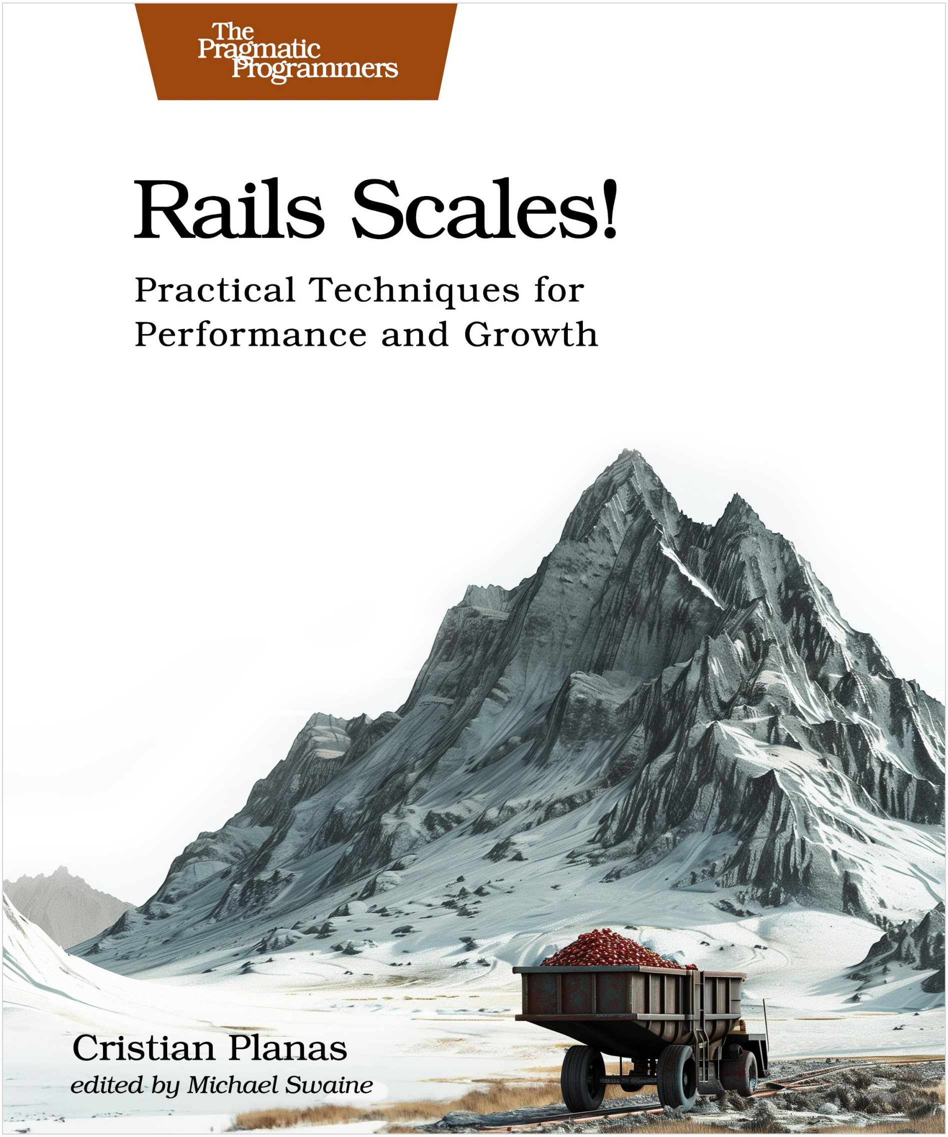


The Pragmatic Programmers

Rails Scales!

Practical Techniques for Performance and Growth

Cristian Planas
edited by Michael Swaine

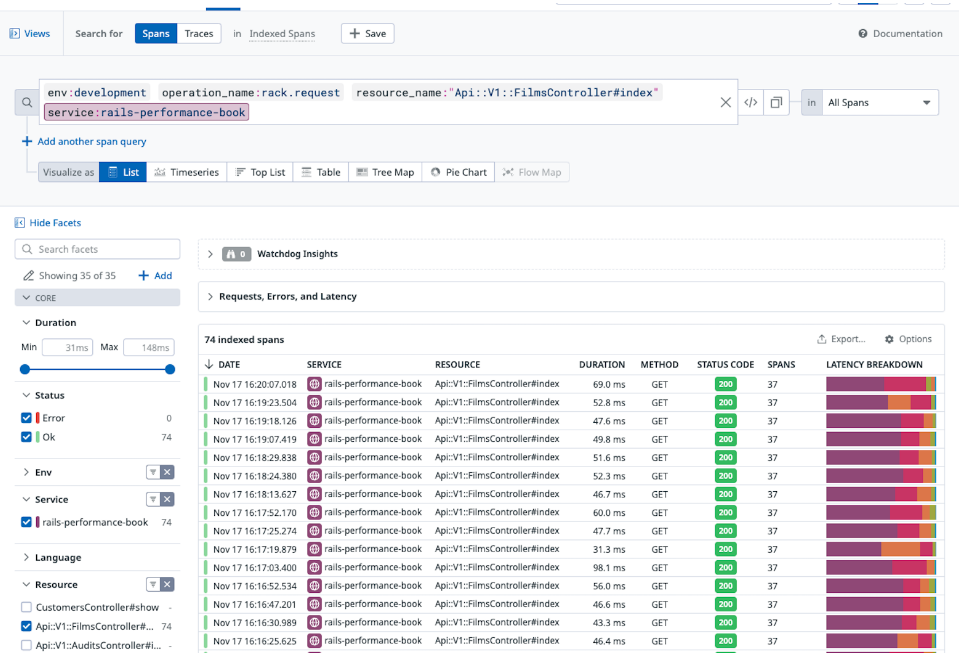
This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Using Traces

The previous dashboards, based on default metrics on request latency will be very useful to understand the “macro” tendencies of your application, even the tendencies in a particular endpoint. However, sometimes it’s necessary to “zoom in”: traces will be the tool to use when you need to understand the details of what is happening with certain requests. You can think of a trace as a kind of “breadcrumb trail” left by each request in their journey through your application. Every time a request hits your application, markers will be generated for the steps taken. With them, a complete storyline of the life of the request in your application can be generated. Using traces, you will be able to investigate slowdowns by examining closely each of the steps that a slow request has followed, to understand what happened and what needs to be optimized.

You can access traces in Datadog in multiple ways. If you are following from the previous section, you can just scroll down in the “Resource Page” for `Api::V1::FilmsController#index`: the `Traces` section is at the bottom. Here you can already see information about each request that hit the action. To see even more, click on the “View all in Trace Explorer” link. You can also access the Trace Explorer in the left sidebar, under APM.



```
env:development  operation_name:rack.request  resource_name:"Api::V1::FilmsController#index"
service:rails-performance-book
```

74 indexed spans

DATE	SERVICE	RESOURCE	DURATION	METHOD	STATUS CODE	SPANS	LATENCY BREAKDOWN
Nov 17 16:20:07.018	rails-performance-book	Api::V1::FilmsController#index	69.0 ms	GET	200	37	
Nov 17 16:19:23.504	rails-performance-book	Api::V1::FilmsController#index	52.8 ms	GET	200	37	
Nov 17 16:19:18.126	rails-performance-book	Api::V1::FilmsController#index	47.6 ms	GET	200	37	
Nov 17 16:19:07.419	rails-performance-book	Api::V1::FilmsController#index	49.8 ms	GET	200	37	
Nov 17 16:18:29.838	rails-performance-book	Api::V1::FilmsController#index	51.6 ms	GET	200	37	
Nov 17 16:18:24.380	rails-performance-book	Api::V1::FilmsController#index	52.3 ms	GET	200	37	
Nov 17 16:18:13.627	rails-performance-book	Api::V1::FilmsController#index	46.7 ms	GET	200	37	
Nov 17 16:17:52.170	rails-performance-book	Api::V1::FilmsController#index	60.0 ms	GET	200	37	
Nov 17 16:17:25.274	rails-performance-book	Api::V1::FilmsController#index	47.7 ms	GET	200	37	
Nov 17 16:17:19.879	rails-performance-book	Api::V1::FilmsController#index	31.3 ms	GET	200	37	
Nov 17 16:17:03.400	rails-performance-book	Api::V1::FilmsController#index	98.1 ms	GET	200	37	
Nov 17 16:16:52.534	rails-performance-book	Api::V1::FilmsController#index	56.0 ms	GET	200	37	
Nov 17 16:16:47.201	rails-performance-book	Api::V1::FilmsController#index	46.6 ms	GET	200	37	
Nov 17 16:16:30.989	rails-performance-book	Api::V1::FilmsController#index	43.3 ms	GET	200	37	
Nov 17 16:16:25.625	rails-performance-book	Api::V1::FilmsController#index	46.4 ms	GET	200	37	

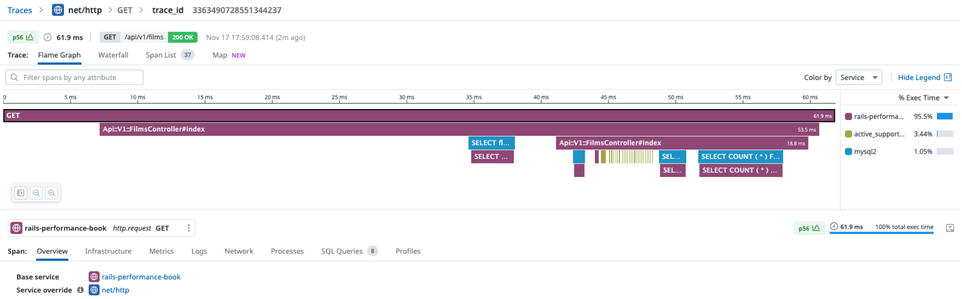
First of all, note that at the top of the page, the Trace Explorer is marking as if you were filtering by “Spans”. While a trace is the entire journey of a request, a span is an individual unit within the trace, typically representing a single operation. A span can contain other spans, so you can "mark" different parts of the request in your application with the target of monitoring them better.

At the top of the Trace Explorer, you will see a search bar showing the current filters you are applying. These should be:

- env:development. Right now you are running the application in development mode.
- operation_name:rack_request. The operation being monitored: in this case, the request that is being processed by Rack.
- service:rails-performance-book. The service that is being monitored.
- resource_name:"Api::V1::FilmsController#index". The resource that we are filtering by: in this case, it's a combination of the controller (Api::V1::FilmsController) and the action(#index).

On the left side of the screen, you will see a list of facets. With those facets, you can change the filters currently applied to your trace search. For example, if you want to check your slowest requests, you can change the "Duration" filter to only see the ones that took over 100ms, for example. You can also check a different resource by changing the selection under "Resource": try to check the traces for CustomersController#show.

Now, let's dive deeper into one particular trace. Check again the requests to Api::V1::FilmsController#index. On the right side, you will get a list of requests that hit that particular action. Click on one of them, and a new panel full of information for that specific request will be displayed: feel free to click on "Open Full Page" to check this more comfortably.



Traces > net/http > GET > trace_id 3363490728551344237

p56 | 61.9 ms | GET /api/v1/films 200 OK | Nov 17 17:59:08.414 (2m ago)

Trace: Flame Graph | Waterfall | Span List 37 | Map NEW

Filter spans by any attribute

Color by: Service | Hide Legend

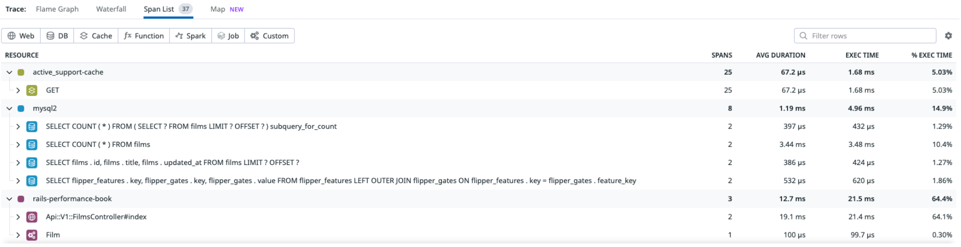
rails-performance-book http.request GET

p56 | 61.9 ms | 100% total exec time

Span: Overview | Infrastructure | Metrics | Logs | Network | Processes | SQL Queries 8 | Profiles

Base service: rails-performance-book
Service override: net/http

The first section is a flame graph. It represents the execution path of the request across multiple services (like MySQL or the cache layer). If you hover over some of the horizontal bars, you will see more granular information on all the steps executed during the request. For example, in the trace above, there was a check on a feature flag in DB (SELECT … FROM flipper_features …) that took 3.8ms. Later, there was a query on the films table that took 922 microseconds. Further down the line, there is a set of accesses to active_support-cache; each one took around 70 microseconds: those were accesses to fetch the cached JSONs corresponding to each object returned by the requests. Finally, there was a kind of big query at the end of the flame graph: SELECT COUNT (*) FROM films taking 6.33ms. If I wanted to optimize this endpoint further, I would take a look at improving that. Apart from "Flame Graph", DataDog offers you other ways to visualize the spans that make up the trace: "Span List" can also be quite useful to see data in a less visual, but more textual, way.



RESOURCE	SPANS	AVG DURATION	EXEC TIME	% EXEC TIME
active_support-cache	25	67.2 µs	1.68 ms	5.03%
GET	25	67.2 µs	1.68 ms	5.03%
mysql2	8	1.19 ms	4.96 ms	14.9%
SELECT COUNT (*) FROM (SELECT ? FROM films LIMIT ? OFFSET ?) subquery_for_count	2	397 µs	432 µs	1.29%
SELECT COUNT (*) FROM films	2	3.44 ms	3.48 ms	10.4%
SELECT films . id, films . title, films . updated_at FROM films LIMIT ? OFFSET ?	2	386 µs	424 µs	1.27%
SELECT flipper_features . key, flipper_gates . key, flipper_gates . value FROM flipper_features LEFT OUTER JOIN flipper_gates ON flipper_features . key = flipper_gates . feature_key	2	532 µs	620 µs	1.86%
rails-performance-book	3	12.7 ms	21.5 ms	64.4%
Api::V1::FilmsController#index	2	19.1 ms	21.4 ms	64.1%
Film	1	100 µs	99.7 µs	0.30%

Under the flame graph (or the span list), you can see a list of information corresponding to the trace. In particular, the most interesting can be the list under "Span Attributes". You can see all kinds of data. Under http, you will see the method (GET), the status code of the response (200), and the path (api/v1/films). There is more low-level information in attributes, like the process ID, the language of the service, etc. All this is automatically generated by Datadog's integration, and sometimes, it can be insufficient to properly understand what is happening in your application. Fortunately, you can customize traces further.

Customizing Traces

Traces and spans are highly customizable, so you can get exactly the information you need. In this section, you are going to customize the monitoring by adding two new elements:

- A new span attribute that will be added every time that a user hits a request associated with a store, like (api/v1/stores/STORE_ID/audits). Adding this attribute can be crucial: if the application you have developed in this book is like a B2B, the store would be something like our customer. Adding the store ID as an attribute will allow you to filter requests by customer, which is basic to diagnose issues happening in one specific account.
- A new service that will track the processes performed by the presenter layer. This is a way to separate the work done by the database from the

> one done by parts of our Rails application. Moreover, it's a clean example of how to further segregate the tasks executed by the application, so it does not become an untraceable blob.

Let's start by adding a new attribute to the current span. To do so, I recommend you create a new middleware in your application. If you did the "Discovering Sharding" section in the "Thinking Architecture for Performance" chapter, you can inspire yourself with the ShardSwitcher middleware you created then. That middleware could look something like this:

```ruby
module Middleware
  class DatadogMiddleware
    def initialize(app)
      @app = app
    end

    def call(env)
      request = Rack::Request.new(env)
      request.path =~ /.*stores\/(\d+).*/
      store_id = $1

      Datadog::Tracing.active_span.set_tag("store_id", store_id) if store_id
      @app.call(env)
    end
  end
end
```

The key methods here are the call to Datadog::Tracing.active_span, which returns the current span, and the call to set_tag(key, value) which sets the tag. Remember also to add the middleware to your application configuration in config/application.rb:

```ruby
require_relative '../app/middleware/shard_switcher'
module Moviestore
  class Application < Rails::Application
    [...]

    config.middleware.use Middleware::DatadogMiddleware

    [...]
  end
end
```

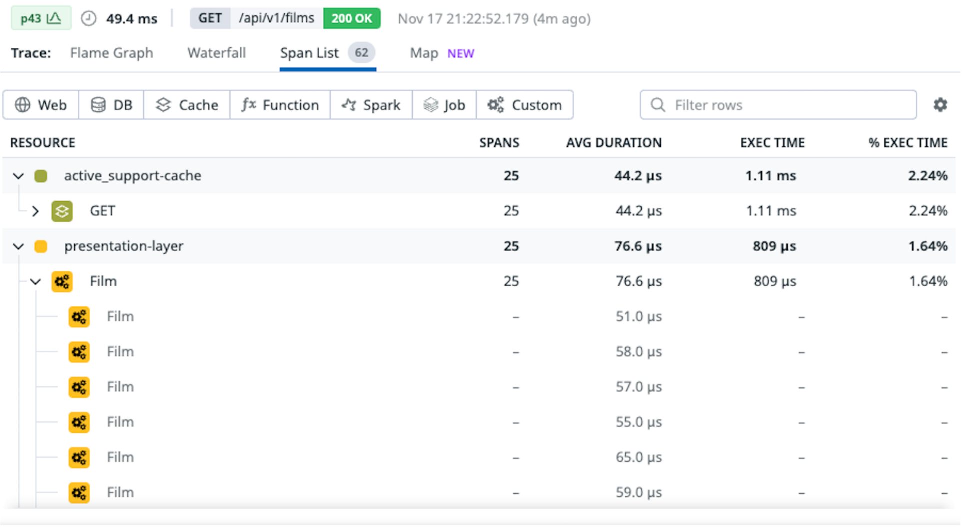
With this done, restart your Rails application and test it out. Restart the rake seed_datadog task, or hit by yourself an endpoint with a store_id parameter, like api/v1/stores/1 or api/v1/stores/1/audits. Now go back to the Trace Explorer in your Datadog instance and check a trace for resources associated with a store: Api::V1::StoresController#show or Api::V1::AuditsController#index. Check the Span Attributes. At the bottom, you will see your new attribute, store_id. If you hover

over it, an options menu should appear, and you should be able to "Filter by &store_id:X", selecting only traces associated with the desired store.

Next, you will create a new service to encapsulate all the presenter logic. Creating a new service in DataDog is simple: you just need to add new traces specifying this new service (as a string parameter). To create a new span, you will need to call the method Datadog::Tracing.trace(trace_name) and pass the logic that makes up the trace as a block. Let's apply this to the presentation layer. Fortunately, all the presentation classes share the same parent class: Api::V1::Presenter. Moreover, this class has only one method (beyond initialize): to_json. This simplifies our task a lot: you just need to wrap all the logic around that method with a new trace. This is a possible solution:

```
# app/presenter/api/v1/presenter.rb

class Api::V1::Presenter
  [...]

  def to_json(exclude: [])
    return nil unless resource
    Datadog::Tracing.trace('presenter.to_json',
      service: 'presentation-layer', resource: resource&.class&.to_s) do

      [...]

    end
  end
end
```

Note that I also added the class of the object presented as the resource of the trace. Once you have created this new trace, hit a few times an endpoint that uses the Presenter class (for example, Api::V1::FilmsController#index). Check the span list of any of its traces. You shall see a new service, presentation-layer.



RESOURCE	SPANS	AVG DURATION	EXEC TIME	% EXEC TIME
active_support-cache	**25**	**44.2 µs**	**1.11 ms**	**2.24%**
GET	25	44.2 µs	1.11 ms	2.24%
presentation-layer	**25**	**76.6 µs**	**809 µs**	**1.64%**
Film	25	76.6 µs	809 µs	1.64%
Film	–	51.0 µs	–	–
Film	–	58.0 µs	–	–
Film	–	57.0 µs	–	–
Film	–	55.0 µs	–	–
Film	–	65.0 µs	–	–
Film	–	59.0 µs	–	–

The new service is also available in the Service Catalog and all other DataDog features. Remember that you can learn more about how to customize your traces by reading DataDog's documentation for its Ruby integration.[5]

[5] https://docs.datadoghq.com/tracing/trace_collection/automatic_instrumentation/dd_libraries/ruby/#integration-instrumentation