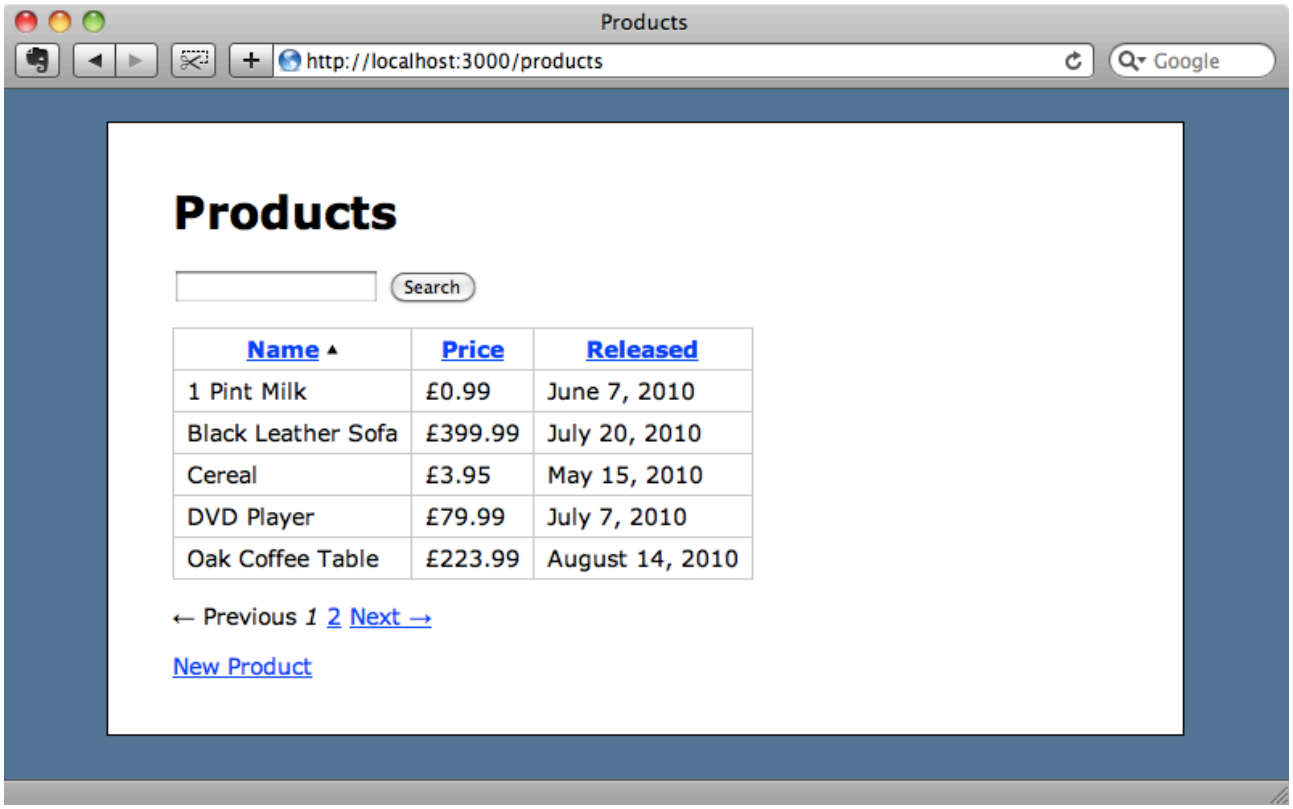
Episode 249

Notifications in Rails 3

Below is a screenshot from a small Rails application that shows a list of products that can be sorted and paginated. In order to see how responsive this application is we want to see how long each request is taking to load and store this information in the database so that we can calculate some metrics about the application's performance.



There are a number of ways that we could gather this information. One solution would be to parse the application's log file and extract the information from there. If we wanted to calculate average load times or each page or other similar statistics it would be much easier to do this if the data was stored in a database. We could, of course, use a third-party tool such as NewRelic's RPM[1] but in this case we want something really simple that just records the amount of time that each request takes.

[1] http://newrelic.com/

Notifications

We'll implement this by using a feature that was introduced in Rails 3: `ActiveSupport::Notifications`[2]. We can use notifications to create instruments. An instrument takes a block of code and when the block finishes executing it will send out a notification event. Any notifications that are subscribed to this event will then be triggered. A typical request in a Rails 3 application sends out several of these notifications and we can subscribe to any of them. Before we write our own instrument let's try subscribing to some of the built-in ones to get a feel for how they work.

The first thing we'll need to do is create a new file in our application's `/config/initalizers` directory that we'll call `notifications.rb` and in this file we'll write the code to subscribe to some notifications. To do this we need to call `ActiveSupport::Notifications.subscribe`. If we pass an argument to this method we can filter the notifications that we subscribe to. We want to see all of the notifications that our application is sending out for each request so we won't pass one in. The method also takes a block which takes five arguments: the `name` of the notification, a `start` and `finish` time, an `id` and finally a `payload` which contains a hash of information about the notification. We'll take this information and send it out to the debugger.

`/config/initializers/notifications.rb`

```
ActiveSupport::Notifications.subscribe do |name, start, ↵
  finish, id, payload|
  Rails.logger.debug(["notification:", name, start, finish, ↵
    id, payload].join(" "))
end
```

If we start up the Rails server now and reload the application's home page we'll see the notifications that were made during that request. Here is the first one.

---

[2] http://api.rubyonrails.org/classes/ActiveSupport/Notifications.html

```
notification: start_processing.action_controller 2011-01-19
21:51:40 +0000 2011-01-19 21:51:40 +0000 bac10ab8b439502ce5ba
{:controller=>"ProductsController", :action=>"index", :params=>
{"controller"=>"products", "action"=>"index"}, :formats=>
[:html], :method=>"GET", :path=>"/products"}
```

In the notification above we can see the parameters that were passed in: the `name`, `start` and `finish` times, the id and finally the `payload` hash. The notifications that were fired during this request were:

```
start_processing.action_controller
notification: sql.active_record
notification: sql.active_record
notification: sql.active_record
notification: !render_template.action_view
notification: !render_template.action_view
notification: render_template.action_view
notification: process_action.action_controller
```

The notifications listed include ones for ActiveRecord queries which means that we could subscribe to those to see how long each database query takes. The final one is the one we're going to focus on as it fires when the request has completed and so we can use it to determine how long the request took to process.

```
notification: process_action.action_controller 2011-01-19 21:51:40
+0000 2011-01-19 21:51:41 +0000 bac10ab8b439502ce5ba
{:controller=>"ProductsController", :action=>"index", :params=>
{"controller"=>"products", "action"=>"index"}, :formats=>
[:html], :method=>"GET", :path=>"/
products", :status=>200, :view_runtime=>208.36210250854492,
:db_runtime=>80.30999999999999}
```

This notification includes other useful information such as the `params`, the `path`, the `status`, the `view_runtime` and the `db_runtime`. We'll make use of some of these later.

We want to track this information in the database so that we can work out which pages are running slowly. If we were using this in a production application then it would be better to use a memory-based store which would be more efficient but for this application we'll store the data in a normal model.

We have Ryan Bates' Nifty Generators[3] installed in our application so we'll use the `nifty:scaffold` generator to create the model, along with an associated controller and view. The model will have a `path` field to store the path and three `duration` fields for the durations of various parts of the request.

```
$ rails g nifty:scaffold page_request rails g nifty:scaffold
page_request path:string page_duration:float view_duration:float
db_duration:float index
```

When the command completes we'll to migrate the database to create the new table.

```
$ rake db:migrate
```

We now have a model in which we can store information about each request. We'll need to note the name of the notification, `process_action.action_controller`, so that we can subscribe to just that notification. We'll also need to note the names of the fields in the payload we want to extract information from.

Back in our `notifications.rb` file we'll pass an argument to the call to `ActiveSupport::Notifications.support` so that we can limit the notifications we subscribe to. One way to do this is to pass in a regular expression so to subscribe to all notifications that contain `action_controller` in their name we can do the following:

/config/initializers/notifications.rb

```
ActiveSupport::Notifications.subscribe /action_controller/ do ↵
  |name, start, finish, id, payload|
  Rails.logger.debug(["notification:", name, start, finish, ↵
    id, payload].join(" "))
end
```

Alternatively we can pass in a string to match a single notification and that's what we'll do. Instead of just logging the notification we'll create a new `PageRequest` and save it to the database.

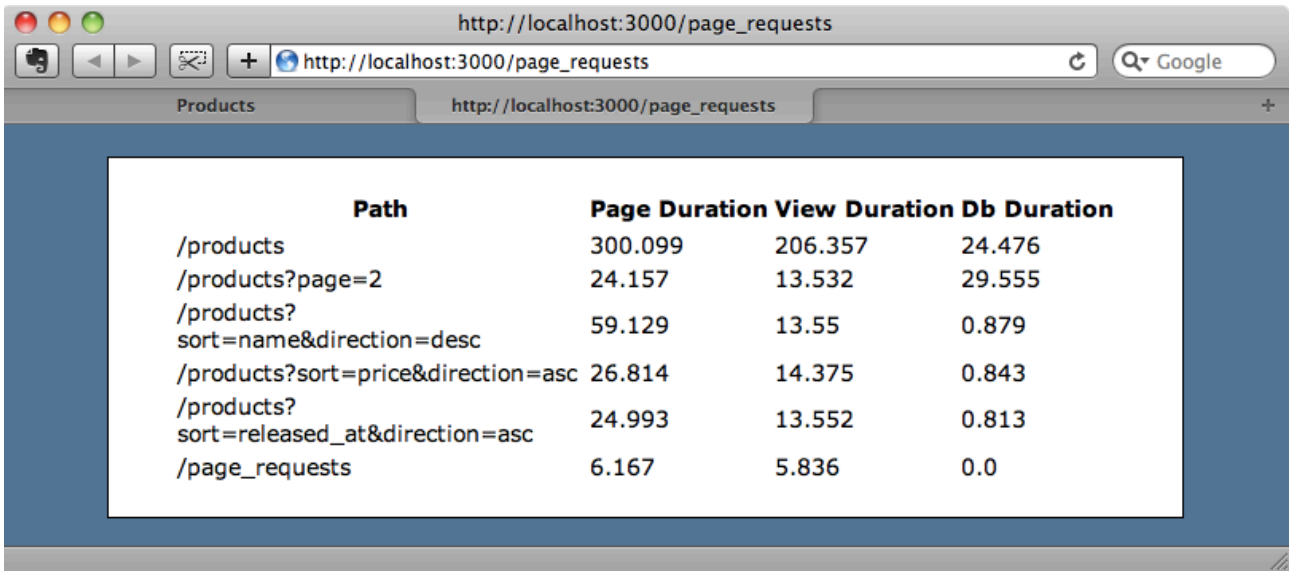
---

[3] https://github.com/ryanb/nifty-generators

/config/initializers/notifications.rb

```
ActiveSupport::Notifications.subscribe
"process_action.action_controller" do |name, start, finish, id,
payload|
  PageRequest.create! do |page_request|
    page_request.path = payload[:path]
    page_request.page_duration = (finish - start) * 1000
    page_request.view_duration = payload[:view_runtime]
    page_request.db_duration = payload[:db_runtime]
  end
end
```

If we restart our Rails server, visit a few pages and then visit `/page_requests` we'll see the list of page requests and the durations for each page that are stored in the database. We can use this information to work out which pages are the slowest, draw graphs to see performance over time and so on.



Path	Page Duration	View Duration	Db Duration
/products	300.099	206.357	24.476
/products?page=2	24.157	13.532	29.555
/products?sort=name&direction=desc	59.129	13.55	0.879
/products?sort=price&direction=asc	26.814	14.375	0.843
/products?sort=released_at&direction=asc	24.993	13.552	0.813
/page_requests	6.167	5.836	0.0

Creating Notifications

The notification events that Rails sends internally are primarily used for metrics which is how we're using them here. We can use notifications in our own applications for any purpose and we'll create an instrument in our store application to show this.

Our application allows users to search for products and we want to log the search terms that are entered. In our `Product` model we have a `search` class method

which is where the search takes place and which is the ideal place to log the search terms. We don't really want our model to be burdened with logging search terms so instead we'll raise a notification in this class that contains the information about this search and which we can then subscribe to anywhere else in our application's code.

To create a notification we have to create a new instrument and we do that by calling `ActiveSupport::Notifications.instrument`. This takes two arguments: the first is the name that we want to give the notification, the second can be anything we want and will passed to the payload. The second argument is usually a hash of arguments and for this notification it will be the search term.

/app/models/product.rb

```
class Product < ActiveRecord::Base
  def self.search(search)
    if search
      ActiveSupport::Notifications.instrument ↵
        ("products.search", :search => search)
      where('name LIKE ?', "%#{search}%")
    else
      scoped
    end
  end
end
```

If we pass a block to `instrument` then the start and finish times passed to the notification will be the time that the code in the block began and finished executing. As we're using scopes here that information isn't very valuable as the database query doesn't take place in the `search` method so we won't do that. This will mean that the start and finish times will be the same.

In our notifications initializer file we can subscribe to our new notification and handle it however we want. As an example we'll log the search term to the log file.

/config/initializers/notifications.rb

```
ActiveSupport::Notifications.subscribe "products.search"  do ↵
  |name, start, finish, id, payload|
  Rails.logger.debug "SEARCH: #{payload[:search]}"
end
```

We subscribe to our own notification the same way we do to one of Rails built-in notifications and we can read the search term from the payload and pass it to the log message. If we restart our server now and search for “milk” we’ll see the search term logged.

```
SEARCH: milk
```

Of course we could create a database table to store this information in the same way we did with the metric information earlier.

That’s how we can use notifications in Rails 3 to move logic outside of a class and handle it elsewhere. We can add as many subscribers as we like to handle a notification in separate places. We need to be careful with the notification pattern in our applications, however as it can be overused. For example lets say that we have a `User` model in our application and we want to send an email when a new user registers. We could use notifications here but it’s not a good use of them. Notifications are best used for logging and for code that doesn’t modify the application’s core behaviour. When using notifications there is a risk of scattering the application’s core logic into various places. Anything can listen to a notification and so the logic could be anywhere in the application’s code. This can make the code much harder to read and debug so it really is best to only use notifications for things that aren’t part of your application’s core logic.

That’s it for this episode. Be sure to check out the core documentation for further information on notifications.