Configuring Rails Applications

January 13, 2015

This guide covers the configuration and initialization features available to Rails applications.
After reading this guide, you will know:

- How to adjust the behavior of your Rails applications.
- How to add additional code to be run at application start time.

1 Locations for Initialization Code

Rails offers four standard spots to place initialization code:

- `config/application.rb`
- Environment-specific configuration files
- Initializers
- After-initializers

2 Running Code Before Rails

In the rare event that your application needs to run some code before Rails itself is loaded, put it above the call to `require 'rails/all'` in `config/application.rb`.

3 Configuring Rails Components

In general, the work of configuring Rails means configuring the components of Rails, as well as configuring Rails itself. The configuration file `config/application.rb` and environment-specific configuration files (such as `config/environments/production.rb`) allow you to specify the various settings that you want to pass down to all of the components.

For example, the `config/application.rb` file includes this setting:

```
config.autoload_paths += %W(#{config.root}/extras)
```

This is a setting for Rails itself. If you want to pass settings to individual Rails components, you can do so via the same `config` object in `config/application.rb`:

```
config.active_record.schema_format = :ruby
```

Rails will use that particular setting to configure Active Record.

3.1 Rails General Configuration

These configuration methods are to be called on a `Rails::Railtie` object, such as a subclass of `Rails::Engine` or `Rails::Application`.

- `config.after_initialize` takes a block which will be run *after* Rails has finished initializing the application. That includes the initialization of the framework itself, engines, and all the application's initializers in `config/initializers`. Note that this block *will* be run for rake tasks. Useful for configuring values set up by other initializers:

```
config.after_initialize do
  ActionView::Base.sanitized_allowed_tags.delete 'div'
end
```

- `config.asset_host` sets the host for the assets. Useful when CDNs are used for hosting assets, or when you want to work around the concurrency constraints built-in in browsers using different domain aliases. Shorter version of `config.action_controller.asset_host`.

- `config.autoload_once_paths` accepts an array of paths from which Rails will autoload constants that won't be wiped per request. Relevant if `config.cache_classes` is false, which is the case in development mode by default. Otherwise, all autoloading happens only once. All elements of this array must also be in `autoload_paths`. Default is an empty array.

- `config.autoload_paths` accepts an array of paths from which Rails will autoload constants. Default is all directories under `app`.

- `config.cache_classes` controls whether or not application classes and modules should be reloaded on each request. Defaults to false in development mode, and true in test and production modes.

- `config.action_view.cache_template_loading` controls whether or not templates should be reloaded on each request. Defaults to whatever is set for `config.cache_classes`.

- `config.beginning_of_week` sets the default beginning of week for the application. Accepts a valid week day symbol (e.g. `:monday`).

- `config.cache_store` configures which cache store to use for Rails caching. Options include one of the symbols `:memory_store`, `:file_store`, `:mem_cache_store`, `:null_store`, or an object that implements the cache API. Defaults to `:file_store` if the directory `tmp/cache` exists, and to `:memory_store` otherwise.

- `config.colorize_logging` specifies whether or not to use ANSI color codes when logging information. Defaults to true.

- `config.consider_all_requests_local` is a flag. If true then any error will cause detailed debugging information to be dumped in the HTTP response, and the `Rails::Info` controller will show the application runtime context in `/rails/info/properties`. True by default in development and test environments, and false in production mode. For finer-grained control, set this to false and implement `local_request?` in controllers to specify which requests should provide debugging information on errors.

- `config.console` allows you to set class that will be used as console you run `rails console`. It's best to run it in `console` block:

```
console do
  # this block is called only when running console,
  # so we can safely require pry here
  require "pry"
  config.console = Pry
end
```

- `config.dependency_loading` is a flag that allows you to disable constant autoloading setting it to false. It only has effect if `config.cache_classes` is true, which it is by default in production mode.
- `config.eager_load` when true, eager loads all registered `config.eager_load_namespaces`. This includes your application, engines, Rails frameworks and any other registered namespace.
- `config.eager_load_namespaces` registers namespaces that are eager loaded when `config.eager_load` is true. All namespaces in the list must respond to the `eager_load!` method.
- `config.eager_load_paths` accepts an array of paths from which Rails will eager load on boot if cache classes is enabled. Defaults to every folder in the `app` directory of the application.
- `config.encoding` sets up the application-wide encoding. Defaults to UTF-8.
- `config.exceptions_app` sets the exceptions application invoked by the ShowException middleware when an exception happens. Defaults to `ActionDispatch::PublicExceptions.new(Rails.public_path)`.
- `config.file_watcher` the class used to detect file updates in the filesystem when `config.reload_classes_only_on_change` is true. Must conform to `ActiveSupport::FileUpdateChecker` API.
- `config.filter_parameters` used for filtering out the parameters that you don't want shown in the logs, such as passwords or credit card numbers. New applications filter out passwords by adding the following `config.filter_parameters+=[:password]` in `config/initializers/filter_parameter_logging.rb`.
- `config.force_ssl` forces all requests to be under HTTPS protocol by using `ActionDispatch::SSL` middleware.
- `config.log_formatter` defines the formatter of the Rails logger. This option defaults to an instance of `ActiveSupport::Logger::SimpleFormatter` for all modes except production, where it defaults to `Logger::Formatter`.
- `config.log_level` defines the verbosity of the Rails logger. This option defaults to `:debug` for all environments.
- `config.log_tags` accepts a list of methods that the `request` object responds to. This makes it easy to tag log lines with debug information like subdomain and request id - both very helpful in debugging multi-user production applications.

- `config.logger` accepts a logger conforming to the interface of Log4r or the default Ruby `Logger` class. Defaults to an instance of `ActiveSupport::Logger`.

- `config.middleware` allows you to configure the application's middleware. This is covered in depth in the Configuring Middleware section below.

- `config.reload_classes_only_on_change` enables or disables reloading of classes only when tracked files change. By default tracks everything on autoload paths and is set to true. If `config.cache_classes` is true, this option is ignored.

- `secrets.secret_key_base` is used for specifying a key which allows sessions for the application to be verified against a known secure key to prevent tampering. Applications get `secrets.secret_key_base` initialized to a random key present in `config/secrets.yml`.

- `config.serve_static_files` configures Rails itself to serve static files. Defaults to true, but in the production environment is turned off as the server software (e.g. NGINX or Apache) used to run the application should serve static assets instead. Unlike the default setting set this to true when running (absolutely not recommended!) or testing your app in production mode using WEBrick. Otherwise you won't be able use page caching and requests for files that exist regularly under the public directory will anyway hit your Rails app.

- `config.session_store` is usually set up in `config/initializers/session_store.rb` and specifies what class to use to store the session. Possible values are `:cookie_store` which is the default, `:mem_cache_store`, and `:disabled`. The last one tells Rails not to deal with sessions. Custom session stores can also be specified:

  ```
  config.session_store :my_custom_store
  ```

  This custom store must be defined as `ActionDispatch::Session::MyCustomStore`.

- `config.time_zone` sets the default time zone for the application and enables time zone awareness for Active Record.

3.2 Configuring Assets

- `config.assets.enabled` a flag that controls whether the asset pipeline is enabled. It is set to true by default.

- `config.assets.raise_runtime_errors` Set this flag to `true` to enable additional runtime error checking. Recommended in `config/environments/development.rb` to minimize unexpected behavior when deploying to `production`.

- `config.assets.compress` a flag that enables the compression of compiled assets. It is explicitly set to true in `config/environments/production.rb`.

- `config.assets.css_compressor` defines the CSS compressor to use. It is set by default by `sass-rails`. The unique alternative value at the moment is `:yui`, which uses the `yui-compressor` gem.

- `config.assets.js_compressor` defines the JavaScript compressor to use. Possible values are `:closure`, `:uglifier` and `:yui` which require the use of the `closure-compiler`, `uglifier` or `yui-compressor` gems respectively.
- `config.assets.paths` contains the paths which are used to look for assets. Appending paths to this configuration option will cause those paths to be used in the search for assets.
- `config.assets.precompile` allows you to specify additional assets (other than `application.css` and `application.js`) which are to be precompiled when `rake assets:precompile` is run.
- `config.assets.prefix` defines the prefix where assets are served from. Defaults to `/assets`.
- `config.assets.manifest` defines the full path to be used for the asset precompiler's manifest file. Defaults to a file named `manifest-<random>.json` in the `config.assets.prefix` directory within the public folder.
- `config.assets.digest` enables the use of MD5 fingerprints in asset names. Set to `true` by default in `production.rb` and `development.rb`.
- `config.assets.debug` disables the concatenation and compression of assets. Set to `true` by default in `development.rb`.
- `config.assets.cache_store` defines the cache store that Sprockets will use. The default is the Rails file store.
- `config.assets.version` is an option string that is used in MD5 hash generation. This can be changed to force all files to be recompiled.
- `config.assets.compile` is a boolean that can be used to turn on live Sprockets compilation in production.
- `config.assets.logger` accepts a logger conforming to the interface of Log4r or the default Ruby `Logger` class. Defaults to the same configured at `config.logger`. Setting `config.assets.logger` to false will turn off served assets logging.

3.3 Configuring Generators

Rails allows you to alter what generators are used with the `config.generators` method. This method takes a block:

```
config.generators do |g|
  g.orm :active_record
  g.test_framework :test_unit
end
```

The full set of methods that can be used in this block are as follows:

- `assets` allows to create assets on generating a scaffold. Defaults to `true`.
- `force_plural` allows pluralized model names. Defaults to `false`.
- `helper` defines whether or not to generate helpers. Defaults to `true`.

- `integration_tool` defines which integration tool to use. Defaults to `nil`.
- `javascripts` turns on the hook for JavaScript files in generators. Used in Rails for when the `scaffold` generator is run. Defaults to `true`.
- `javascript_engine` configures the engine to be used (for eg. coffee) when generating assets. Defaults to `nil`.
- `orm` defines which orm to use. Defaults to `false` and will use Active Record by default.
- `resource_controller` defines which generator to use for generating a controller when using `rails generate resource`. Defaults to `:controller`.
- `scaffold_controller` different from `resource_controller`, defines which generator to use for generating a *scaffolded* controller when using `rails generate scaffold`. Defaults to `:scaffold_controller`.
- `stylesheets` turns on the hook for stylesheets in generators. Used in Rails for when the `scaffold` generator is run, but this hook can be used in other generates as well. Defaults to `true`.
- `stylesheet_engine` configures the stylesheet engine (for eg. sass) to be used when generating assets. Defaults to `:css`.
- `test_framework` defines which test framework to use. Defaults to `false` and will use Test::Unit by default.
- `template_engine` defines which template engine to use, such as ERB or Haml. Defaults to `:erb`.

3.4 Configuring Middleware

Every Rails application comes with a standard set of middleware which it uses in this order in the development environment:

- `ActionDispatch::SSL` forces every request to be under HTTPS protocol. Will be available if `config.force_ssl` is set to `true`. Options passed to this can be configured by using `config.ssl_options`.
- `ActionDispatch::Static` is used to serve static assets. Disabled if `config.serve_static_assets` is `false`.
- `Rack::Lock` wraps the app in mutex so it can only be called by a single thread at a time. Only enabled when `config.cache_classes` is `false`.
- `ActiveSupport::Cache::Strategy::LocalCache` serves as a basic memory backed cache. This cache is not thread safe and is intended only for serving as a temporary memory cache for a single thread.
- `Rack::Runtime` sets an `X-Runtime` header, containing the time (in seconds) taken to execute the request.
- `Rails::Rack::Logger` notifies the logs that the request has begun. After request is complete, flushes all the logs.
- `ActionDispatch::ShowExceptions` rescues any exception returned by the application and renders nice exception pages if the request is local or if `config.consider_all_requests_local` is set to `true`. If `config.action_dispatch.show_exceptions` is set to `false`, exceptions will be raised regardless.
- `ActionDispatch::RequestId` makes a unique X-Request-Id header available to the response and enables the `ActionDispatch::Request#uuid` method.
- `ActionDispatch::RemoteIp` checks for IP spoofing attacks and gets valid `client_ip` from request headers. Configurable with the `config.action_dispatch.ip_spoofing_check`, and `config.action_dispatch.trusted_proxies` options.
- `Rack::Sendfile` intercepts responses whose body is being served from a file and replaces it with a server specific X-Sendfile header. Configurable with `config.action_dispatch.x_sendfile_header`.
- `ActionDispatch::Callbacks` runs the prepare callbacks before serving the request.

- `ActiveRecord::ConnectionAdapters::ConnectionManagement` cleans active connections after each request, unless the `rack.test` key in the request environment is set to `true`.
- `ActiveRecord::QueryCache` caches all SELECT queries generated in a request. If any INSERT or UPDATE takes place then the cache is cleaned.
- `ActionDispatch::Cookies` sets cookies for the request.
- `ActionDispatch::Session::CookieStore` is responsible for storing the session in cookies. An alternate middleware can be used for this by changing the `config.action_controller.session_store` to an alternate value. Additionally, options passed to this can be configured by using `config.action_controller.session_options`.
- `ActionDispatch::Flash` sets up the `flash` keys. Only available if `config.action_controller.session_store` is set to a value.
- `ActionDispatch::ParamsParser` parses out parameters from the request into `params`.
- `Rack::MethodOverride` allows the method to be overridden if `params[:_method]` is set. This is the middleware which supports the PATCH, PUT, and DELETE HTTP method types.
- `Rack::Head` converts HEAD requests to GET requests and serves them as so.

Besides these usual middleware, you can add your own by using the `config.middleware.use` method:

```
config.middleware.use Magical::Unicorns
```

This will put the `Magical::Unicorns` middleware on the end of the stack. You can use `insert_before` if you wish to add a middleware before another.

```
config.middleware.insert_before Rack::Head, Magical::Unicorns
```

There's also `insert_after` which will insert a middleware after another:

```
config.middleware.insert_after Rack::Head, Magical::Unicorns
```

Middlewares can also be completely swapped out and replaced with others:

```
config.middleware.swap ActionController::Failsafe, Lifo::Failsafe
```

They can also be removed from the stack completely:

```
config.middleware.delete "Rack::MethodOverride"
```

3.5 Configuring i18n

All these configuration options are delegated to the `I18n` library.

- `config.i18n.available_locales` whitelists the available locales for the app. Defaults to all locale keys found in locale files, usually only `:en` on a new application.
- `config.i18n.default_locale` sets the default locale of an application used for i18n. Defaults to `:en`.
- `config.i18n.enforce_available_locales` ensures that all locales passed through i18n must be declared in the `available_locales` list, raising an `I18n::InvalidLocale` exception when setting an unavailable locale. Defaults to `true`. It is recommended not to disable this option unless strongly required, since this works as a security measure against setting any invalid locale from user input.
- `config.i18n.load_path` sets the path Rails uses to look for locale files. Defaults to `config/locales/*.{yml,rb}`.

3.6 Configuring Active Record

`config.active_record` includes a variety of configuration options:

- `config.active_record.logger` accepts a logger conforming to the interface of Log4r or the default Ruby Logger class, which is then passed on to any new database connections made. You can retrieve this logger by calling `logger` on either an Active Record model class or an Active Record model instance. Set to `nil` to disable logging.
- `config.active_record.primary_key_prefix_type` lets you adjust the naming for primary key columns. By default, Rails assumes that primary key columns are named `id` (and this configuration option doesn't need to be set.) There are two other choices: ** `:table_name` would make the primary key for the Customer class `customerid` ** `:table_name_with_underscore` would make the primary key for the Customer class `customer_id`
- `config.active_record.table_name_prefix` lets you set a global string to be prepended to table names. If you set this to `northwest_`, then the Customer class will look for `northwest_customers` as its table. The default is an empty string.
- `config.active_record.table_name_suffix` lets you set a global string to be appended to table names. If you set this to `_northwest`, then the Customer class will look for `customers_northwest` as its table. The default is an empty string.
- `config.active_record.schema_migrations_table_name` lets you set a string to be used as the name of the schema migrations table.
- `config.active_record.pluralize_table_names` specifies whether Rails will look for singular or plural table names in the database. If set to true (the default), then the Customer class will use the `customers` table. If set to false, then the Customer class will use the `customer` table.
- `config.active_record.default_timezone` determines whether to use `Time.local` (if set to `:local`) or `Time.utc` (if set to `:utc`) when pulling dates and times from the database. The default is `:utc`.
- `config.active_record.schema_format` controls the format for dumping the database schema to a file. The options are `:ruby` (the default) for a database-independent version that depends on migrations, or `:sql` for a set of (potentially database-dependent) SQL statements.
- `config.active_record.timestamped_migrations` controls whether migrations are numbered with serial integers or with timestamps. The default is true, to use timestamps, which are preferred if there are multiple developers working on the same application.
- `config.active_record.lock_optimistically` controls whether Active Record will use optimistic locking and is true by default.
- `config.active_record.cache_timestamp_format` controls the format of the timestamp value in the cache key. Default is `:number`.
- `config.active_record.record_timestamps` is a boolean value which controls whether or not timestamping of `create` and `update` operations on a model occur. The default value is `true`.

- `config.active_record.partial_writes` is a boolean value and controls whether or not partial writes are used (i.e. whether updates only set attributes that are dirty). Note that when using partial writes, you should also use optimistic locking `config.active_record.lock_optimistically` since concurrent updates may write attributes based on a possibly stale read state. The default value is `true`.

- `config.active_record.maintain_test_schema` is a boolean value which controls whether Active Record should try to keep your test database schema up-to-date with `db/schema.rb` (or `db/structure.sql`) when you run your tests. The default is true.

- `config.active_record.dump_schema_after_migration` is a flag which controls whether or not schema dump should happen (`db/schema.rb` or `db/structure.sql`) when you run migrations. This is set to false in `config/environments/production.rb` which is generated by Rails. The default value is true if this configuration is not set.

The MySQL adapter adds one additional configuration option:

- `ActiveRecord::ConnectionAdapters::MysqlAdapter.emulate_booleans` controls whether Active Record will consider all `tinyint(1)` columns in a MySQL database to be booleans and is true by default.

The schema dumper adds one additional configuration option:

- `ActiveRecord::SchemaDumper.ignore_tables` accepts an array of tables that should *not* be included in any generated schema file. This setting is ignored unless `config.active_record.schema_format == :ruby`.

3.7 Configuring Action Controller

`config.action_controller` includes a number of configuration settings:

- `config.action_controller.asset_host` sets the host for the assets. Useful when CDNs are used for hosting assets rather than the application server itself.

- `config.action_controller.perform_caching` configures whether the application should perform caching or not. Set to false in development mode, true in production.

- `config.action_controller.default_static_extension` configures the extension used for cached pages. Defaults to `.html`.

- `config.action_controller.default_charset` specifies the default character set for all renders. The default is "utf-8".

- `config.action_controller.logger` accepts a logger conforming to the interface of Log4r or the default Ruby Logger class, which is then used to log information from Action Controller. Set to `nil` to disable logging.

- `config.action_controller.request_forgery_protection_token` sets the token parameter name for RequestForgery. Calling `protect_from_forgery` sets it to `:authenticity_token` by default.

- `config.action_controller.allow_forgery_protection` enables or disables CSRF protection. By default this is `false` in test mode and `true` in all other modes.

- `config.action_controller.relative_url_root` can be used to tell Rails that you are deploying to a subdirectory. The default is `ENV['RAILS_RELATIVE_URL_ROOT']`.
- `config.action_controller.permit_all_parameters` sets all the parameters for mass assignment to be permitted by default. The default value is `false`.
- `config.action_controller.action_on_unpermitted_parameters` enables logging or raising an exception if parameters that are not explicitly permitted are found. Set to `:log` or `:raise` to enable. The default value is `:log` in development and test environments, and `false` in all other environments.
- `config.action_controller.always_permitted_parameters` sets a list of whitelisted parameters that are permitted by default. The default values are `['controller', 'action']`.

3.8 Configuring Action Dispatch

- `config.action_dispatch.session_store` sets the name of the store for session data. The default is `:cookie_store`; other valid options include `:active_record_store`, `:mem_cache_store` or the name of your own custom class.
- `config.action_dispatch.default_headers` is a hash with HTTP headers that are set by default in each response. By default, this is defined as:

```
config.action_dispatch.default_headers = {
  'X-Frame-Options' => 'SAMEORIGIN',
  'X-XSS-Protection' => '1; mode=block',
  'X-Content-Type-Options' => 'nosniff'
}
```

- `config.action_dispatch.tld_length` sets the TLD (top-level domain) length for the application. Defaults to `1`.
- `config.action_dispatch.http_auth_salt` sets the HTTP Auth salt value. Defaults to `'http authentication'`.
- `config.action_dispatch.signed_cookie_salt` sets the signed cookies salt value. Defaults to `'signed cookie'`.
- `config.action_dispatch.encrypted_cookie_salt` sets the encrypted cookies salt value. Defaults to `'encrypted cookie'`.
- `config.action_dispatch.encrypted_signed_cookie_salt` sets the signed encrypted cookies salt value. Defaults to `'signed encrypted cookie'`.
- `config.action_dispatch.perform_deep_munge` configures whether `deep_munge` method should be performed on the parameters. See Security Guide for more information. It defaults to true.
- `config.action_dispatch.rescue_responses` configures what exceptions are assigned to an HTTP status. It accepts a hash and you can specify pairs of exception/status. By default, this is defined as:

```
config.action_dispatch.rescue_responses = {
  'ActionController::RoutingError'              => :not_found,
  'AbstractController::ActionNotFound'          => :not_found,
  'ActionController::MethodNotAllowed'          => :method_not_allowed,
  'ActionController::UnknownHttpMethod'         => :method_not_allowed,
  'ActionController::NotImplemented'            => :not_implemented,
  'ActionController::UnknownFormat'             => :not_acceptable,
  'ActionController::InvalidAuthenticityToken'  => :unprocessable_entity,
  'ActionController::InvalidCrossOriginRequest' => :unprocessable_entity,
  'ActionDispatch::ParamsParser::ParseError'    => :bad_request,
  'ActionController::BadRequest'                => :bad_request,
  'ActionController::ParameterMissing'          => :bad_request,
  'ActiveRecord::RecordNotFound'                => :not_found,
  'ActiveRecord::StaleObjectError'              => :conflict,
  'ActiveRecord::RecordInvalid'                 => :unprocessable_entity,
  'ActiveRecord::RecordNotSaved'                => :unprocessable_entity
}
```

Any exceptions that are not configured will be mapped to 500 Internal Server Error.

- `ActionDispatch::Callbacks.before` takes a block of code to run before the request.
- `ActionDispatch::Callbacks.to_prepare` takes a block to run after `ActionDispatch::Callbacks.before`, but before the request. Runs for every request in `development` mode, but only once for `production` or environments with `cache_classes` set to `true`.
- `ActionDispatch::Callbacks.after` takes a block of code to run after the request.

3.9 Configuring Action View

`config.action_view` includes a small number of configuration settings:

- `config.action_view.field_error_proc` provides an HTML generator for displaying errors that come from Active Record. The default is

  ```
  Proc.new do |html_tag, instance|
    %Q(<div class="field_with_errors">#{html_tag}</div>).html_safe
  end
  ```

- `config.action_view.default_form_builder` tells Rails which form builder to use by default. The default is `ActionView::Helpers::FormBuilder`. If you want your form builder class to be loaded after initialization (so it's reloaded on each request in development), you can pass it as a `String`
- `config.action_view.logger` accepts a logger conforming to the interface of Log4r or the default Ruby Logger class, which is then used to log information from Action View. Set to `nil` to disable logging.
- `config.action_view.erb_trim_mode` gives the trim mode to be used by ERB. It defaults to `'-'`, which turns on trimming of tail spaces and newline when using <%= -%> or <%= =%>. See the Erubis documentation for more information.

- `config.action_view.embed_authenticity_token_in_remote_forms` allows you to set the default behavior for `authenticity_token` in forms with `:remote => true`. By default it's set to false, which means that remote forms will not include `authenticity_token`, which is helpful when you're fragment-caching the form. Remote forms get the authenticity from the `meta` tag, so embedding is unnecessary unless you support browsers without JavaScript. In such case you can either pass `:authenticity_token => true` as a form option or set this config setting to `true`

- `config.action_view.prefix_partial_path_with_controller_namespace` determines whether or not partials are looked up from a subdirectory in templates rendered from namespaced controllers. For example, consider a controller named `Admin::ArticlesController` which renders this template:

  ```
  <%= render @article %>
  ```

  The default setting is `true`, which uses the partial at `/admin/articles/_article.erb`. Setting the value to `false` would render `/articles/_article.erb`, which is the same behavior as rendering from a non-namespaced controller such as `ArticlesController`.

- `config.action_view.raise_on_missing_translations` determines whether an error should be raised for missing translations

3.10 Configuring Action Mailer

There are a number of settings available on `config.action_mailer`:

- `config.action_mailer.logger` accepts a logger conforming to the interface of Log4r or the default Ruby Logger class, which is then used to log information from Action Mailer. Set to `nil` to disable logging.

- `config.action_mailer.smtp_settings` allows detailed configuration for the `:smtp` delivery method. It accepts a hash of options, which can include any of these options:

  - `:address` - Allows you to use a remote mail server. Just change it from its default "localhost" setting.
  - `:port` - On the off chance that your mail server doesn't run on port 25, you can change it.
  - `:domain` - If you need to specify a HELO domain, you can do it here.
  - `:user_name` - If your mail server requires authentication, set the username in this setting.
  - `:password` - If your mail server requires authentication, set the password in this setting.
  - `:authentication` - If your mail server requires authentication, you need to specify the authentication type here. This is a symbol and one of `:plain`, `:login`, `:cram_md5`.

- `config.action_mailer.sendmail_settings` allows detailed configuration for the `sendmail` delivery method. It accepts a hash of options, which can include any of these options:

  - `:location` - The location of the sendmail executable. Defaults to `/usr/sbin/sendmail`.
  - `:arguments` - The command line arguments. Defaults to `-i -t`.

- `config.action_mailer.raise_delivery_errors` specifies whether to raise an error if email delivery cannot be completed. It defaults to true.

- `config.action_mailer.delivery_method` defines the delivery method and defaults to `:smtp`. See the configuration section in the Action Mailer guide for more info.
- `config.action_mailer.perform_deliveries` specifies whether mail will actually be delivered and is true by default. It can be convenient to set it to false for testing.
- `config.action_mailer.default_options` configures Action Mailer defaults. Use to set options like `from` or `reply_to` for every mailer. These default to:

```
mime_version:  "1.0",
charset:       "UTF-8",
content_type: "text/plain",
parts_order:  ["text/plain", "text/enriched", "text/html"]
```

  Assign a hash to set additional options:

```
config.action_mailer.default_options = {
  from: "noreply@example.com"
}
```

- `config.action_mailer.observers` registers observers which will be notified when mail is delivered.

```
config.action_mailer.observers = ["MailObserver"]
```

- `config.action_mailer.interceptors` registers interceptors which will be called before mail is sent.

```
config.action_mailer.interceptors = ["MailInterceptor"]
```

- `config.action_mailer.preview_path` specifies the location of mailer previews.

```
config.action_mailer.preview_path = "#{Rails.root}/lib/mailer_previews"
```

- `config.action_mailer.show_previews` enable or disable mailer previews. By default this is `true` in development.

```
config.action_mailer.show_previews = false
```

3.11 Configuring Active Support

There are a few configuration options available in Active Support:

- `config.active_support.bare` enables or disables the loading of `active_support/all` when booting Rails. Defaults to `nil`, which means `active_support/all` is loaded.
- `config.active_support.test_order` sets the order that test cases are executed. Possible values are `:sorted` and `:random`. Currently defaults to `:sorted`. In Rails 5.0, the default will be changed to `:random` instead.

- `config.active_support.escape_html_entities_in_json` enables or disables the escaping of HTML entities in JSON serialization. Defaults to `false`.
- `config.active_support.use_standard_json_time_format` enables or disables serializing dates to ISO 8601 format. Defaults to `true`.
- `config.active_support.time_precision` sets the precision of JSON encoded time values. Defaults to `3`.
- `ActiveSupport::Logger.silencer` is set to `false` to disable the ability to silence logging in a block. The default is `true`.
- `ActiveSupport::Cache::Store.logger` specifies the logger to use within cache store operations.
- `ActiveSupport::Deprecation.behavior` alternative setter to `config.active_support.deprecation` which configures the behavior of deprecation warnings for Rails.
- `ActiveSupport::Deprecation.silence` takes a block in which all deprecation warnings are silenced.
- `ActiveSupport::Deprecation.silenced` sets whether or not to display deprecation warnings.

3.12 Configuring a Database

Just about every Rails application will interact with a database. You can connect to the database by setting an environment variable `ENV['DATABASE_URL']` or by using a configuration file called `config/database.yml`.

Using the `config/database.yml` file you can specify all the information needed to access your database:

```
development:
  adapter: postgresql
  database: blog_development
  pool: 5
```

This will connect to the database named `blog_development` using the `postgresql` adapter. This same information can be stored in a URL and provided via an environment variable like this:

```
> puts ENV['DATABASE_URL']
postgresql://localhost/blog_development?pool=5
```

The `config/database.yml` file contains sections for three different environments in which Rails can run by default:

- The `development` environment is used on your development/local computer as you interact manually with the application.
- The `test` environment is used when running automated tests.
- The `production` environment is used when you deploy your application for the world to use.

If you wish, you can manually specify a URL inside of your `config/database.yml`

```
development:
  url: postgresql://localhost/blog_development?pool=5
```

The `config/database.yml` file can contain ERB tags <%= %>. Anything in the tags will be evaluated as Ruby code. You can use this to pull out data from an environment variable or to perform calculations to generate the needed connection information.

You don't have to update the database configurations manually. If you look at the options of the application generator, you will see that one of the options is named `--database`. This option allows you to choose an adapter from a list of the most used relational databases. You can even run the generator repeatedly: `cd .. && rails new blog --database=mysql`. When you confirm the overwriting of the `config/database.yml` file, your application will be configured for MySQL instead of SQLite. Detailed examples of the common database connections are below.

3.13 Connection Preference

Since there are two ways to set your connection, via environment variable it is important to understand how the two can interact.

If you have an empty `config/database.yml` file but your `ENV['DATABASE_URL']` is present, then Rails will connect to the database via your environment variable:

```
$ cat config/database.yml

$ echo $DATABASE_URL
postgresql://localhost/my_database
```

If you have a `config/database.yml` but no `ENV['DATABASE_URL']` then this file will be used to connect to your database:

```
$ cat config/database.yml
development:
  adapter: postgresql
  database: my_database
  host: localhost

$ echo $DATABASE_URL
```

If you have both `config/database.yml` and `ENV['DATABASE_URL']` set then Rails will merge the configuration together. To better understand this we must see some examples.

When duplicate connection information is provided the environment variable will take precedence:

```
$ cat config/database.yml
development:
  adapter: sqlite3
  database: NOT_my_database
  host: localhost

$ echo $DATABASE_URL
postgresql://localhost/my_database

$ bin/rails runner 'puts ActiveRecord::Base.configurations'
```

```
{"development"=>{"adapter"=>"postgresql", "host"=>"localhost", "database"=>
"my_database"}}
```

Here the adapter, host, and database match the information in `ENV['DATABASE_URL']`.

If non-duplicate information is provided you will get all unique values, environment variable still takes precedence in cases of any conflicts.

```
$ cat config/database.yml
development:
  adapter: sqlite3
  pool: 5

$ echo $DATABASE_URL
postgresql://localhost/my_database

$ bin/rails runner 'puts ActiveRecord::Base.configurations'
{"development"=>{"adapter"=>"postgresql", "host"=>"localhost", "database"=>
"my_database", "pool"=>5}}
```

Since pool is not in the `ENV['DATABASE_URL']` provided connection information its information is merged in. Since `adapter` is duplicate, the `ENV['DATABASE_URL']` connection information wins.

The only way to explicitly not use the connection information in `ENV['DATABASE_URL']` is to specify an explicit URL connection using the `"url"` sub key:

```
$ cat config/database.yml
development:
  url: sqlite3:NOT_my_database

$ echo $DATABASE_URL
postgresql://localhost/my_database

$ bin/rails runner 'puts ActiveRecord::Base.configurations'
{"development"=>{"adapter"=>"sqlite3", "database"=>"NOT_my_database"}}
```

Here the connection information in `ENV['DATABASE_URL']` is ignored, note the different adapter and database name.

Since it is possible to embed ERB in your `config/database.yml` it is best practice to explicitly show you are using the `ENV['DATABASE_URL']` to connect to your database. This is especially useful in production since you should not commit secrets like your database password into your source control (such as Git).

```
$ cat config/database.yml
production:
  url: <%= ENV['DATABASE_URL'] %>
```

Now the behavior is clear, that we are only using the connection information in `ENV['DATABASE_URL']`.

3.13.1 Configuring an SQLite3 Database Rails comes with built-in support for SQLite3, which is a lightweight serverless database application. While a busy production environment may overload SQLite, it works well for development and testing. Rails defaults to using an SQLite database when creating a new project, but you can always change it later.

Here's the section of the default configuration file (`config/database.yml`) with connection information for the development environment:

```
development:
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000
```

Rails uses an SQLite3 database for data storage by default because it is a zero configuration database that just works. Rails also supports MySQL and PostgreSQL "out of the box", and has plugins for many database systems. If you are using a database in a production environment Rails most likely has an adapter for it.

3.13.2 Configuring a MySQL Database If you choose to use MySQL instead of the shipped SQLite3 database, your `config/database.yml` will look a little different. Here's the development section:

```
development:
  adapter: mysql2
  encoding: utf8
  database: blog_development
  pool: 5
  username: root
  password:
  socket: /tmp/mysql.sock
```

If your development computer's MySQL installation includes a root user with an empty password, this configuration should work for you. Otherwise, change the username and password in the `development` section as appropriate.

3.13.3 Configuring a PostgreSQL Database If you choose to use PostgreSQL, your `config/database.yml` will be customized to use PostgreSQL databases:

```
development:
  adapter: postgresql
  encoding: unicode
  database: blog_development
  pool: 5
```

Prepared Statements are enabled by default on PostgreSQL. You can be disable prepared statements by setting `prepared_statements` to `false`:

```
production:
  adapter: postgresql
  prepared_statements: false
```

If enabled, Active Record will create up to 1000 prepared statements per database connection by default. To modify this behavior you can set `statement_limit` to a different value:

```
production:
  adapter: postgresql
  statement_limit: 200
```

The more prepared statements in use: the more memory your database will require. If your PostgreSQL database is hitting memory limits, try lowering `statement_limit` or disabling prepared statements.

3.13.4 Configuring an SQLite3 Database for JRuby Platform If you choose to use SQLite3 and are using JRuby, your `config/database.yml` will look a little different. Here's the development section:

```
development:
  adapter: jdbcsqlite3
  database: db/development.sqlite3
```

3.13.5 Configuring a MySQL Database for JRuby Platform If you choose to use MySQL and are using JRuby, your `config/database.yml` will look a little different. Here's the development section:

```
development:
  adapter: jdbcmysql
  database: blog_development
  username: root
  password:
```

3.13.6 Configuring a PostgreSQL Database for JRuby Platform If you choose to use PostgreSQL and are using JRuby, your `config/database.yml` will look a little different. Here's the development section:

```
development:
  adapter: jdbcpostgresql
  encoding: unicode
  database: blog_development
  username: blog
  password:
```

Change the username and password in the `development` section as appropriate.

3.14 Creating Rails Environments

By default Rails ships with three environments: "development", "test", and "production". While these are sufficient for most use cases, there are circumstances when you want more environments.

Imagine you have a server which mirrors the production environment but is only used for testing. Such a server is commonly called a "staging server". To define an environment called "staging" for this server, just create a file called `config/environments/staging.rb`. Please use the contents of any existing file in `config/environments` as a starting point and make the necessary changes from there.

That environment is no different than the default ones, start a server with `rails server -e staging`, a console with `rails console staging`, `Rails.env.staging?` works, etc.

3.15 Deploy to a subdirectory (relative url root)

By default Rails expects that your application is running at the root (eg. `/`). This section explains how to run your application inside a directory.

Let's assume we want to deploy our application to "/app1". Rails needs to know this directory to generate the appropriate routes:

```
config.relative_url_root = "/app1"
```

alternatively you can set the `RAILS_RELATIVE_URL_ROOT` environment variable.

Rails will now prepend "/app1" when generating links.

3.15.1 Using Passenger Passenger makes it easy to run your application in a subdirectory. You can find the relevant configuration in the Passenger manual.

3.15.2 Using a Reverse Proxy Deploying your application using a reverse proxy has definite advantages over traditional deploys. They allow you to have more control over your server by layering the components required by your application.

Many modern web servers can be used as a proxy server to balance third-party elements such as caching servers or application servers.

One such application server you can use is Unicorn to run behind a reverse proxy.

In this case, you would need to configure the proxy server (NGINX, Apache, etc) to accept connections from your application server (Unicorn). By default Unicorn will listen for TCP connections on port 8080, but you can change the port or configure it to use sockets instead.

You can find more information in the Unicorn readme and understand the philosophy behind it.

Once you've configured the application server, you must proxy requests to it by configuring your web server appropriately. For example your NGINX config may include:

```
upstream application_server {
  server 0.0.0.0:8080
}

server {
  listen 80;
  server_name localhost;

  root /root/path/to/your_app/public;

  try_files $uri/index.html $uri.html @app;

  location @app {
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;
    proxy_redirect off;
    proxy_pass http://application_server;
  }
```

```
  # some other configuration
}
```

Be sure to read the NGINX documentation for the most up-to-date information.

3.15.3 Considerations when deploying to a subdirectory Deploying to a subdirectory in production has implications on various parts of Rails.

- development environment:
- testing environment:
- serving static assets:
- asset pipeline:

4 Rails Environment Settings

Some parts of Rails can also be configured externally by supplying environment variables. The following environment variables are recognized by various parts of Rails:

- `ENV["RAILS_ENV"]` defines the Rails environment (production, development, test, and so on) that Rails will run under.
- `ENV["RAILS_RELATIVE_URL_ROOT"]` is used by the routing code to recognize URLs when you deploy your application to a subdirectory.
- `ENV["RAILS_CACHE_ID"]` and `ENV["RAILS_APP_VERSION"]` are used to generate expanded cache keys in Rails' caching code. This allows you to have multiple separate caches from the same application.

5 Using Initializer Files

After loading the framework and any gems in your application, Rails turns to loading initializers. An initializer is any Ruby file stored under `config/initializers` in your application. You can use initializers to hold configuration settings that should be made after all of the frameworks and gems are loaded, such as options to configure settings for these parts.

You can use subfolders to organize your initializers if you like, because Rails will look into the whole file hierarchy from the initializers folder on down.

If you have any ordering dependency in your initializers, you can control the load order through naming. Initializer files are loaded in alphabetical order by their path. For example, `01_critical.rb` will be loaded before `02_normal.rb`.

6 Initialization events

Rails has 5 initialization events which can be hooked into (listed in the order that they are run):

- `before_configuration`: This is run as soon as the application constant inherits from `Rails::Application`. The `config` calls are evaluated before this happens.

- `before_initialize`: This is run directly before the initialization process of the application occurs with the `:bootstrap_hook` initializer near the beginning of the Rails initialization process.
- `to_prepare`: Run after the initializers are run for all Railties (including the application itself), but before eager loading and the middleware stack is built. More importantly, will run upon every request in `development`, but only once (during boot-up) in `production` and `test`.
- `before_eager_load`: This is run directly before eager loading occurs, which is the default behavior for the `production` environment and not for the `development` environment.
- `after_initialize`: Run directly after the initialization of the application, after the application initializers in `config/initializers` are run.

To define an event for these hooks, use the block syntax within a `Rails::Application`, `Rails::Railtie` or `Rails::Engine` subclass:

```
module YourApp
  class Application < Rails::Application
    config.before_initialize do
      # initialization code goes here
    end
  end
end
```

Alternatively, you can also do it through the `config` method on the `Rails.application` object:

```
Rails.application.config.before_initialize do
  # initialization code goes here
end
```

Some parts of your application, notably routing, are not yet set up at the point where the `after_initialize` block is called.

6.1 Rails::Railtie#initializer

Rails has several initializers that run on startup that are all defined by using the `initializer` method from `Rails::Railtie`. Here's an example of the `set_helpers_path` initializer from Action Controller:

```
initializer "action_controller.set_helpers_path" do |app|
  ActionController::Helpers.helpers_path = app.helpers_paths
end
```

The `initializer` method takes three arguments with the first being the name for the initializer and the second being an options hash (not shown here) and the third being a block. The `:before` key in the options hash can be specified to specify which initializer this new initializer must run before, and the `:after` key will specify which initializer to run this initializer *after*.

Initializers defined using the `initializer` method will be run in the order they are defined in, with the exception of ones that use the `:before` or `:after` methods.

You may put your initializer before or after any other initializer in the chain, as long as it is logical. Say you have 4 initializers called "one" through "four" (defined in that order) and you define "four" to go *before* "four" but *after* "three", that just isn't logical and Rails will not be able to determine your initializer order.

The block argument of the `initializer` method is the instance of the application itself, and so we can access the configuration on it by using the `config` method as done in the example.

Because `Rails::Application` inherits from `Rails::Railtie` (indirectly), you can use the `initializer` method in `config/application.rb` to define initializers for the application.

6.2 Initializers

Below is a comprehensive list of all the initializers found in Rails in the order that they are defined (and therefore run in, unless otherwise stated).

- `load_environment_hook` Serves as a placeholder so that `:load_environment_config` can be defined to run before it.
- `load_active_support` Requires `active_support/dependencies` which sets up the basis for Active Support. Optionally requires `active_support/all` if `config.active_support.bare` is un-truthful, which is the default.
- `initialize_logger` Initializes the logger (an `ActiveSupport::Logger` object) for the application and makes it accessible at `Rails.logger`, provided that no initializer inserted before this point has defined `Rails.logger`.
- `initialize_cache` If `Rails.cache` isn't set yet, initializes the cache by referencing the value in `config.cache_store` and stores the outcome as `Rails.cache`. If this object responds to the `middleware` method, its middleware is inserted before `Rack::Runtime` in the middleware stack.
- `set_clear_dependencies_hook` Provides a hook for `active_record.set_dispatch_hooks` to use, which will run before this initializer. This initializer - which runs only if `cache_classes` is set to `false` - uses `ActionDispatch::Callbacks.after` to remove the constants which have been referenced during the request from the object space so that they will be reloaded during the following request.
- `initialize_dependency_mechanism` If `config.cache_classes` is true, configures `ActiveSupport::Dependencies.mechanism` to `require` dependencies rather than `load` them.
- `bootstrap_hook` Runs all configured `before_initialize` blocks.
- `i18n.callbacks` In the development environment, sets up a `to_prepare` callback which will call `I18n.reload!` if any of the locales have changed since the last request. In production mode this callback will only run on the first request.
- `active_support.deprecation_behavior` Sets up deprecation reporting for environments, defaulting to `:log` for development, `:notify` for production and `:stderr` for test. If a value isn't set for `config.active_support.deprecation` then this initializer will prompt the user to configure this line in the current environment's `config/environments` file. Can be set to an array of values.
- `active_support.initialize_time_zone` Sets the default time zone for the application based on the `config.time_zone` setting, which defaults to "UTC".

- `active_support.initialize_beginning_of_week` Sets the default beginning of week for the application based on `config.beginning_of_week` setting, which defaults to `:monday`.
- `action_dispatch.configure` Configures the `ActionDispatch::Http::URL.tld_length` to be set to the value of `config.action_dispatch.tld_length`.
- `action_view.set_configs` Sets up Action View by using the settings in `config.action_view` by `send`'ing the method names as setters to `ActionView::Base` and passing the values through.
- `action_controller.logger` Sets `ActionController::Base.logger` - if it's not already set - to `Rails.logger`.
- `action_controller.initialize_framework_caches` Sets `ActionController::Base.cache_store` - if it's not already set - to `Rails.cache`.
- `action_controller.set_configs` Sets up Action Controller by using the settings in `config.action_controller` by `send`'ing the method names as setters to `ActionController::Base` and passing the values through.
- `action_controller.compile_config_methods` Initializes methods for the config settings specified so that they are quicker to access.
- `active_record.initialize_timezone` Sets `ActiveRecord::Base.time_zone_aware_attributes` to true, as well as setting `ActiveRecord::Base.default_timezone` to UTC. When attributes are read from the database, they will be converted into the time zone specified by `Time.zone`.
- `active_record.logger` Sets `ActiveRecord::Base.logger` - if it's not already set - to `Rails.logger`.
- `active_record.set_configs` Sets up Active Record by using the settings in `config.active_record` by `send`'ing the method names as setters to `ActiveRecord::Base` and passing the values through.
- `active_record.initialize_database` Loads the database configuration (by default) from `config/database.yml` and establishes a connection for the current environment.
- `active_record.log_runtime` Includes `ActiveRecord::Railties::ControllerRuntime` which is responsible for reporting the time taken by Active Record calls for the request back to the logger.
- `active_record.set_dispatch_hooks` Resets all reloadable connections to the database if `config.cache_classes` is set to `false`.
- `action_mailer.logger` Sets `ActionMailer::Base.logger` - if it's not already set - to `Rails.logger`.
- `action_mailer.set_configs` Sets up Action Mailer by using the settings in `config.action_mailer` by `send`'ing the method names as setters to `ActionMailer::Base` and passing the values through.
- `action_mailer.compile_config_methods` Initializes methods for the config settings specified so that they are quicker to access.
- `set_load_path` This initializer runs before `bootstrap_hook`. Adds the `vendor`, `lib`, all directories of `app` and any paths specified by `config.load_paths` to `$LOAD_PATH`.

- `set_autoload_paths` This initializer runs before `bootstrap_hook`. Adds all sub-directories of `app` and paths specified by `config.autoload_paths` to `ActiveSupport::Dependencies.autoload_paths`.
- `add_routing_paths` Loads (by default) all `config/routes.rb` files (in the application and railties, including engines) and sets up the routes for the application.
- `add_locales` Adds the files in `config/locales` (from the application, railties and engines) to `I18n.load_path`, making available the translations in these files.
- `add_view_paths` Adds the directory `app/views` from the application, railties and engines to the lookup path for view files for the application.
- `load_environment_config` Loads the `config/environments` file for the current environment.
- `append_asset_paths` Finds asset paths for the application and all attached railties and keeps a track of the available directories in `config.static_asset_paths`.
- `prepend_helpers_path` Adds the directory `app/helpers` from the application, railties and engines to the lookup path for helpers for the application.
- `load_config_initializers` Loads all Ruby files from `config/initializers` in the application, railties and engines. The files in this directory can be used to hold configuration settings that should be made after all of the frameworks are loaded.
- `engines_blank_point` Provides a point-in-initialization to hook into if you wish to do anything before engines are loaded. After this point, all railtie and engine initializers are run.
- `add_generator_templates` Finds templates for generators at `lib/templates` for the application, railties and engines and adds these to the `config.generators.templates` setting, which will make the templates available for all generators to reference.
- `ensure_autoload_once_paths_as_subset` Ensures that the `config.autoload_once_paths` only contains paths from `config.autoload_paths`. If it contains extra paths, then an exception will be raised.
- `add_to_prepare_blocks` The block for every `config.to_prepare` call in the application, a railtie or engine is added to the `to_prepare` callbacks for Action Dispatch which will be run per request in development, or before the first request in production.
- `add_builtin_route` If the application is running under the development environment then this will append the route for `rails/info/properties` to the application routes. This route provides the detailed information such as Rails and Ruby version for `public/index.html` in a default Rails application.
- `build_middleware_stack` Builds the middleware stack for the application, returning an object which has a `call` method which takes a Rack environment object for the request.
- `eager_load!` If `config.eager_load` is true, runs the `config.before_eager_load` hooks and then calls `eager_load!` which will load all `config.eager_load_namespaces`.
- `finisher_hook` Provides a hook for after the initialization of process of the application is complete, as well as running all the `config.after_initialize` blocks for the application, railties and engines.

- `set_routes_reloader` Configures Action Dispatch to reload the routes file using `ActionDispatch::Callbacks.to_prepare`.
- `disable_dependency_loading` Disables the automatic dependency loading if the `config.eager_load` is set to true.

7 Database pooling

Active Record database connections are managed by `ActiveRecord::ConnectionAdapters::ConnectionPool` which ensures that a connection pool synchronizes the amount of thread access to a limited number of database connections. This limit defaults to 5 and can be configured in `database.yml`.

```
development:
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000
```

Since the connection pooling is handled inside of Active Record by default, all application servers (Thin, mongrel, Unicorn etc.) should behave the same. Initially, the database connection pool is empty and it will create additional connections as the demand for them increases, until it reaches the connection pool limit.

Any one request will check out a connection the first time it requires access to the database, after which it will check the connection back in, at the end of the request, meaning that the additional connection slot will be available again for the next request in the queue.

If you try to use more connections than are available, Active Record will block and wait for a connection from the pool. When it cannot get connection, a timeout error similar to given below will be thrown.

```
ActiveRecord::ConnectionTimeoutError - could not obtain a database connection within 5
seconds. The max pool size is currently 5; consider increasing it:
```

If you get the above error, you might want to increase the size of connection pool by incrementing the `pool` option in `database.yml`

If you are running in a multi-threaded environment, there could be a chance that several threads may be accessing multiple connections simultaneously. So depending on your current request load, you could very well have multiple threads contending for a limited amount of connections.

8 Custom configuration

You can configure your own code through the Rails configuration object with custom configuration. It works like this:

```
config.x.payment_processing.schedule = :daily
config.x.payment_processing.retries  = 3
config.x.super_debugger = true
```

These configuration points are then available through the configuration object:

```
Rails.configuration.x.payment_processing.schedule # => :daily
Rails.configuration.x.payment_processing.retries  # => 3
Rails.configuration.x.super_debugger              # => true
Rails.configuration.x.super_debugger.not_set      # => nil
```

9 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.