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

Rails on Rack

This guide covers Rails integration with Rack and interfacing with other Rack components.

After reading this guide, you will know:

- How to use Rack Middlewares in your Rails applications.
- Action Pack's internal Middleware stack.
- How to define a custom Middleware stack.

WARNING: This guide assumes a working knowledge of Rack protocol and Rack concepts such as middlewares, URL maps, and `Rack::Builder`.

Introduction to Rack

Rack provides a minimal, modular, and adaptable interface for developing web applications in Ruby. By wrapping HTTP requests and responses in the simplest way possible, it unifies and distills the API for web servers, web frameworks, and software in between (the so-called middleware) into a single method call.

Explaining how Rack works is not really in the scope of this guide. In case you are not familiar with Rack's basics, you should check out the Resources section below.

Rails on Rack

Rails Application's Rack Object

`Rails.application` is the primary Rack application object of a Rails application. Any Rack compliant web server should be using `Rails.application` object to serve a Rails application.

`bin/rails server`

`bin/rails server` does the basic job of creating a `Rack::Server` object and starting the web server.

Here's how `bin/rails server` creates an instance of `Rack::Server`

```
Rails::Server.new.tap do |server|
  require APP_PATH
  Dir.chdir(Rails.application.root)
  server.start
end
```

The `Rails::Server` inherits from `Rack::Server` and calls the `Rack::Server#start` method this way:

```
class Server < ::Rack::Server
  def start
    # ...
    super
  end
end
```

`rackup`

To use `rackup` instead of Rails' `bin/rails server`, you can put the following inside `config.ru` of your Rails application's root directory:

```
# Rails.root/config.ru
require_relative "config/environment"
run Rails.application
```

And start the server:

```
$ rackup config.ru
```

To find out more about different `rackup` options, you can run:

```
$ rackup --help
```

Development and auto-reloading

Middlewares are loaded once and are not monitored for changes. You will have to restart the server for changes to be reflected in the running application.

Action Dispatcher Middleware Stack

Many of Action Dispatcher's internal components are implemented as Rack middlewares. `Rails::Application` uses `ActionDispatch::MiddlewareStack` to combine various internal and external middlewares to form a complete Rails Rack application.

NOTE: `ActionDispatch::MiddlewareStack` is Rails' equivalent of `Rack::Builder`, but is built for better flexibility and more features to meet Rails' requirements.

Inspecting Middleware Stack

Rails has a handy command for inspecting the middleware stack in use:

```
$ bin/rails middleware
```

For a freshly generated Rails application, this might produce something like:

```
use ActionDispatch::HostAuthorization
use Rack::Sendfile
use ActionDispatch::Static
use ActionDispatch::Executor
use ActionDispatch::ServerTiming
use ActiveSupport::Cache::Strategy::LocalCache::Middleware
use Rack::Runtime
use Rack::MethodOverride
use ActionDispatch::RequestId
use ActionDispatch::RemoteIp
```

```
use Sprockets::Rails::QuietAssets
use Rails::Rack::Logger
use ActionDispatch::ShowExceptions
use WebConsole::Middleware
use ActionDispatch::DebugExceptions
use ActionDispatch::ActionableExceptions
use ActionDispatch::Reloader
use ActionDispatch::Callbacks
use ActiveRecord::Migration::CheckPending
use ActionDispatch::Cookies
use ActionDispatch::Session::CookieStore
use ActionDispatch::Flash
use ActionDispatch::ContentSecurityPolicy::Middleware
use Rack::Head
use Rack::ConditionalGet
use Rack::ETag
use Rack::TempfileReaper
run MyApp::Application.routes
```

The default middlewares shown here (and some others) are each summarized in the Internal Middlewares section, below.

Configuring Middleware Stack

Rails provides a simple configuration interface `config.middleware` for adding, removing, and modifying the middlewares in the middleware stack via `application.rb` or the environment specific configuration file `environments/<environment>.rb`.

Adding a Middleware

You can add a new middleware to the middleware stack using any of the following methods:

- `config.middleware.use(new_middleware, args)` - Adds the new middleware at the bottom of the middleware stack.
- `config.middleware.insert_before(existing_middleware, new_middleware, args)` - Adds the new middleware before the specified existing middleware in the middleware stack.
- `config.middleware.insert_after(existing_middleware, new_middleware, args)` - Adds the new middleware after the specified existing middleware in the middleware stack.

```
# config/application.rb

# Push Rack::BounceFavicon at the bottom
config.middleware.use Rack::BounceFavicon

# Add Lifo::Cache after ActionDispatch::Executor.
# Pass { page_cache: false } argument to Lifo::Cache.
config.middleware.insert_after ActionDispatch::Executor, Lifo::Cache, page_cache:
false
```

Swapping a Middleware

You can swap an existing middleware in the middleware stack using `config.middleware.swap`.

```
# config/application.rb

# Replace ActionDispatch::ShowExceptions with Lifo::ShowExceptions
config.middleware.swap ActionDispatch::ShowExceptions, Lifo::ShowExceptions
```

Deleting a Middleware

Add the following lines to your application configuration:

```
# config/application.rb
config.middleware.delete Rack::Runtime
```

And now if you inspect the middleware stack, you'll find that `Rack::Runtime` is not a part of it.

```
$ bin/rails middleware
(in /Users/lifo/Rails/blog)
use ActionDispatch::Static
use #<ActiveSupport::Cache::Strategy::LocalCache::Middleware:0x00000001c304c8>
...
run Rails.application.routes
```

If you want to remove session related middleware, do the following:

```
# config/application.rb
config.middleware.delete ActionDispatch::Cookies
config.middleware.delete ActionDispatch::Session::CookieStore
config.middleware.delete ActionDispatch::Flash
```

And to remove browser related middleware,

```
# config/application.rb
config.middleware.delete Rack::MethodOverride
```

If you want an error to be raised when you try to delete a non-existent item, use `delete!` instead.

```
# config/application.rb
config.middleware.delete! ActionDispatch::Executor
```

Internal Middleware Stack

Much of Action Controller's functionality is implemented as Middlewares. The following list explains the purpose of each of them:

`ActionDispatch::HostAuthorization`

- Guards from DNS rebinding attacks by explicitly permitting the hosts a request can be sent to. See the configuration guide for configuration instructions.

`Rack::Sendfile`

- Sets server specific X-Sendfile header. Configure this via `config.action_dispatch.x_sendfile_header` option.

`ActionDispatch::Static`

- Used to serve static files from the public directory. Disabled if `config.public_file_server.enabled` is `false`.

`Rack::Lock`

- Sets `env["rack.multithread"]` flag to `false` and wraps the application within a Mutex.

`ActionDispatch::Executor`

- Used for thread safe code reloading during development.

`ActionDispatch::ServerTiming`

- Sets a `Server-Timing` header containing performance metrics for the request.

`ActiveSupport::Cache::Strategy::LocalCache::Middleware`

- Used for memory caching. This cache is not thread safe.

`Rack::Runtime`

- Sets an X-Runtime header, containing the time (in seconds) taken to execute the request.

`Rack::MethodOverride`

- Allows the method to be overridden if `params[:_method]` is set. This is the middleware which supports the PUT and DELETE HTTP method types.

`ActionDispatch::RequestId`

- Makes a unique `X-Request-Id` header available to the response and enables the `ActionDispatch::Request#request_id` method.

`ActionDispatch::RemoteIp`

- Checks for IP spoofing attacks.

`Sprockets::Rails::QuietAssets`

- Suppresses logger output for asset requests.

`Rails::Rack::Logger`

- Notifies the logs that the request has begun. After the request is complete, flushes all the logs.

`ActionDispatch::ShowExceptions`

- Rescues any exception returned by the application and calls an exceptions app that will wrap it in a format for the end user.

`ActionDispatch::DebugExceptions`

- Responsible for logging exceptions and showing a debugging page in case the request is local.

`ActionDispatch::ActionableExceptions`

- Provides a way to dispatch actions from Rails' error pages.

`ActionDispatch::Reloader`

- Provides prepare and cleanup callbacks, intended to assist with code reloading during development.

`ActionDispatch::Callbacks`

- Provides callbacks to be executed before and after dispatching the request.

`ActiveRecord::Migration::CheckPending`

- Checks pending migrations and raises `ActiveRecord::PendingMigrationError` if any migrations are pending.

`ActionDispatch::Cookies`

- Sets cookies for the request.

`ActionDispatch::Session::CookieStore`

- Responsible for storing the session in cookies.

`ActionDispatch::Flash`

- Sets up the flash keys. Only available if `config.session_store` is set to a value.

`ActionDispatch::ContentSecurityPolicy::Middleware`

- Provides a DSL to configure a Content-Security-Policy header.

`Rack::Head`

- Converts HEAD requests to `GET` requests and serves them as so.

`Rack::ConditionalGet`

- Adds support for "Conditional `GET` " so that server responds with nothing if the page wasn't changed.

`Rack::ETag`

- Adds ETag header on all String bodies. ETags are used to validate cache.

`Rack::TempfileReaper`

- Cleans up tempfiles used to buffer multipart requests.

TIP: It's possible to use any of the above middlewares in your custom Rack stack.

Resources

Learning Rack

- Official Rack Website
- Introducing Rack

Understanding Middlewares

- Railscast on Rack Middlewares