoload_paths`. First checks if `app/assets/application_controller.rb` exists. If it does not, which is normally the case, it continues and finds `app/controllers/application_controller.rb`.

If the file defines the constant `ApplicationController` all is fine, otherwise `LoadError` is raised:

```
unable to autoload constant ApplicationController, expected
<full path to application_controller.rb> to define it (LoadError)
```

Rails does not require the value of autoloaded constants to be a class or module object. For example, if the file `app/models/max_clients.rb` defines `MAX_CLIENTS = 100` autoloading `MAX_CLIENTS` works just fine.

6.1.3 Namespaces Autoloading `ApplicationController` looks directly under the directories of `autoload_paths` because the nesting in that spot is empty. The situation of `Post` is different, the nesting in that line is `[PostsController]` and support for namespaces comes into play.

The basic idea is that given

```
module Admin
  class BaseController < ApplicationController
    @@all_roles = Role.all
  end
end
```

to autoload `Role` we are going to check if it is defined in the current or parent namespaces, one at a time. So, conceptually we want to try to autoload any of

```
Admin::BaseController::Role
Admin::Role
Role
```

in that order. That's the idea. To do so, Rails looks in `autoload_paths` respectively for file names like these:

```
admin/base_controller/role.rb
admin/role.rb
role.rb
```

modulus some additional directory lookups we are going to cover soon.

`'Constant::Name'.underscore` gives the relative path without extension of the file name where `Constant::Name` is expected to be defined.

Let's see how Rails autoloads the `Post` constant in the `PostsController` above assuming the application has a `Post` model defined in `app/models/post.rb`.

First it checks for `posts_controller/post.rb` in `autoload_paths`:

```
app/assets/posts_controller/post.rb
app/controllers/posts_controller/post.rb
app/helpers/posts_controller/post.rb
...
test/mailers/previews/posts_controller/post.rb
```

Since the lookup is exhausted without success, a similar search for a directory is performed, we are going to see why in the next section:

```
app/assets/posts_controller/post
app/controllers/posts_controller/post
app/helpers/posts_controller/post
...
test/mailers/previews/posts_controller/post
```

If all those attempts fail, then Rails starts the lookup again in the parent namespace. In this case only the top-level remains:

```
app/assets/post.rb
app/controllers/post.rb
app/helpers/post.rb
app/mailers/post.rb
app/models/post.rb
```

A matching file is found in `app/models/post.rb`. The lookup stops there and the file is loaded. If the file actually defines `Post` all is fine, otherwise `LoadError` is raised.

6.2 Qualified References

When a qualified constant is missing Rails does not look for it in the parent namespaces. But there is a caveat: When a constant is missing, Rails is unable to tell if the trigger was a relative reference or a qualified one.

For example, consider

```
module Admin
  User
end
```

and

```
Admin::User
```

If `User` is missing, in either case all Rails knows is that a constant called "User" was missing in a module called "Admin".

If there is a top-level `User` Ruby would resolve it in the former example, but wouldn't in the latter. In general, Rails does not emulate the Ruby constant resolution algorithms, but in this case it tries using the following heuristic:

> If none of the parent namespaces of the class or module has the missing constant then Rails assumes the reference is relative. Otherwise qualified.

For example, if this code triggers autoloading

```
Admin::User
```

and the `User` constant is already present in `Object`, it is not possible that the situation is

```
module Admin
  User
end
```

because otherwise Ruby would have resolved `User` and no autoloading would have been triggered in the first place. Thus, Rails assumes a qualified reference and considers the file `admin/user.rb` and directory `admin/user` to be the only valid options.

In practice, this works quite well as long as the nesting matches all parent namespaces respectively and the constants that make the rule apply are known at that time.

However, autoloading happens on demand. If by chance the top-level `User` was not yet loaded, then Rails assumes a relative reference by contract.

Naming conflicts of this kind are rare in practice, but if one occurs, `require_dependency` provides a solution by ensuring that the constant needed to trigger the heuristic is defined in the conflicting place.

6.3 Automatic Modules

When a module acts as a namespace, Rails does not require the application to defines a file for it, a directory matching the namespace is enough.

Suppose an application has a back office whose controllers are stored in `app/controllers/admin`. If the `Admin` module is not yet loaded when `Admin::UsersController` is hit, Rails needs first to autoload the constant `Admin`.

If `autoload_paths` has a file called `admin.rb` Rails is going to load that one, but if there's no such file and a directory called `admin` is found, Rails creates an empty module and assigns it to the `Admin` constant on the fly.

6.4 Generic Procedure

Relative references are reported to be missing in the cref where they were hit, and qualified references are reported to be missing in their parent. (See Resolution Algorithm for Relative Constants at the beginning of this guide for the definition of *cref*, and Resolution Algorithm for Qualified Constants for the definition of *parent*.)

The procedure to autoload constant `C` in an arbitrary situation is as follows:

```
if the class or module in which C is missing is Object
  let ns = ''
else
  let M = the class or module in which C is missing

  if M is anonymous
    let ns = ''
  else
    let ns = M.name
  end
end

loop do
  # Look for a regular file.
  for dir in autoload_paths
    if the file "#{dir}/#{ns.underscore}/c.rb" exists
      load/require "#{dir}/#{ns.underscore}/c.rb"

      if C is now defined
        return
      else
        raise LoadError
      end
    end
  end

  # Look for an automatic module.
  for dir in autoload_paths
```

```
    if the directory "#{dir}/#{ns.underscore}/c" exists
      if ns is an empty string
        let C = Module.new in Object and return
      else
        let C = Module.new in ns.constantize and return
      end
    end
  end

  if ns is empty
    # We reached the top-level without finding the constant.
    raise NameError
  else
    if C exists in any of the parent namespaces
      # Qualified constants heuristic.
      raise NameError
    else
      # Try again in the parent namespace.
      let ns = the parent namespace of ns and retry
    end
  end
end
```

7 require_dependency

Constant autoloading is triggered on demand and therefore code that uses a certain constant may have it already defined or may trigger an autoload. That depends on the execution path and it may vary between runs.

There are times, however, in which you want to make sure a certain constant is known when the execution reaches some code. `require_dependency` provides a way to load a file using the current loading mechanism, and keeping track of constants defined in that file as if they were autoloaded to have them reloaded as needed.

`require_dependency` is rarely needed, but see a couple of use-cases in Autoloading and STI and When Constants aren't Triggered.

Unlike autoloading, `require_dependency` does not expect the file to define any particular constant. Exploiting this behavior would be a bad practice though, file and constant paths should match.

8 Constant Reloading

When `config.cache_classes` is false Rails is able to reload autoloaded constants.

For example, in you're in a console session and edit some file behind the scenes, the code can be reloaded with the `reload!` command:

```
> reload!
```

When the application runs, code is reloaded when something relevant to this logic changes. In order to do that, Rails monitors a number of things:

- `config/routes.rb`.
- Locales.
- Ruby files under `autoload_paths`.
- `db/schema.rb` and `db/structure.sql`.

If anything in there changes, there is a middleware that detects it and reloads the code.

Autoloading keeps track of autoloaded constants. Reloading is implemented by removing them all from their respective classes and modules using `Module#remove_const`. That way, when the code goes on, those constants are going to be unknown again, and files reloaded on demand.

This is an all-or-nothing operation, Rails does not attempt to reload only what changed since dependencies between classes makes that really tricky. Instead, everything is wiped.

9 Module#autoload isn't Involved

`Module#autoload` provides a lazy way to load constants that is fully integrated with the Ruby constant lookup algorithms, dynamic constant API, etc. It is quite transparent.

Rails internals make extensive use of it to defer as much work as possible from the boot process. But constant autoloading in Rails is **not** implemented with `Module#autoload`.

One possible implementation based on `Module#autoload` would be to walk the application tree and issue `autoload` calls that map existing file names to their conventional constant name.

There are a number of reasons that prevent Rails from using that implementation.

For example, `Module#autoload` is only capable of loading files using `require`, so reloading would not be possible. Not only that, it uses an internal `require` which is not `Kernel#require`.

Then, it provides no way to remove declarations in case a file is deleted. If a constant gets removed with `Module#remove_const` its `autoload` is not triggered again. Also, it doesn't support qualified names, so files with namespaces should be interpreted during the walk tree to install their own `autoload` calls, but those files could have constant references not yet configured.

An implementation based on `Module#autoload` would be awesome but, as you see, at least as of today it is not possible. Constant autoloading in Rails is implemented with `Module#const_missing`, and that's why it has its own contract, documented in this guide.

10 Common Gotchas

10.1 Nesting and Qualified Constants

Let's consider

```
module Admin
  class UsersController < ApplicationController
    def index
      @users = User.all
    end
  end
end
```

and

```
class Admin::UsersController < ApplicationController
  def index
    @users = User.all
  end
end
```

To resolve `User` Ruby checks `Admin` in the former case, but it does not in the latter because it does not belong to the nesting. (See Nesting and Resolution Algorithms.)

Unfortunately Rails autoloading does not know the nesting in the spot where the constant was missing and so it is not able to act as Ruby would. In particular, `Admin::User` will get autoloaded in either case.

Albeit qualified constants with `class` and `module` keywords may technically work with autoloading in some cases, it is preferable to use relative constants instead:

```
module Admin
  class UsersController < ApplicationController
    def index
      @users = User.all
    end
  end
end
```

10.2 Autoloading and STI

Single Table Inheritance (STI) is a feature of Active Record that easies storing a hierarchy of models in one single table. The API of such models is aware of the hierarchy and encapsulates some common needs. For example, given these classes:

```
# app/models/polygon.rb
class Polygon < ActiveRecord::Base
end

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

# app/models/rectangle.rb
class Rectangle < Polygon
end
```

`Triangle.create` creates a row that represents a triangle, and `Rectangle.create` creates a row that represents a rectangle. If `id` is the ID of an existing record, `Polygon.find(id)` returns an object of the correct type.

Methods that operate on collections are also aware of the hierarchy. For example, `Polygon.all` returns all the records of the table, because all rectangles and triangles are polygons. Active Record takes care of returning instances of their corresponding class in the result set.

Types are autoloaded as needed. For example, if `Polygon.first` is a rectangle and `Rectangle` has not yet been loaded, Active Record autoloads it and the record is correctly instantiated.

All good, but if instead of performing queries based on the root class we need to work on some subclass, things get interesting.

While working with `Polygon` you do not need to be aware of all its descendants, because anything in the table is by definition a polygon, but when working with subclasses Active Record needs to be able to enumerate the types it is looking for. Let's see an example.

`Rectangle.all` only loads rectangles by adding a type constraint to the query:

```
SELECT "polygons".* FROM "polygons"
WHERE "polygons"."type" IN ("Rectangle")
```

Let's introduce now a subclass of `Rectangle`:

```
# app/models/square.rb
class Square < Rectangle
end
```

`Rectangle.all` should now return rectangles **and** squares:

```
SELECT "polygons".* FROM "polygons"
WHERE "polygons"."type" IN ("Rectangle", "Square")
```

But there's a caveat here: How does Active Record know that the class `Square` exists at all?

Even if the file `app/models/square.rb` exists and defines the `Square` class, if no code yet used that class, `Rectangle.all` issues the query

```
SELECT "polygons".* FROM "polygons"
WHERE "polygons"."type" IN ("Rectangle")
```

That is not a bug, the query includes all *known* descendants of `Rectangle`.

A way to ensure this works correctly regardless of the order of execution is to load the leaves of the tree by hand at the bottom of the file that defines the root class:

```
# app/models/polygon.rb
class Polygon < ActiveRecord::Base
end
require_dependency 'square'
```

Only the leaves that are **at least grandchildren** need to be loaded this way. Direct subclasses do not need to be preloaded. If the hierarchy is deeper, intermediate classes will be autoloaded recursively from the bottom because their constant will appear in the class definitions as superclass.

10.3 Autoloading and `require`

Files defining constants to be autoloaded should never be `required`:

```
require 'user' # DO NOT DO THIS

class UsersController < ApplicationController
  ...
end
```

There are two possible gotchas here in development mode:

1. If `User` is autoloaded before reaching the `require`, `app/models/user.rb` runs again because `load` does not update `$LOADED_FEATURES`.
2. If the `require` runs first Rails does not mark `User` as an autoloaded constant and changes to `app/models/user.rb` aren't reloaded.

Just follow the flow and use constant autoloading always, never mix autoloading and `require`. As a last resort, if some file absolutely needs to load a certain file use `require_dependency` to play nice with constant autoloading. This option is rarely needed in practice, though.

Of course, using `require` in autoloaded files to load ordinary 3rd party libraries is fine, and Rails is able to distinguish their constants, they are not marked as autoloaded.

10.4 Autoloading and Initializers

Consider this assignment in `config/initializers/set_auth_service.rb`:

```
AUTH_SERVICE = if Rails.env.production?
  RealAuthService
else
  MockedAuthService
end
```

The purpose of this setup would be that the application uses the class that corresponds to the environment via `AUTH_SERVICE`. In development mode `MockedAuthService` gets autoloaded when the initializer runs. Let's suppose we do some requests, change its implementation, and hit the application again. To our surprise the changes are not reflected. Why?

As we saw earlier, Rails removes autoloaded constants, but `AUTH_SERVICE` stores the original class object. Stale, non-reachable using the original constant, but perfectly functional.

The following code summarizes the situation:

```
class C
  def quack
    'quack!'
  end
end
```

```
X = C
Object.instance_eval { remove_const(:C) }
X.new.quack # => quack!
X.name      # => C
C           # => uninitialized constant C (NameError)
```

Because of that, it is not a good idea to autoload constants on application initialization.

In the case above we could implement a dynamic access point:

```
# app/models/auth_service.rb
class AuthService
  if Rails.env.production?
    def self.instance
      RealAuthService
    end
  else
    def self.instance
      MockedAuthService
    end
  end
end
```

and have the application use `AuthService.instance` instead. `AuthService` would be loaded on demand and be autoload-friendly.

10.5 `require_dependency` and Initializers

As we saw before, `require_dependency` loads files in an autoloading-friendly way. Normally, though, such a call does not make sense in an initializer.

One could think about doing some `require_dependency` calls in an initializer to make sure certain constants are loaded upfront, for example as an attempt to address the gotcha with STIs.

Problem is, in development mode autoloaded constants are wiped if there is any relevant change in the file system. If that happens then we are in the very same situation the initializer wanted to avoid!

Calls to `require_dependency` have to be strategically written in autoloaded spots.

10.6 When Constants aren't Missed

10.6.1 Relative References Let's consider a flight simulator. The application has a default flight model

```
# app/models/flight_model.rb
class FlightModel
end
```

that can be overridden by each airplane, for instance

```
# app/models/bell_x1/flight_model.rb
module BellX1
```

```
  class FlightModel < FlightModel
  end
end

# app/models/bell_x1/aircraft.rb
module BellX1
  class Aircraft
    def initialize
      @flight_model = FlightModel.new
    end
  end
end
```

The initializer wants to create a `BellX1::FlightModel` and nesting has `BellX1`, that looks good. But if the default flight model is loaded and the one for the Bell-X1 is not, the interpreter is able to resolve the top-level `FlightModel` and autoloading is thus not triggered for `BellX1::FlightModel`.

That code depends on the execution path.

These kind of ambiguities can often be resolved using qualified constants:

```
module BellX1
  class Plane
    def flight_model
      @flight_model ||= BellX1::FlightModel.new
    end
  end
end
```

Also, `require_dependency` is a solution:

```
require_dependency 'bell_x1/flight_model'

module BellX1
  class Plane
    def flight_model
      @flight_model ||= FlightModel.new
    end
  end
end
```

10.6.2 Qualified References

Given

```
# app/models/hotel.rb
class Hotel
end

# app/models/image.rb
class Image
```

```
end

# app/models/hotel/image.rb
class Hotel
  class Image < Image
  end
end
```

the expression `Hotel::Image` is ambiguous, depends on the execution path.

As we saw before, Ruby looks up the constant in `Hotel` and its ancestors. If `app/models/image.rb` has been loaded but `app/models/hotel/image.rb` hasn't, Ruby does not find `Image` in `Hotel`, but it does in `Object`:

```
$ bin/rails r 'Image; p Hotel::Image' 2>/dev/null
Image # NOT Hotel::Image!
```

The code evaluating `Hotel::Image` needs to make sure `app/models/hotel/image.rb` has been loaded, possibly with `require_dependency`.

In these cases the interpreter issues a warning though:

```
warning: toplevel constant Image referenced by Hotel::Image
```

This surprising constant resolution can be observed with any qualifying class:

```
2.1.5 :001 > String::Array
(irb):1: warning: toplevel constant Array referenced by String::Array
 => Array
```

To find this gotcha the qualifying namespace has to be a class, `Object` is not an ancestor of modules.

10.7 Autoloading within Singleton Classes

Let's suppose we have these class definitions:

```
# app/models/hotel/services.rb
module Hotel
  class Services
  end
end

# app/models/hotel/geo_location.rb
module Hotel
  class GeoLocation
    class << self
      Services
    end
  end
end
```

If `Hotel::Services` is known by the time `app/models/hotel/geo_location.rb` is being loaded, `Services` is resolved by Ruby because `Hotel` belongs to the nesting when the singleton class of `Hotel::GeoLocation` is opened.

But if `Hotel::Services` is not known, Rails is not able to autoload it, the application raises `NameError`.

The reason is that autoloading is triggered for the singleton class, which is anonymous, and as we saw before, Rails only checks the top-level namespace in that edge case.

An easy solution to this caveat is to qualify the constant:

```
module Hotel
  class GeoLocation
    class << self
      Hotel::Services
    end
  end
end
```

10.8 Autoloading in `BasicObject`

Direct descendants of `BasicObject` do not have `Object` among their ancestors and cannot resolve top-level constants:

```
class C < BasicObject
  String # NameError: uninitialized constant C::String
end
```

When autoloading is involved that plot has a twist. Let's consider:

```
class C < BasicObject
  def user
    User # WRONG
  end
end
```

Since Rails checks the top-level namespace `User` gets autoloaded just fine the first time the `user` method is invoked. You only get the exception if the `User` constant is known at that point, in particular in a *second* call to `user`:

```
c = C.new
c.user # surprisingly fine, User
c.user # NameError: uninitialized constant C::User
```

because it detects a parent namespace already has the constant (see Qualified References.)

As with pure Ruby, within the body of a direct descendant of `BasicObject` use always absolute constant paths:

```
class C < BasicObject
  ::String # RIGHT
```

```
  def user
    ::User # RIGHT
  end
end
```

11 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 master. Make sure to check Edge Guides first to verify if the issues are already fixed or not on the master 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 in the rubyonrails-docs mailing list.