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

Active Support Instrumentation

Active Support is a part of core Rails that provides Ruby language extensions, utilities, and other things. One of the things it includes is an instrumentation API that can be used inside an application to measure certain actions that occur within Ruby code, such as that inside a Rails application or the framework itself. It is not limited to Rails, however. It can be used independently in other Ruby scripts if it is so desired.

In this guide, you will learn how to use the instrumentation API inside of Active Support to measure events inside of Rails and other Ruby code.

After reading this guide, you will know:

- What instrumentation can provide.
- How to add a subscriber to a hook.
- The hooks inside the Rails framework for instrumentation.
- How to build a custom instrumentation implementation.

Introduction to instrumentation

The instrumentation API provided by Active Support allows developers to provide hooks which other developers may hook into. There are several of these within the Rails framework. With this API, developers can choose to be notified when certain events occur inside their application or another piece of Ruby code.

For example, there is a hook provided within Active Record that is called every time Active Record uses an SQL query on a database. This hook could be **subscribed** to, and used to track the number of queries during a certain action. There's another hook around the processing of an action of a controller. This could be used, for instance, to track how long a specific action has taken.

You are even able to create your own events inside your application which you can later subscribe to.

Subscribing to an event

Subscribing to an event is easy. Use `ActiveSupport::Notifications.subscribe` with a block to listen to any notification.

The block receives the following arguments:

- The name of the event
- Time when it started
- Time when it finished
- A unique ID for the instrumenter that fired the event
- The payload (described in future sections)

```
ActiveSupport::Notifications.subscribe "process_action.action_controller" do |name,
started, finished, unique_id, data|
  # your own custom stuff
  Rails.logger.info "#{name} Received! (started: #{started}, finished: #{finished})"
# process_action.action_controller Received (started: 2019-05-05 13:43:57 -0800,
finished: 2019-05-05 13:43:58 -0800)
end
```

If you are concerned about the accuracy of `started` and `finished` to compute a precise elapsed time then use `ActiveSupport::Notifications.monotonic_subscribe`. The given block would receive the same arguments as above but the `started` and `finished` will have values with an accurate monotonic time instead of wall-clock time.

```
ActiveSupport::Notifications.monotonic_subscribe "process_action.action_controller"
do |name, started, finished, unique_id, data|
  # your own custom stuff
  Rails.logger.info "#{name} Received! (started: #{started}, finished: #{finished})"
# process_action.action_controller Received (started: 1560978.425334, finished:
1560979.429234)
end
```

Defining all those block arguments each time can be tedious. You can easily create an `ActiveSupport::Notifications::Event` from block arguments like this:

```
ActiveSupport::Notifications.subscribe "process_action.action_controller" do |*args|
  event = ActiveSupport::Notifications::Event.new *args

  event.name      # => "process_action.action_controller"
  event.duration  # => 10 (in milliseconds)
  event.payload   # => {:extra=>information}

  Rails.logger.info "#{event} Received!"
end
```

You may also pass a block that accepts only one argument, and it will receive an event object:

```
ActiveSupport::Notifications.subscribe "process_action.action_controller" do |event|
  event.name      # => "process_action.action_controller"
  event.duration  # => 10 (in milliseconds)
  event.payload   # => {:extra=>information}

  Rails.logger.info "#{event} Received!"
end
```

Most times you only care about the data itself. Here is a shortcut to just get the data.

```
ActiveSupport::Notifications.subscribe "process_action.action_controller" do |*args|
  data = args.extract_options!
  data # { extra: :information }
end
```

You may also subscribe to events matching a regular expression. This enables you to subscribe to multiple events at once. Here's how to subscribe to everything from `ActionController`.

```
ActiveSupport::Notifications.subscribe /action_controller/ do |*args|
  # inspect all ActionController events
end
```

Rails framework hooks

Within the Ruby on Rails framework, there are a number of hooks provided for common events. These are detailed below.

Action Controller

write_fragment.action_controller

Key	Value
`:key`	The complete key

```
{
  key: 'posts/1-dashboard-view'
}
```

read_fragment.action_controller

Key	Value
`:key`	The complete key

```
{
  key: 'posts/1-dashboard-view'
}
```

expire_fragment.action_controller

Key	Value
`:key`	The complete key

```
{
  key: 'posts/1-dashboard-view'
}
```

exist_fragment?.action_controller

Key	Value
`:key`	The complete key

```
{
  key: 'posts/1-dashboard-view'
```

```
}
```

start_processing.action_controller

Key	Value
`:controller`	The controller name
`:action`	The action
`:params`	Hash of request parameters without any filtered parameter
`:headers`	Request headers
`:format`	html/js/json/xml etc
`:method`	HTTP request verb
`:path`	Request path

```
{
  controller: "PostsController",
  action: "new",
  params: { "action" => "new", "controller" => "posts" },
  headers: #<ActionDispatch::Http::Headers:0x0055a67a519b88>,
  format: :html,
  method: "GET",
  path: "/posts/new"
}
```

process_action.action_controller

Key	Value
`:controller`	The controller name
`:action`	The action
`:params`	Hash of request parameters without any filtered parameter
`:headers`	Request headers
`:format`	html/js/json/xml etc
`:method`	HTTP request verb
`:path`	Request path
`:request`	The `ActionDispatch::Request`
`:response`	The `ActionDispatch::Response`
`:status`	HTTP status code
`:view_runtime`	Amount spent in view in ms
`:db_runtime`	Amount spent executing database queries in ms

```
{
  controller: "PostsController",
  action: "index",
  params: {"action" => "index", "controller" => "posts"},
  headers: #<ActionDispatch::Http::Headers:0x0055a67a519b88>,
  format: :html,
  method: "GET",
  path: "/posts",
  request: #<ActionDispatch::Request:0x00007ff1cb9bd7b8>,
  response: #<ActionDispatch::Response:0x00007f8521841ec8>,
  status: 200,
  view_runtime: 46.848,
  db_runtime: 0.157
}
```

send_file.action_controller

Key	Value
`:path`	Complete path to the file

INFO. Additional keys may be added by the caller.

send_data.action_controller

`ActionController` does not add any specific information to the payload. All options are passed through to the payload.

redirect_to.action_controller

Key	Value
`:status`	HTTP response code
`:location`	URL to redirect to
`:request`	The `ActionDispatch::Request`

```
{
  status: 302,
  location: "http://localhost:3000/posts/new",
  request: #<ActionDispatch::Request:0x00007ff1cb9bd7b8>
}
```

halted_callback.action_controller

Key	Value
`:filter`	Filter that halted the action

```
{
  filter: ":halting_filter"
}
```

unpermitted_parameters.action_controller

Key	Value
`:keys`	The unpermitted keys
`:context`	Hash with the following keys: `:controller`, `:action`, `:params`, `:request`

Action Dispatch

process_middleware.action_dispatch

Key	Value
`:middleware`	Name of the middleware

Action View

render_template.action_view

Key	Value
`:identifier`	Full path to template
`:layout`	Applicable layout

```
{
  identifier: "/Users/adam/projects/notifications/app/views/posts/index.html.erb",
  layout: "layouts/application"
}
```

render_partial.action_view

Key	Value
`:identifier`	Full path to template

```
{
  identifier: "/Users/adam/projects/notifications/app/views/posts/_form.html.erb"
}
```

render_collection.action_view

Key	Value
`:identifier`	Full path to template
`:count`	Size of collection

`:cache_hits`	Number of partials fetched from cache

`:cache_hits` is only included if the collection is rendered with `cached: true`.

```
{
  identifier: "/Users/adam/projects/notifications/app/views/posts/_post.html.erb",
  count: 3,
  cache_hits: 0
}
```

render_layout.action_view

Key	Value
`:identifier`	Full path to template

```
{
  identifier:
"/Users/adam/projects/notifications/app/views/layouts/application.html.erb"
}
```

Active Record

sql.active_record

Key	Value
`:sql`	SQL statement
`:name`	Name of the operation
`:connection`	Connection object
`:binds`	Bind parameters
`:type_casted_binds`	Typecasted bind parameters
`:statement_name`	SQL Statement name
`:cached`	`true` is added when cached queries used

INFO. The adapters will add their own data as well.

```
{
  sql: "SELECT \"posts\".* FROM \"posts\" ",
  name: "Post Load",
  connection: #
<ActiveRecord::ConnectionAdapters::SQLite3Adapter:0x00007f9f7a838850>,
  binds: [#<ActiveModel::Attribute::WithCastValue:0x00007fe19d15dc00>],
  type_casted_binds: [11],
  statement_name: nil
}
```

instantiation.active_record

Key	Value
`:record_count`	Number of records that instantiated
`:class_name`	Record's class

```
{
  record_count: 1,
  class_name: "User"
}
```

Action Mailer

deliver.action_mailer

Key	Value
`:mailer`	Name of the mailer class
`:message_id`	ID of the message, generated by the Mail gem
`:subject`	Subject of the mail
`:to`	To address(es) of the mail
`:from`	From address of the mail
`:bcc`	BCC addresses of the mail
`:cc`	CC addresses of the mail
`:date`	Date of the mail
`:mail`	The encoded form of the mail
`:perform_deliveries`	Whether delivery of this message is performed or not

```
{
  mailer: "Notification",
  message_id: "4f5b5491f1774_181b23fc3d4434d38138e5@mba.local.mail",
  subject: "Rails Guides",
  to: ["users@rails.com", "dhh@rails.com"],
  from: ["me@rails.com"],
  date: Sat, 10 Mar 2012 14:18:09 +0100,
  mail: "...", # omitted for brevity
  perform_deliveries: true
}
```

process.action_mailer

Key	Value
`:mailer`	Name of the mailer class

`:action`	The action
`:args`	The arguments

```
{
  mailer: "Notification",
  action: "welcome_email",
  args: []
}
```

Active Support

cache_read.active_support

Key	Value
`:key`	Key used in the store
`:store`	Name of the store class
`:hit`	If this read is a hit
`:super_operation`	`:fetch` is added when a read is used with `#fetch`

cache_generate.active_support

This event is only used when `#fetch` is called with a block.

Key	Value
`:key`	Key used in the store
`:store`	Name of the store class

INFO. Options passed to fetch will be merged with the payload when writing to the store

```
{
  key: "name-of-complicated-computation",
  store: "ActiveSupport::Cache::MemCacheStore"
}
```

cache_fetch_hit.active_support

This event is only used when `#fetch` is called with a block.

Key	Value
`:key`	Key used in the store
`:store`	Name of the store class

INFO. Options passed to fetch will be merged with the payload.

```
{
  key: "name-of-complicated-computation",
  store: "ActiveSupport::Cache::MemCacheStore"
}
```

cache_write.active_support

Key	Value
`:key`	Key used in the store
`:store`	Name of the store class

INFO. Cache stores may add their own keys

```
{
  key: "name-of-complicated-computation",
  store: "ActiveSupport::Cache::MemCacheStore"
}
```

cache_delete.active_support

Key	Value
`:key`	Key used in the store
`:store`	Name of the store class

```
{
  key: "name-of-complicated-computation",
  store: "ActiveSupport::Cache::MemCacheStore"
}
```

cache_exist?.active_support

Key	Value
`:key`	Key used in the store
`:store`	Name of the store class

```
{
  key: "name-of-complicated-computation",
  store: "ActiveSupport::Cache::MemCacheStore"
}
```

Active Job

enqueue_at.active_job

Key	Value

`:adapter`	QueueAdapter object processing the job
`:job`	Job object

enqueue.active_job

Key	Value
`:adapter`	QueueAdapter object processing the job
`:job`	Job object

enqueue_retry.active_job

Key	Value
`:job`	Job object
`:adapter`	QueueAdapter object processing the job
`:error`	The error that caused the retry
`:wait`	The delay of the retry

perform_start.active_job

Key	Value
`:adapter`	QueueAdapter object processing the job
`:job`	Job object

perform.active_job

Key	Value
`:adapter`	QueueAdapter object processing the job
`:job`	Job object

retry_stopped.active_job

Key	Value
`:adapter`	QueueAdapter object processing the job
`:job`	Job object
`:error`	The error that caused the retry

discard.active_job

Key	Value
`:adapter`	QueueAdapter object processing the job
`:job`	Job object

`:error`	The error that caused the discard

Action Cable

perform_action.action_cable

Key	Value
`:channel_class`	Name of the channel class
`:action`	The action
`:data`	A hash of data

transmit.action_cable

Key	Value
`:channel_class`	Name of the channel class
`:data`	A hash of data
`:via`	Via

transmit_subscription_confirmation.action_cable

Key	Value
`:channel_class`	Name of the channel class

transmit_subscription_rejection.action_cable

Key	Value
`:channel_class`	Name of the channel class

broadcast.action_cable

Key	Value
`:broadcasting`	A named broadcasting
`:message`	A hash of message
`:coder`	The coder

Active Storage

service_upload.active_storage

Key	Value
`:key`	Secure token
`:service`	Name of the service
`:checksum`	Checksum to ensure integrity

service_streaming_download.active_storage

Key	Value
`:key`	Secure token
`:service`	Name of the service

service_download_chunk.active_storage

Key	Value
`:key`	Secure token
`:service`	Name of the service
`:range`	Byte range attempted to be read

service_download.active_storage

Key	Value
`:key`	Secure token
`:service`	Name of the service

service_delete.active_storage

Key	Value
`:key`	Secure token
`:service`	Name of the service

service_delete_prefixed.active_storage

Key	Value
`:prefix`	Key prefix
`:service`	Name of the service

service_exist.active_storage

Key	Value
`:key`	Secure token
`:service`	Name of the service
`:exist`	File or blob exists or not

service_url.active_storage

Key	Value
`:key`	Secure token

`:service`	Name of the service
`:url`	Generated URL

service_update_metadata.active_storage

Key	Value
`:key`	Secure token
`:service`	Name of the service
`:content_type`	HTTP Content-Type field
`:disposition`	HTTP Content-Disposition field

INFO. The only ActiveStorage service that provides this hook so far is GCS.

preview.active_storage

Key	Value
`:key`	Secure token

transform.active_storage

analyze.active_storage

Key	Value
`:analyzer`	Name of analyzer e.g., ffprobe

Action Mailbox

process.action_mailbox

Key	Value
`:mailbox`	Instance of the Mailbox class inheriting from ActionMailbox::Base
`:inbound_email`	Hash with data about the inbound email being processed

```
{
  mailbox: #<RepliesMailbox:0x00007f9f7a8388>,
  inbound_email: {
    id: 1,
    message_id: "0CB459E0-0336-41DA-BC88-E6E28C697DDB@37signals.com",
    status: "processing"
  }
}
```

Railties

load_config_initializer.railties

Key	Value
`:initializer`	Path to loaded initializer from `config/initializers`

Rails

deprecation.rails

Key	Value
`:message`	The deprecation warning
`:callstack`	Where the deprecation came from

Exceptions

If an exception happens during any instrumentation the payload will include information about it.

Key	Value
`:exception`	An array of two elements. Exception class name and the message
`:exception_object`	The exception object

Creating custom events

Adding your own events is easy as well. `ActiveSupport::Notifications` will take care of all the heavy lifting for you. Simply call `instrument` with a `name`, `payload` and a block. The notification will be sent after the block returns. `ActiveSupport` will generate the start and end times and add the instrumenter's unique ID. All data passed into the `instrument` call will make it into the payload.

Here's an example:

```
ActiveSupport::Notifications.instrument "my.custom.event", this: :data do
  # do your custom stuff here
end
```

Now you can listen to this event with:

```
ActiveSupport::Notifications.subscribe "my.custom.event" do |name, started,
finished, unique_id, data|
  puts data.inspect # {:this=>:data}
end
```

You also have the option to call instrument without passing a block. This lets you leverage the instrumentation infrastructure for other messaging uses.

```
ActiveSupport::Notifications.instrument "my.custom.event", this: :data

ActiveSupport::Notifications.subscribe "my.custom.event" do |name, started,
finished, unique_id, data|
```

```
  puts data.inspect # {:this=>:data}
end
```

You should follow Rails conventions when defining your own events. The format is: `event.library`. If your application is sending Tweets, you should create an event named `tweet.twitter`.