The Functional Web

How Functional Programming Concepts Could Improve Rails

Who I am

- Sean Griffin
- Software Developer at thoughtbot
- Twitter: @sgrif
- Github: sgrif
- Email: sean@thoughtbot.com

Rails has to evolve

New languages and frameworks crop up every day, each with new ideas on how to solve complex problems. Many times these ideas can be brought back to Rails as gems or libraries, but some concepts are difficult or impossible to bring to Rails without fundamentally changing the framework.

Pain Point:
Concurrency

What is concurrency?

What does concurrency look like in Rails today?

```
gem ‘unicorn’

gem ‘delayed_job’
```

The easiest way to achieve some level of concurrency in a Rails app is to use a concurrent web server. For MRI apps deployed on Heroku, most use Unicorn, which creates multiple processes that can each handle one request at a time.

The next step we can take is to move processes that don't need to happen to serve a response off of the web server. For example, we don't need to send the user a welcome email to them to the next page after signing up. Instead we place this into a job queue, for another process to pick up later.

This is only a solution
for the simplest case

Why isn’t this a solution?

- Assumes the entire application has the same needs
- Assumes 100% of time is being spent doing work on the server

This is the biggest issue. Being concurrent at the server level, and ignoring it everywhere else only works if you're spending the majority of time running Ruby code. In most applications, that simply isn't the case.

But Ruby can’t run concurrently!

At this point, when I talk about concurrency, I'm talking about threads. If we want to do multiple things in our application, and work with the results of the concurrent operations, threads are the only option. In MRI, we have what's called the GIL which prevents multiple threads from executing Ruby code at the same time.

Things you can do concurrently in Ruby

- IO

Things that should always be done concurrently

- IO

IO is fundamental to web applications. You spend time in IO when you are sending the response to the client, or when you connect to another service within your application. At minimum, you'll be interacting with a database service, and a cache service. That's IO. If you're interacting with a third party API, that's IO.

Why would I want concurrency inside my controller actions?

In complex applications, we often have several operations that we need to perform which are sometimes, but not always, dependent on the results of the others. When that operation is a simple calculation in our application, the overhead of creating threads for each unrelated calculation is often not worth it, but the one case where it's always guaranteed to be worth the effort is IO.

Let's say for example we want to access a protected file for a resource. Both the file, the resource, and the user credentials all come from a third party API. In order to get the file, we need to exchange the user credentials for a token, and get details about the resource, but the first two operations can be done at the same time, as they're not dependent.

Let's look at what this might look like with threads in Ruby.

```
class ProtectedFileController < ApplicationController
  def show
    auth_token = load_auth_token(current_user.third_party_id)
    resource_details = load_resource_details(params[:id])
    file = load_file_data(auth_token, resource_details)
    send_data file.data, type: file.content_type
  end
end
```

```
def file
  auth_token = load_auth_token(current_user.third_party_id)
  resource_details = load_resource_details(params[:id])
  load_file_data(auth_token, resource_details)
end
```

```ruby
def file
  auth_token_thread = Thread.new do
    Thread.current[:output] =
      load_auth_token(current_user.third_party_id)
  end

  resource_details_thread = Thread.new do
    Thread.current[:output] =
      load_resource_details(params[:id])
  end

  auth_token_thread.join
  resource_details_thread.join
  auth_token = auth_token_thread[:output]
  resource_details = resource_details_thread[:output]

  load_file_data(auth_token, resource_details)
end
```

This is what the implementation might look like in Ruby, and it's a mess. We have to concern ourselves with the dispatching of various threads, and use thread local variables to get the results of each computation out.

The biggest issue with this code is how far removed it is from what we're trying to do, and this isn't something that could easily be refactored into something more readable.

Threads force us to focus on order of execution

The code is more concerned with the semantics of threads, and order of execution than what we're actually trying to accomplish.

In some languages, we use a monad called a Future or Promise to represent the result of a single asynchronous operation. Let's look at what loading the file might look like in Scala.

```
def file = {
  val authToken = loadAuthToken(currentUser.thirdPartyId)
  val resourceDetails = loadResourceDetails(params.id)
  loadFileData(authToken, resourceDetails)
}
```

```
def file = {
  val authToken = future { loadAuthToken(currentUser.thirdPartyId) }
  val resourceDetails = future { loadResourceDetails(params.id) }
  authToken.zip(resourceDetails).map(loadFileData)
}
```

This is much cleaner. If you've ever used a promise in JavaScript, Scala Futures are basically the same idea, you can just do more with them. By wrapping our call to `loadAuthToken` in `future`, we get back a `Future[AuthToken]` instead of an `AuthToken`.

When we're working with futures, the easiest way to think about them is as an array with a single item inside of it. So if we want to do something with the result, we call `map`. `zip` combines our first two futures into a new future representing the result of both.

```
def authToken = future { loadAuthToken(currentUser.thirdPartyId) }

def resourceDetails = future { loadResourceDetails(params.id) }

def file = authToken.zip(resourceDetails).map(loadFileData)
```

By using futures, it’s also trivial to break this out into separate instance methods. Since our `file` method is no longer concerned with the semantics of forking and joining threads, or how to get the result back out, each line can become a separate method with no additional work.

Futures allow us to concern ourselves with transformations and relationships between data

However, if we tried to use a similar concept in Rails, we'd run into a major problem. Once you've wrapped one value in a future, everything that relies on the result of that operation will also end up being wrapped in a future. However, Rails assumes that we're done with our operation as soon as our action returns, which means we couldn't call `render` or `send_file` in the result of a future, since that would happen asynchronously.

As a stop gap measure, we could change our code to do something like this:

```
def authToken = future { loadAuthToken(currentUser.thirdPartyId) }

def resourceDetails = future { loadResourceDetails(params.id) }

def futureFile = authToken.zip(resourceDetails).map(loadFileData)

def file = Await.result(futureFile, 30 seconds)
```

This would work, but it's ugly, and forces us to once again litter our code with boilerplate concerned only with order of execution. We've also added one more place to introduce a bug. What if I forget that I need to await the result any time I use a future? What if I choose the wrong timeout duration. We've also littered our code with concerns that have nothing to do with our application.
I chose that 30 second timeout, as it's the maximum duration for a connection to respond on Heroku, something our controller should *definitely* have no idea about.

How do other
languages do it?

Scala Play Controllers

Scala is an OO-Functional hybrid language that runs on the JVM. The Play Framework is a popular MVC Web Framework that is inspired by Rails. One thing it handles amazingly is concurrency. The simplest Controller in play looks something like this:

Hello World

```
object HelloController extends Controller {
  def hello = Action {
    Ok("Hello, World!")
  }
}
```

The keyword `object` specifies a singleton in Scala. It's kind of like having a class that you can't call `new` on. You just specify it by name, and call methods on it directly. Our hello method is calling `Action` and passing it a block. An `Action` in Scala has a signature that looks like this:

```
Action.apply: (Request) ⇒ Result
```

So an `Action` gets passed a function that takes a `Request` and returns a `Result`. A `Result` equates to an HTTP response. If you've done much Ruby programming, this probably sounds pretty similar to the interface for a `Rack` application. (Rack is what Rails uses under the hood to hook up to a server). The signature for a Rack application looks like this:

```
Rack::Application.call:
(Map[String, String]) ⇒
(Int, Map[String, String], String)
```

In Rack, the argument to the application is called the environment, but it's just a hash containing the request data. The application has to return an array containing 3 items, the response code, the response headers, and the response body. In ruby code, this means an integer, a hash, and a string.

These are accomplishing essentially the same thing, Play's API is just using objects to represent the concepts rather than primitives. There's one thing that Play's API can do that Rack can't, however. If we want to perform an action concurrently, all we need to do is this

```
object HelloController extends Controller {
  def helloAsync = Action.async {
    future {
      Ok("Hello, Asynchronous World!")
    }
  }
}
```

This is a contrived example, as we've just arbitrarily said go process this on another thread. However, what's important here is that the type signature just changed.

We've changed from

`(Request) ⇒ Result`

to

`(Request) ⇒ Future[Result]`

If you wrap any piece of code in a call to `future`, it will run that code on a separate thread, and return a future of whatever the result is.

What's extremely important is that Scala will *let* you return a `Future[Result]`. You're *allowed* to say, "Hey, I need to go do a couple of things before I can serve this response, so I can't give you anything right now. You shouldn't block while you wait for me, but I'll let you know when I'm finished."

```
object PiController extends Controller {
  def pi = Action.async {
    val futurePi = computePiAsynchronously
    futurePi.map { pi =>
      Ok("Pi is " + pi.toString)
    }
  }
}
```

When looking at `Future`'s interface, the easiest way to think of it is like it's an array with a single item. So when we call map on a `Future[Int]`, and pass map an `Int ⇒ Result`, we get back a `Future[Result]`

How can we bring this to Rails?

So let's swing back to Rails, and how we could have the same concept in our Rails controllers. We just need to implement Futures, and have Rails start handling it, right?

Our API is too far removed from HTTP

In Rails, we don't take a request and return a response. We don't actually return anything. If you look into `action_controller`'s internals, the method that is ultimately responsible for calling our actions is this

```ruby
# actionpack/lib/action_controller/metal.rb
def dispatch(name, request) #:nodoc:
  @_request = request
  @_env = request.env
  @_env['action_controller.instance'] = self
  process(name)
  to_a
end
```

Our API is built around mutability. If you call `render`, it's going to mutate the `response_body` property of our controller. The way we signal to Rails that we're done is by returning from our action. This doesn't work if the actual construction of our response is done asynchronously.

So what can we do? How do we signal that we're actually ready to send the response? We could go back to the blocking solution presented earlier, but that's asking for trouble. The issue with that solution and our current API is that if we forgot to block, we wouldn't get an error. Rails would do what it normally does if you don't call `render` or an equivalent function. It'll assume you want to call `render` with the name of the controller and action.

We could modify methods like `render` and `redirect_to` to indicate that we're ready to send the response, but this will break the existing API. We can no longer implicitly render a view at the end of a response, because not calling

We will break Rack

Even if we solve this problem for Rails, we're going to run into an identical problem with Rack. Rack's API assumes a single synchronous standard HTTP response, and we have a ton of middleware built on that API. The only solution would be to add a piece of middleware that blocks until the response is finished processing, and then send it up the stack. While this is slightly better than handling it in the application, we end up duplicating configuration, and adding code for things that really should be handled at the bottom of the stack, the server level, not the top.

Doesn’t Play have the
same problem?

```
object Action {
  def apply(f: (Request) => Result) = {
    async { (request) =>
      future { f(request) }
    }
  }
}
```

What should we do?

If we go down this road, whatever decisions we make will fundamentally change how we work from this point forward. We shouldn't take this lightly, and we need to do it right. We don't want to be having the same conversation in another 8 years.