Miles more maintainable: Building APIs with the middleware pattern

I'm @timrogers 💁🏼‍♂️

My name is Tim.

I’m from London 🇬🇧☔️

And as you might be able to tell from my accent, I’m from London, in the rainy United Kingdom.

I'm a Software Engineer at GoCardless

I'm a software engineer at a fintech startup called GoCardless.

We're building an API-driven bank-to-bank payments network for the internet

So to give you some context, let me tell you what we do at GoCardless.

We're building an API-driven bank to bank payments network for the internet. What does this mean?

Well - we're all familiar with the big credit card networks: Visa, MasterCard and American Express.

DIRECT Debit

bankgirot

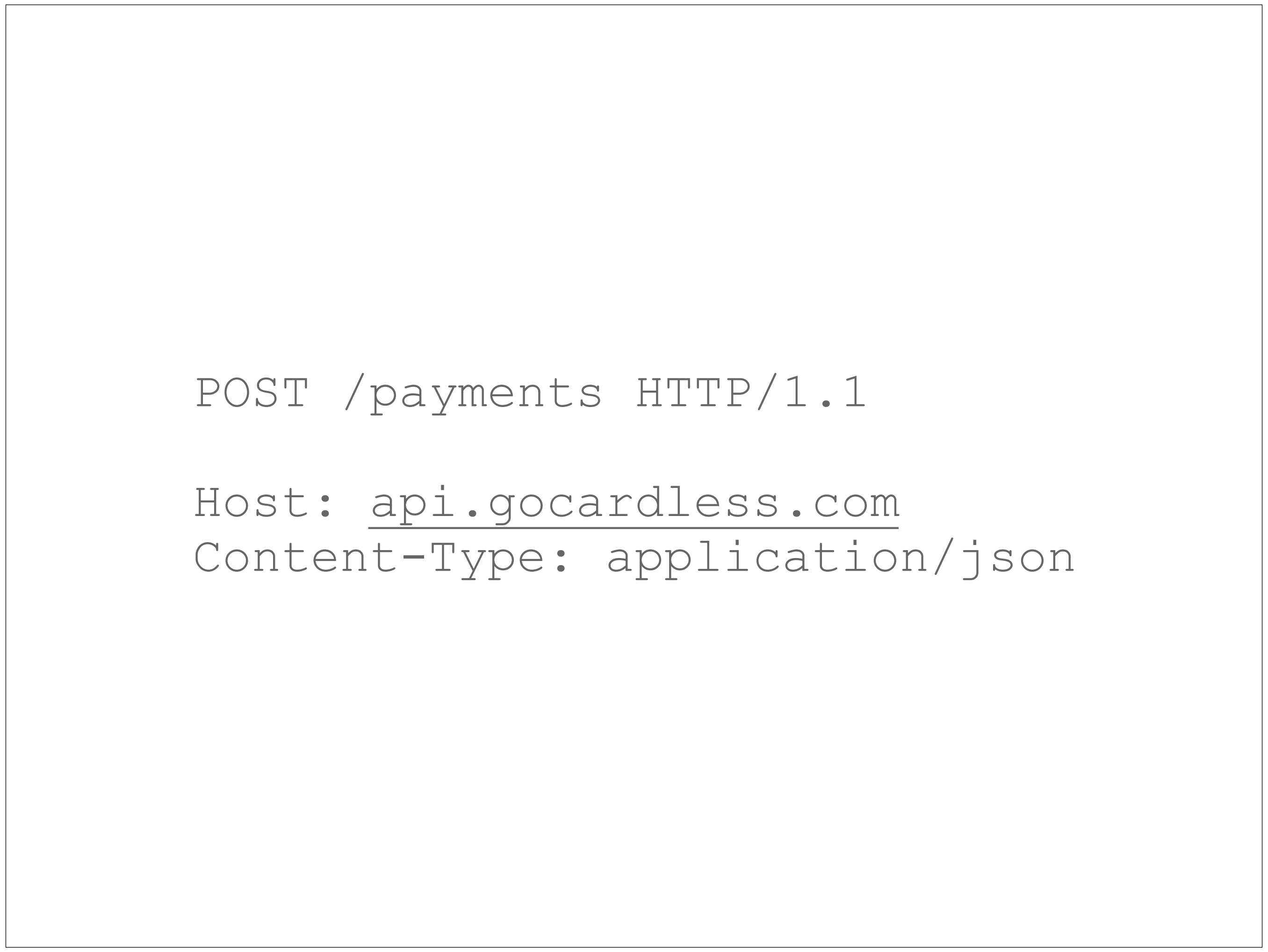
SEPA
Single Euro Payments Area

Alongside those, there are alternative, bank to bank payment networks. The credit card system has lots of intermediaries taking a cut, whereas these bank to bank systems just involve pulling money from one bank account, and dropping it in another.

Most of these systems are country-specific, like Direct Debit in the UK or Autogiro in Sweden. For the Eurozone there's a cross-border system called SEPA Direct Debit.

These systems are usually cheaper than cards, and people's bank accounts don't expire. This means less churn and lower costs.

GoCardless aims to bridge these local networks together, providing a global solution for bank-to-bank payments.



```
POST /payments HTTP/1.1

Host: api.gocardless.com
Content-Type: application/json
```

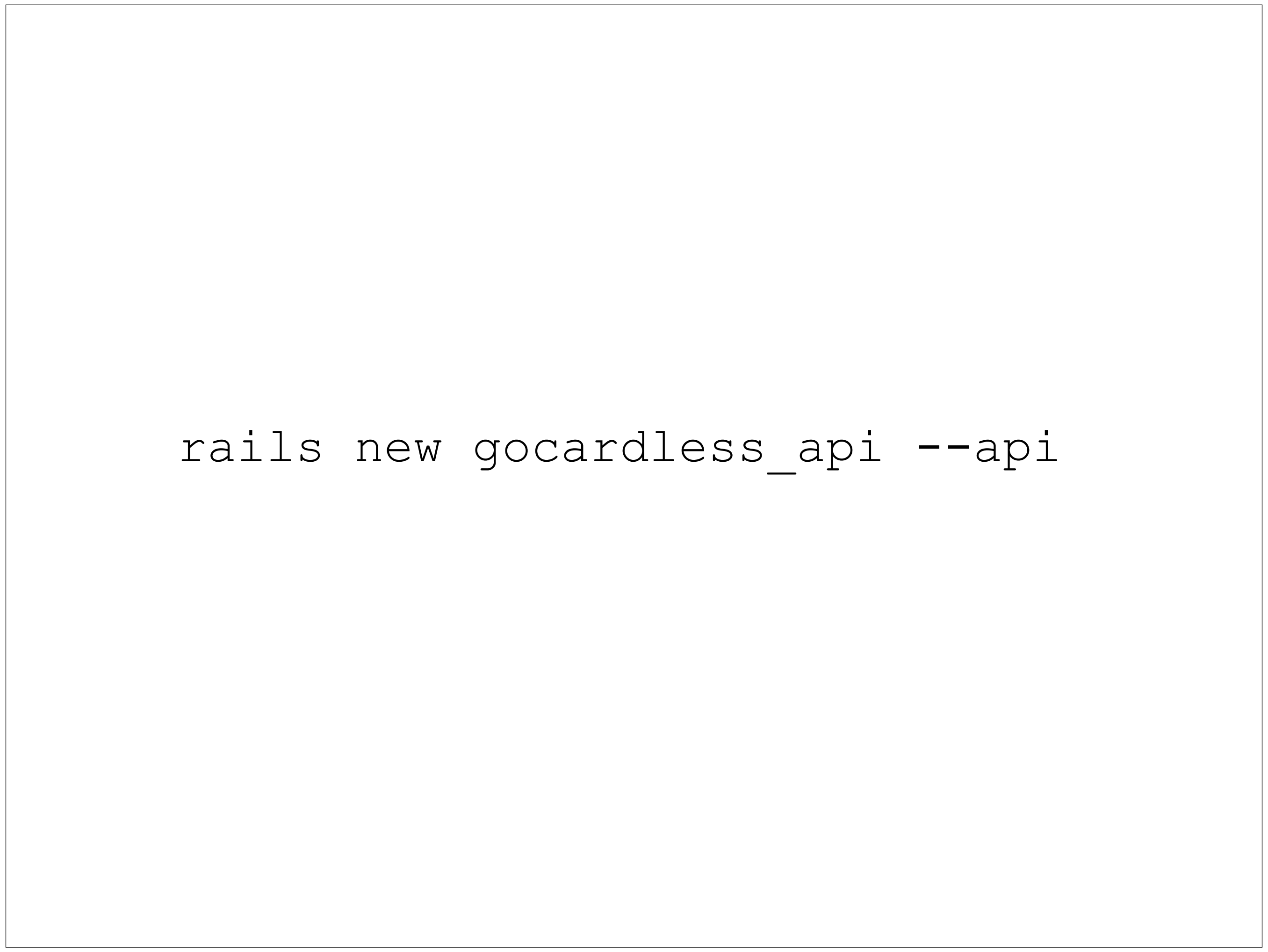
APIs are at the heart of our business.

GoCardless lets you access all of these Direct Debit schemes with a single, simple API.

Today, I'm going to talk about building APIs with Rails 🚂

Building APIs brings some particular software engineering challenges - especially when users are relying on your API to run their business. At GoCardless, we've built up a lot of expertise around designing and implementing great APIs, since they're at the core of what we do.

I'm going to talk about building APIs with Rails.



```
rails new gocardless_api --api
```

It’s not as simple as just turning API mode in our Rails application.

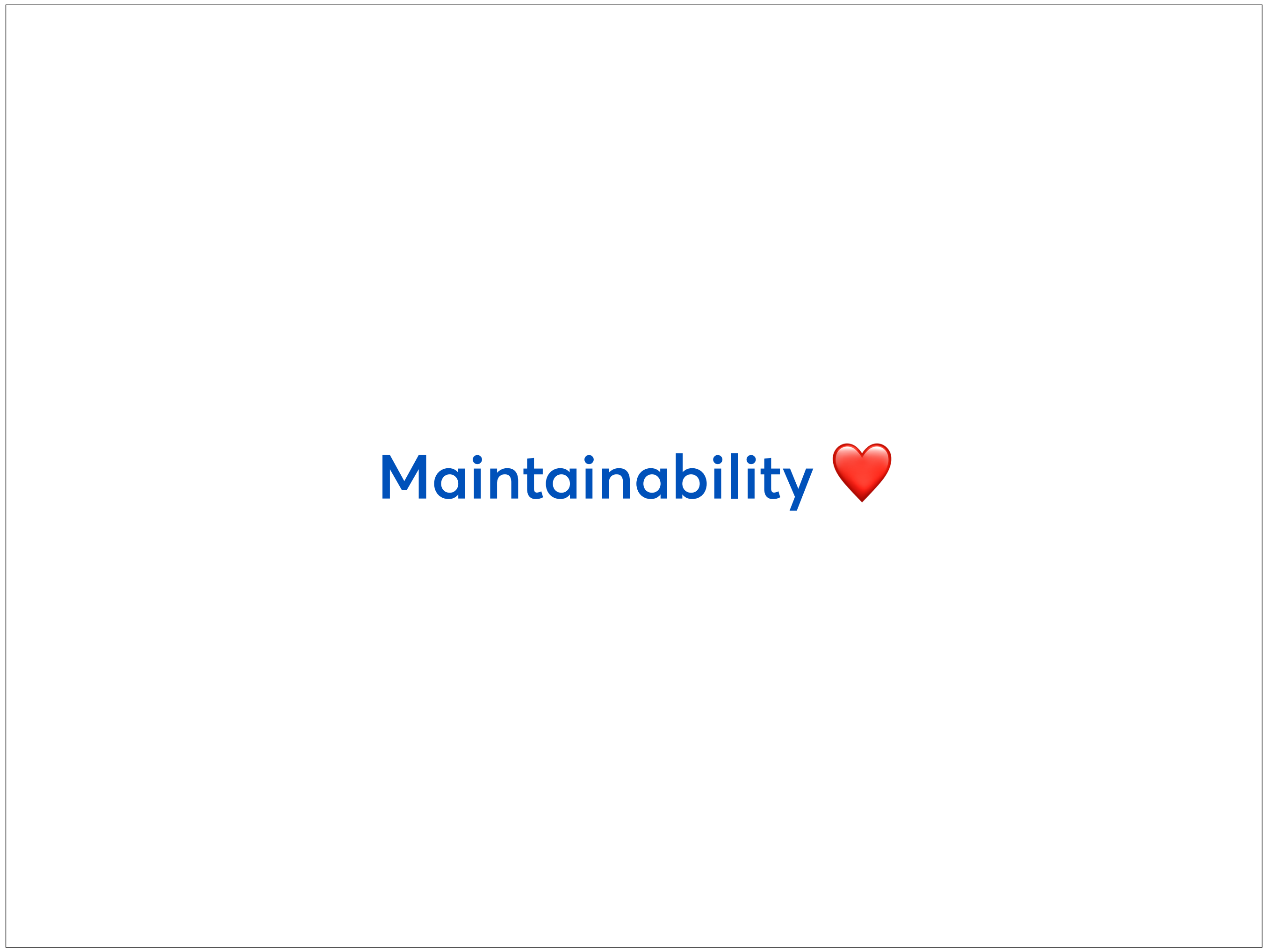
SOFTWARE ENGINEERS HATE HIM

Man from London discovers how to build maintainable APIs with this one WEIRD TRICK

LEARN THE TRUTH NOW

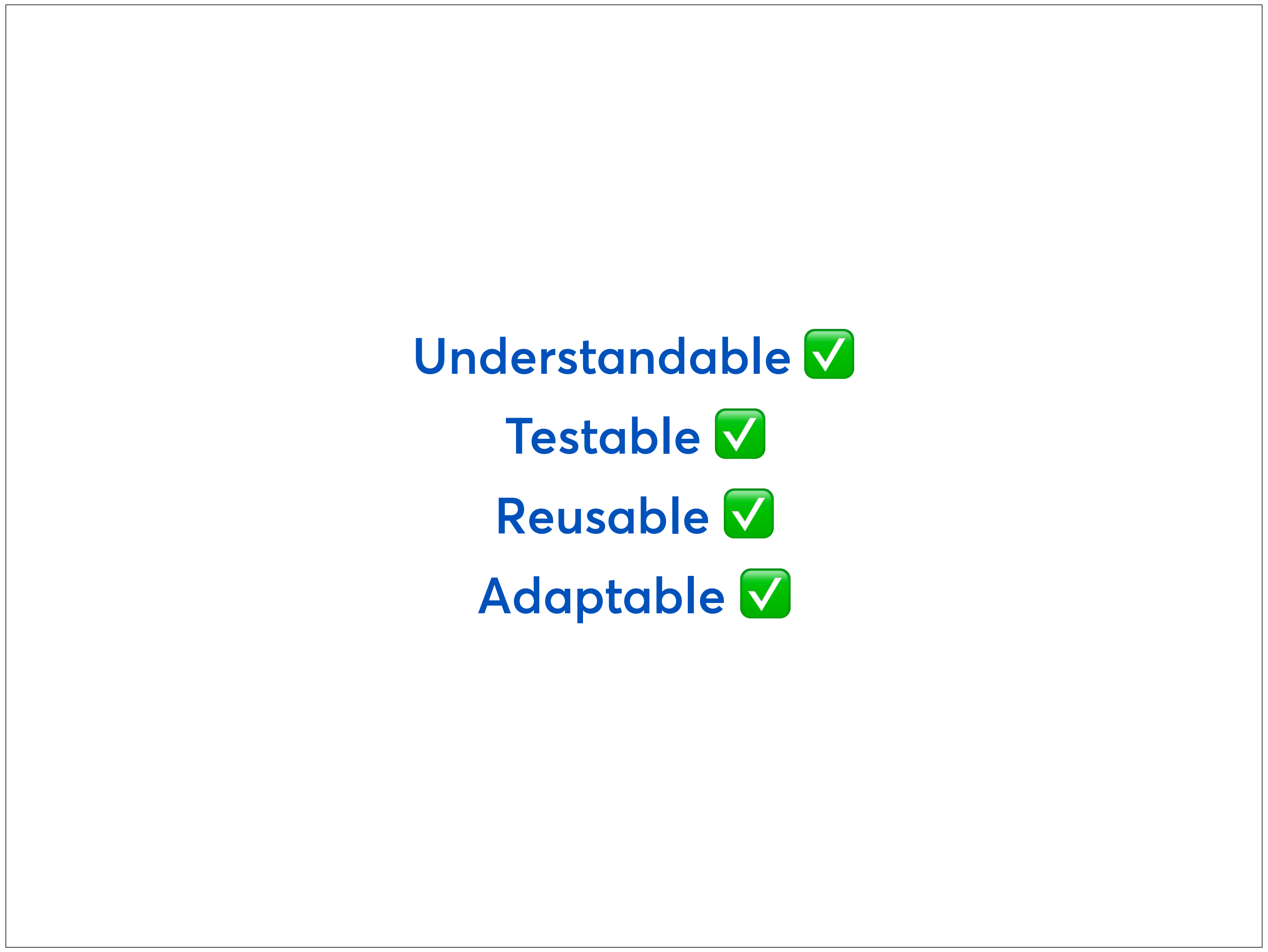
In this talk, I'm not going to show you some kind of incredible secret.

I'm just going to show you how you can apply key software engineering design principles to build better APIs and make your code more maintainable.



Maintainability ❤️

In particular, I'm going to talk about the middleware pattern, which is based on these principles and helps you write better, more maintainable code.


Understandable ✅

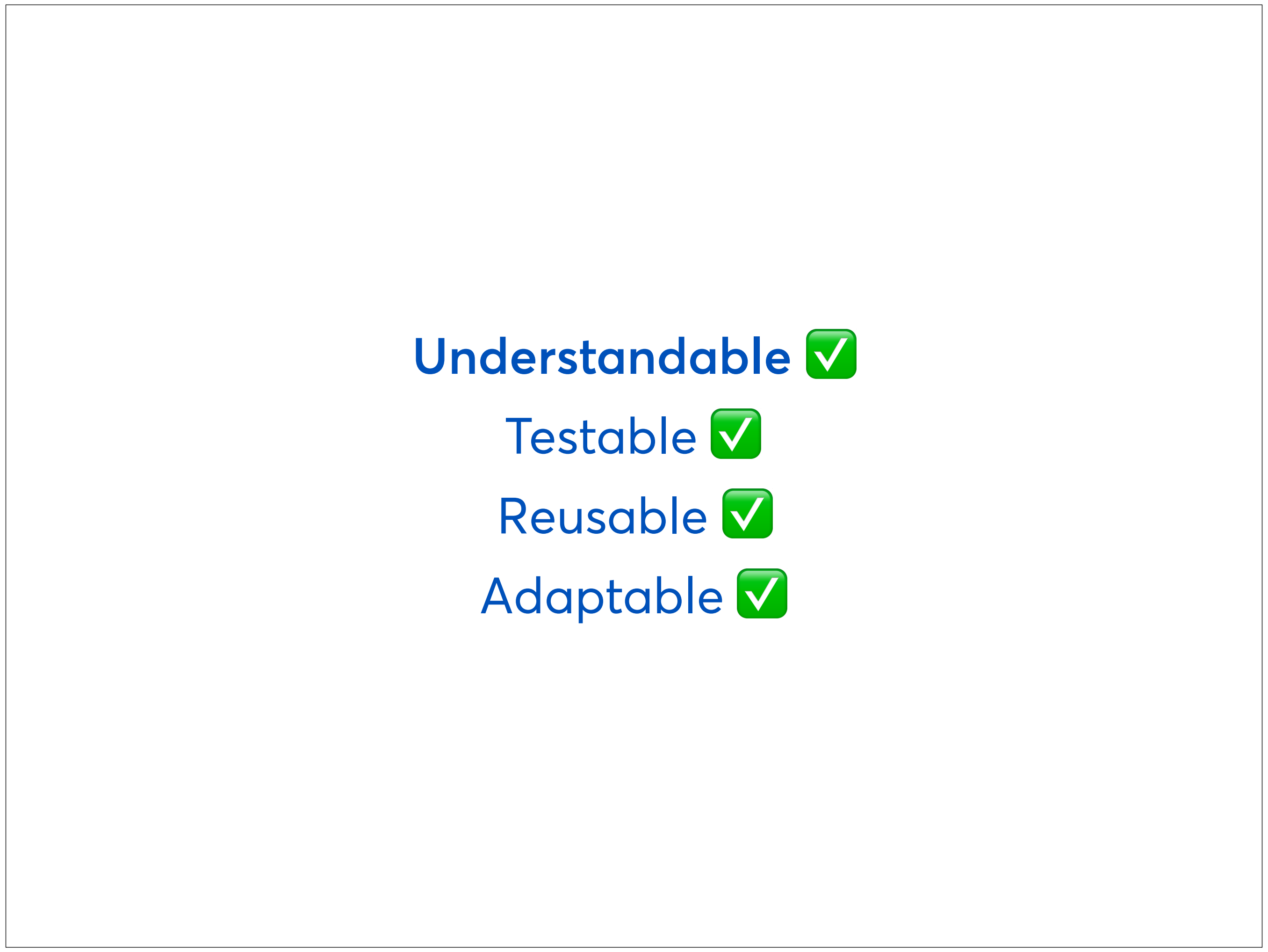
Testable ✅

Reusable ✅

Adaptable ✅

When I say maintainable, what do I mean?

We want our code to be understandable, testable reusable, and adaptable,



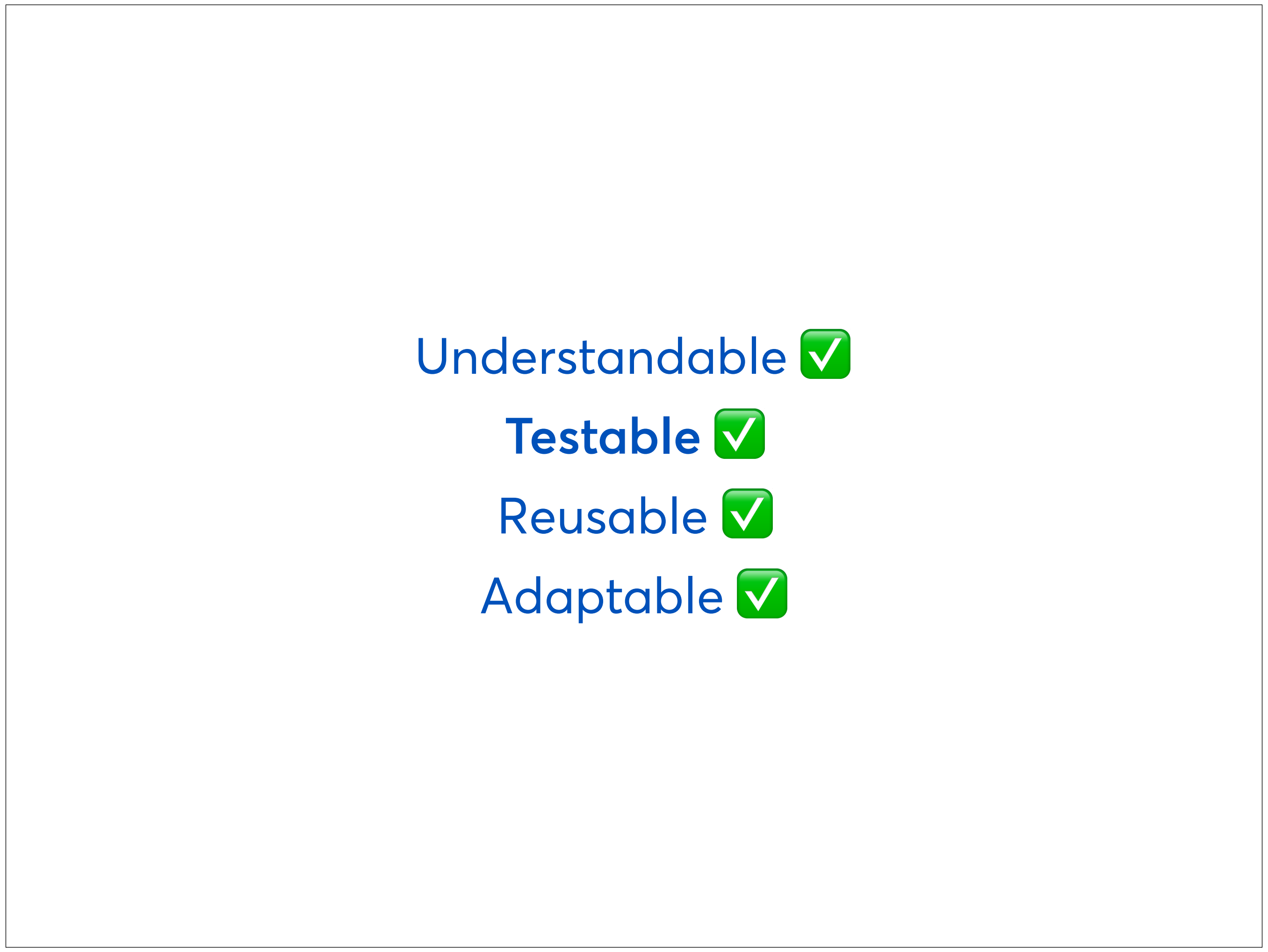
Understandable ✅

Testable ✅

Reusable ✅

Adaptable ✅

We want our code to be understandable. We want code that we - or a member of our team - can come back to in a year’s time and easily understand what it does.



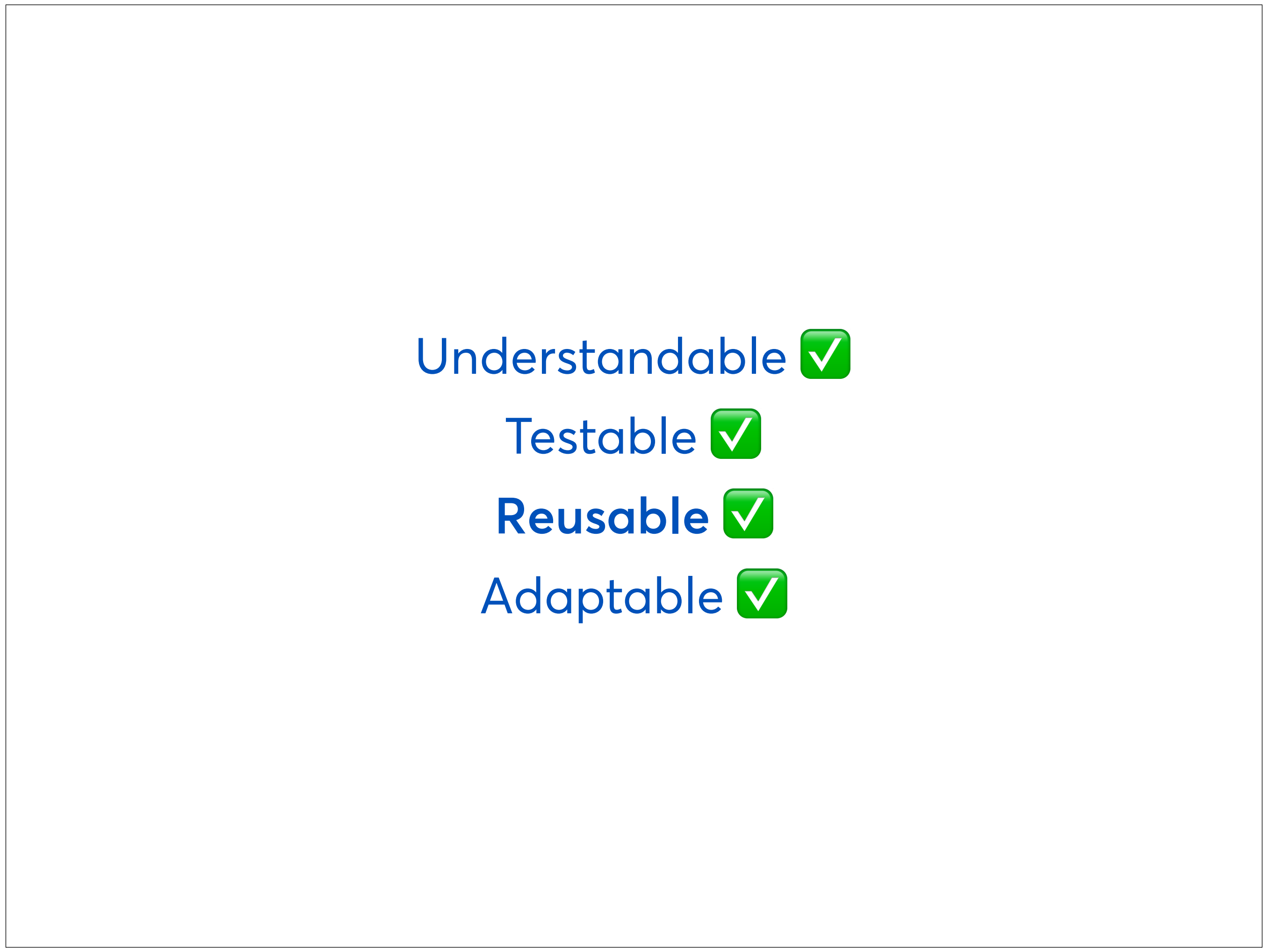
Understandable ✅

Testable ✅

Reusable ✅

Adaptable ✅

We want our code to be testable. It should be written in such a way that makes it easy to test, so we can be confident that it works and confident to make changes to it.



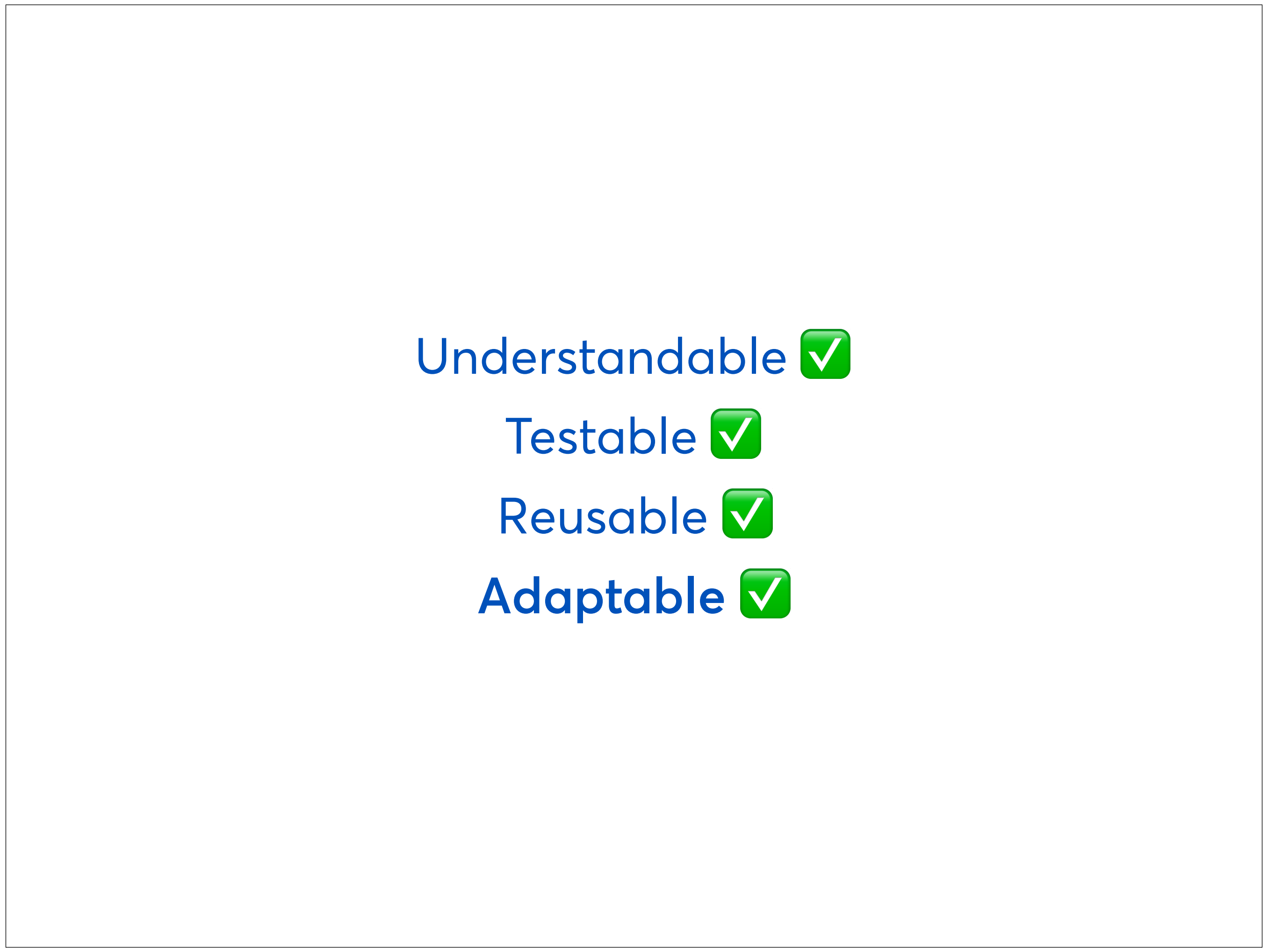
Understandable ✅

Testable ✅

Reusable ✅

Adaptable ✅

We want code to be reusable, so we can provide a consistent interface across our API - for example, having our authentication work the same everywhere. When we want to make changes to our API, we don't want to be forced to change code in hundreds of places.



Understandable ✅

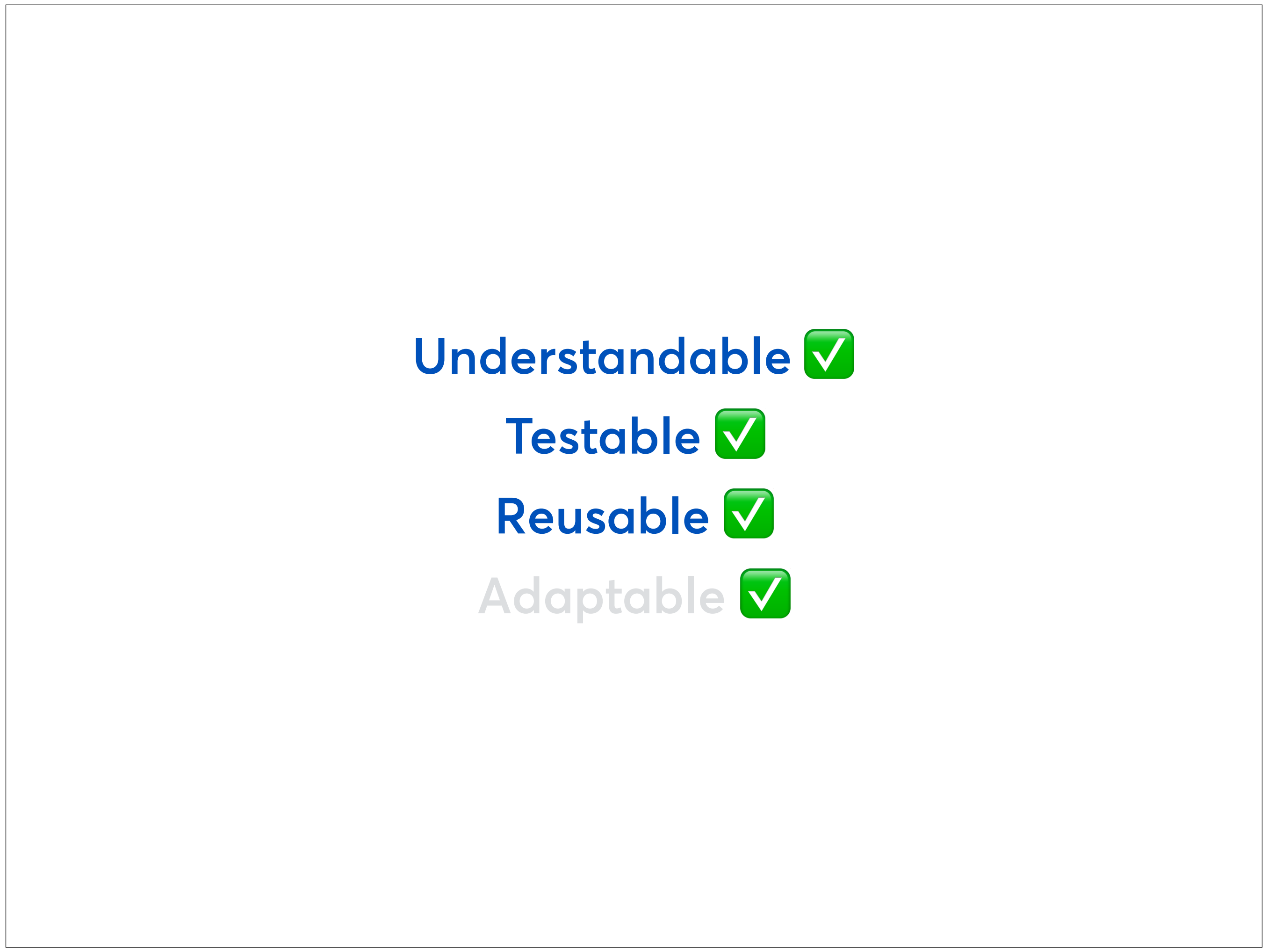
Testable ✅

Reusable ✅

Adaptable ✅

Finally, we want our code to be adaptable.

It should be able to evolve over time, rather than being resistant to change.



Understandable ✅

Testable ✅

Reusable ✅

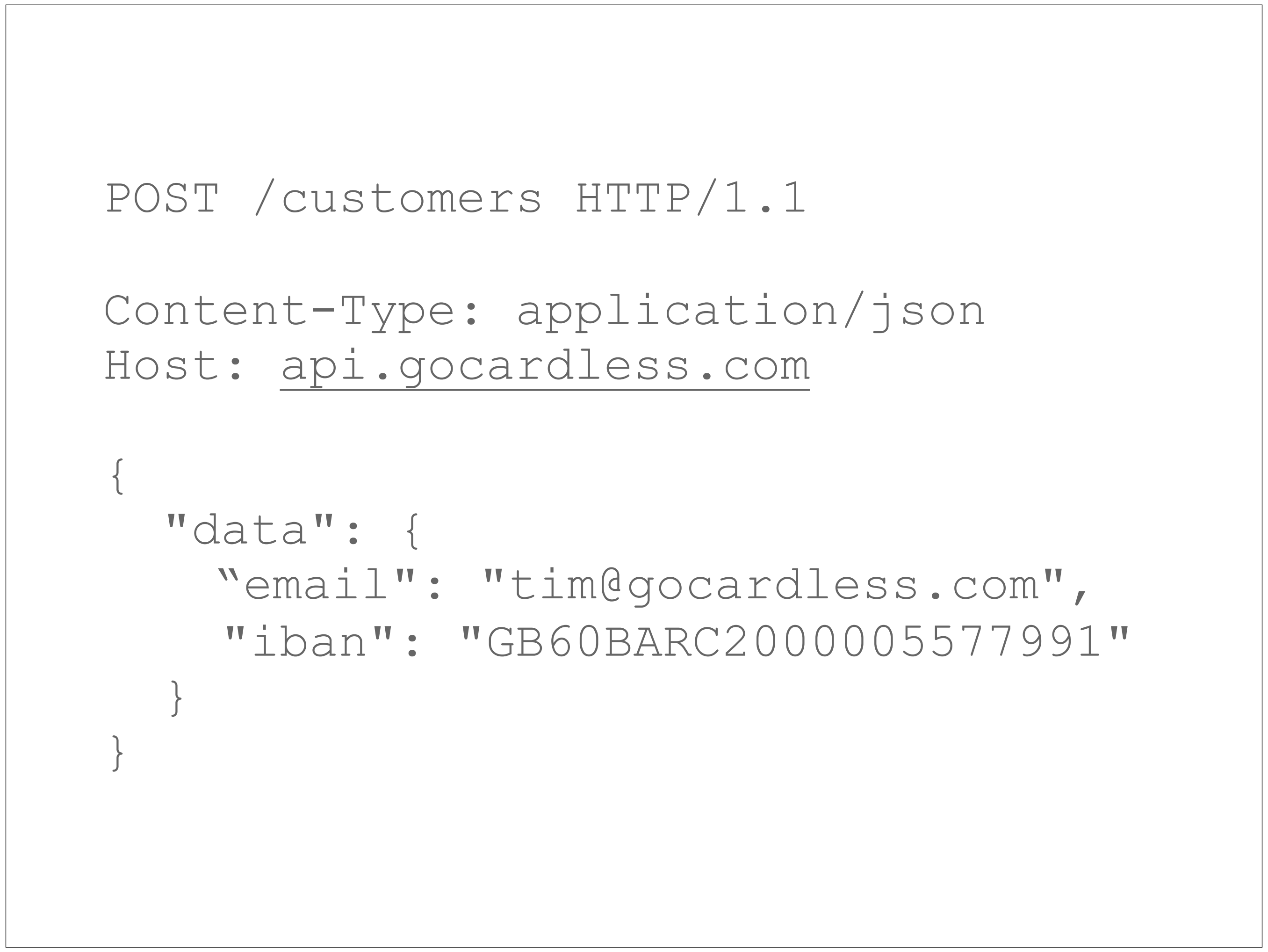
Adaptable ✅

In this talk, we'll focus on how we can make our APIs more understandable, testable and reusable using the middleware pattern.

Let’s start by building a simple API using Rails 🛠️

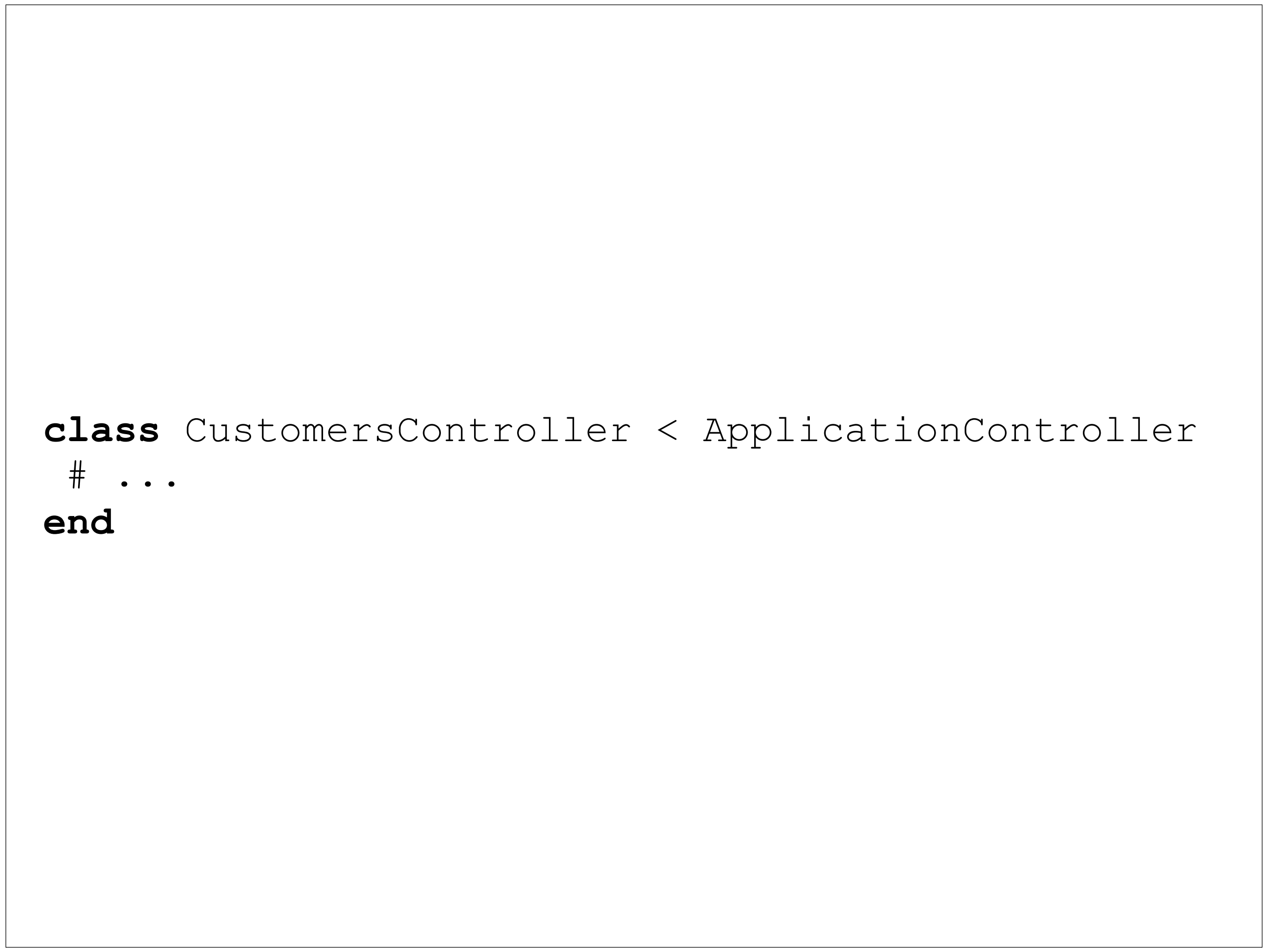
Let’s get started by building a simple API using ActionController. That’ll help us to understand the maintainability challenges we face.

We’ll then think about how we can refactor our code to make it miles more maintainable.



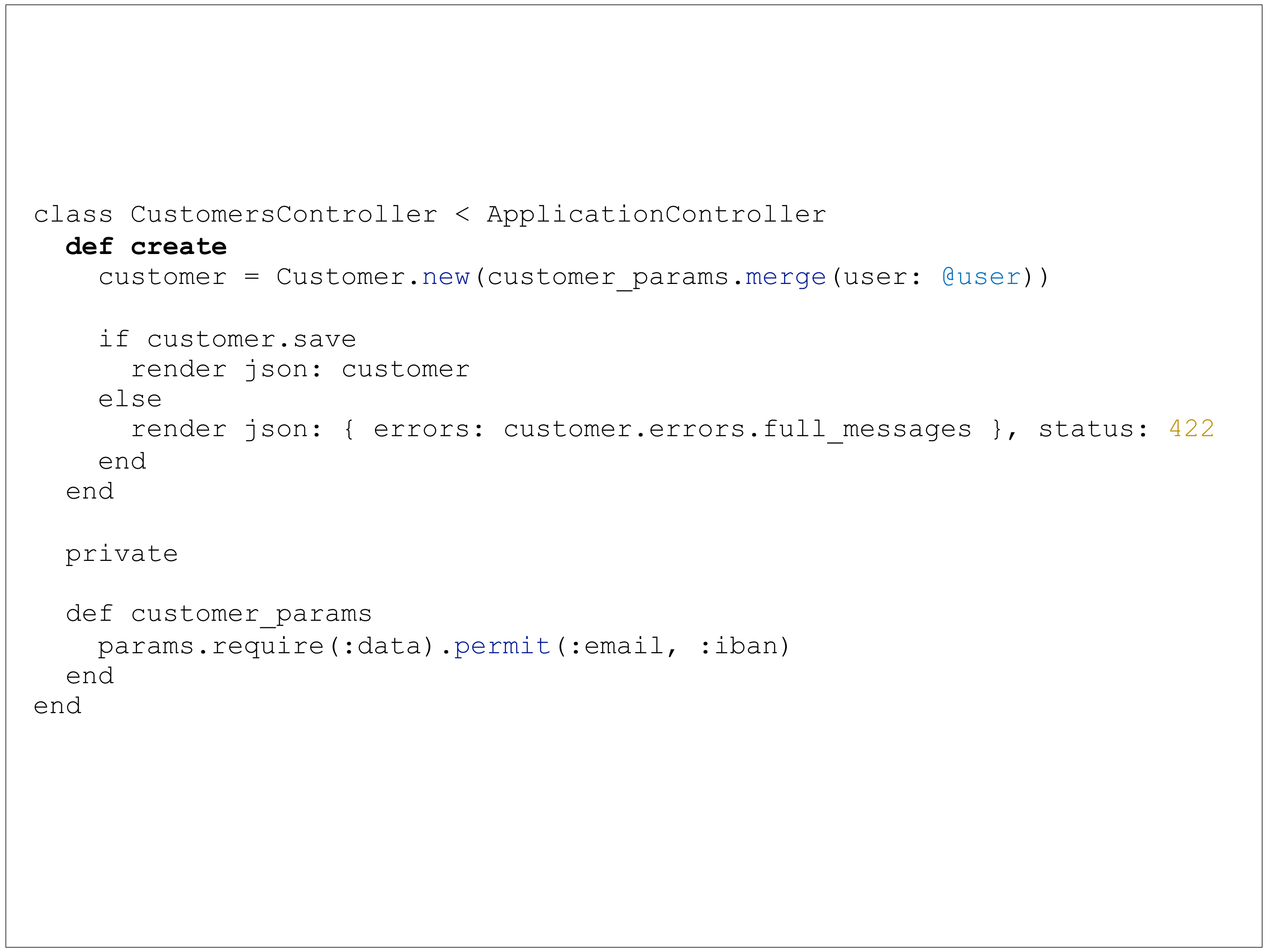
```
POST /customers HTTP/1.1

Content-Type: application/json
Host: api.gocardless.com

{
  "data": {
    “email": "tim@gocardless.com",
     "iban": "GB60BARC20000055779911"
  }
}
```

Let’s start off by imagining a very simple API - it creates a customer based on their email address and bank details.



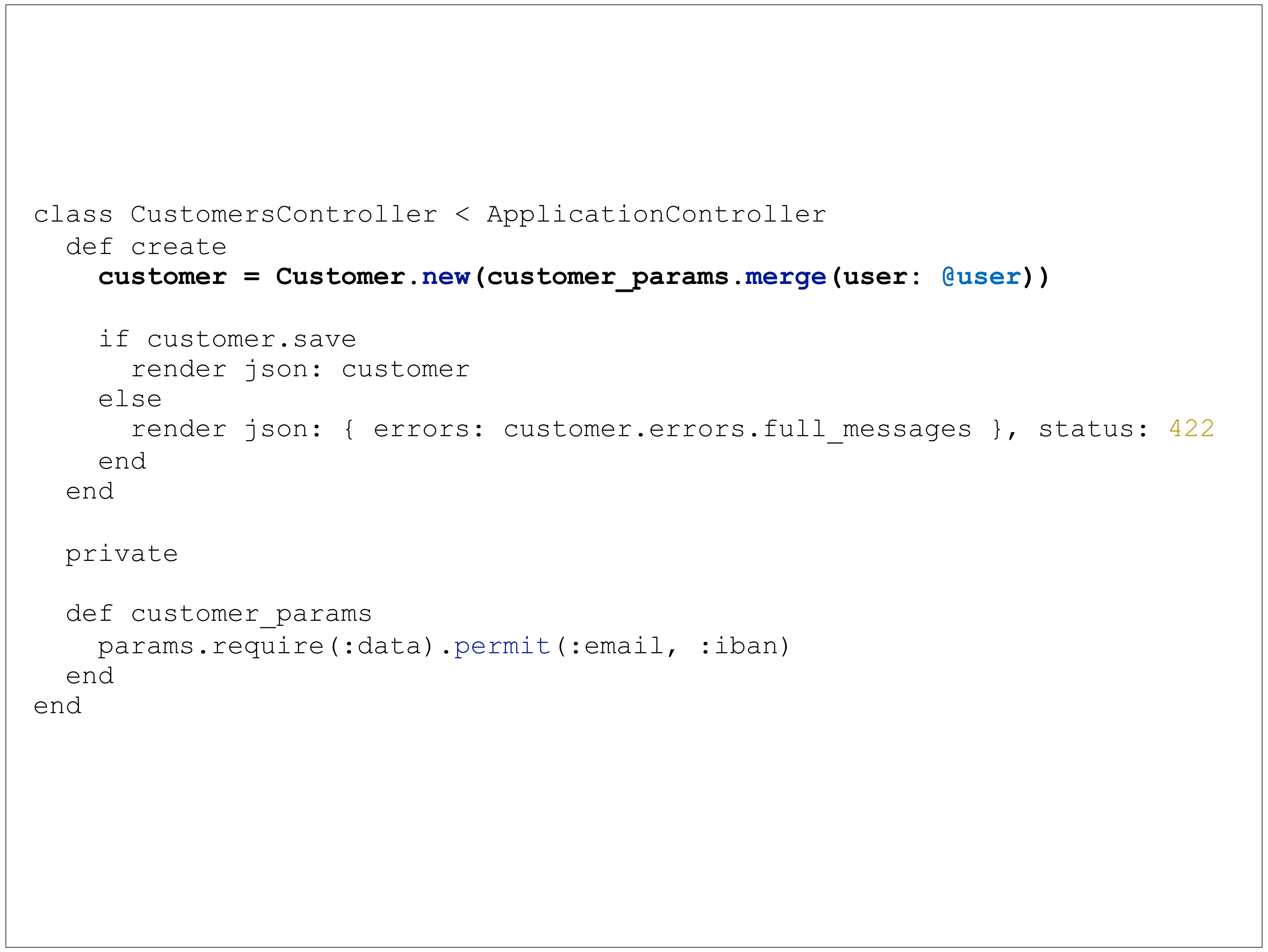
```
class CustomersController < ApplicationController
 # ...
end
```

We’d make a “customers controller”.



```ruby
class CustomersController < ApplicationController
  def create
    customer = Customer.new(customer_params.merge(user: @user))

    if customer.save
      render json: customer
    else
      render json: { errors: customer.errors.full_messages }, status: 422
    end
  end

  private

  def customer_params
    params.require(:data).permit(:email, :iban)
  end
end
```

We’d then add a “create” method, our first controller action.

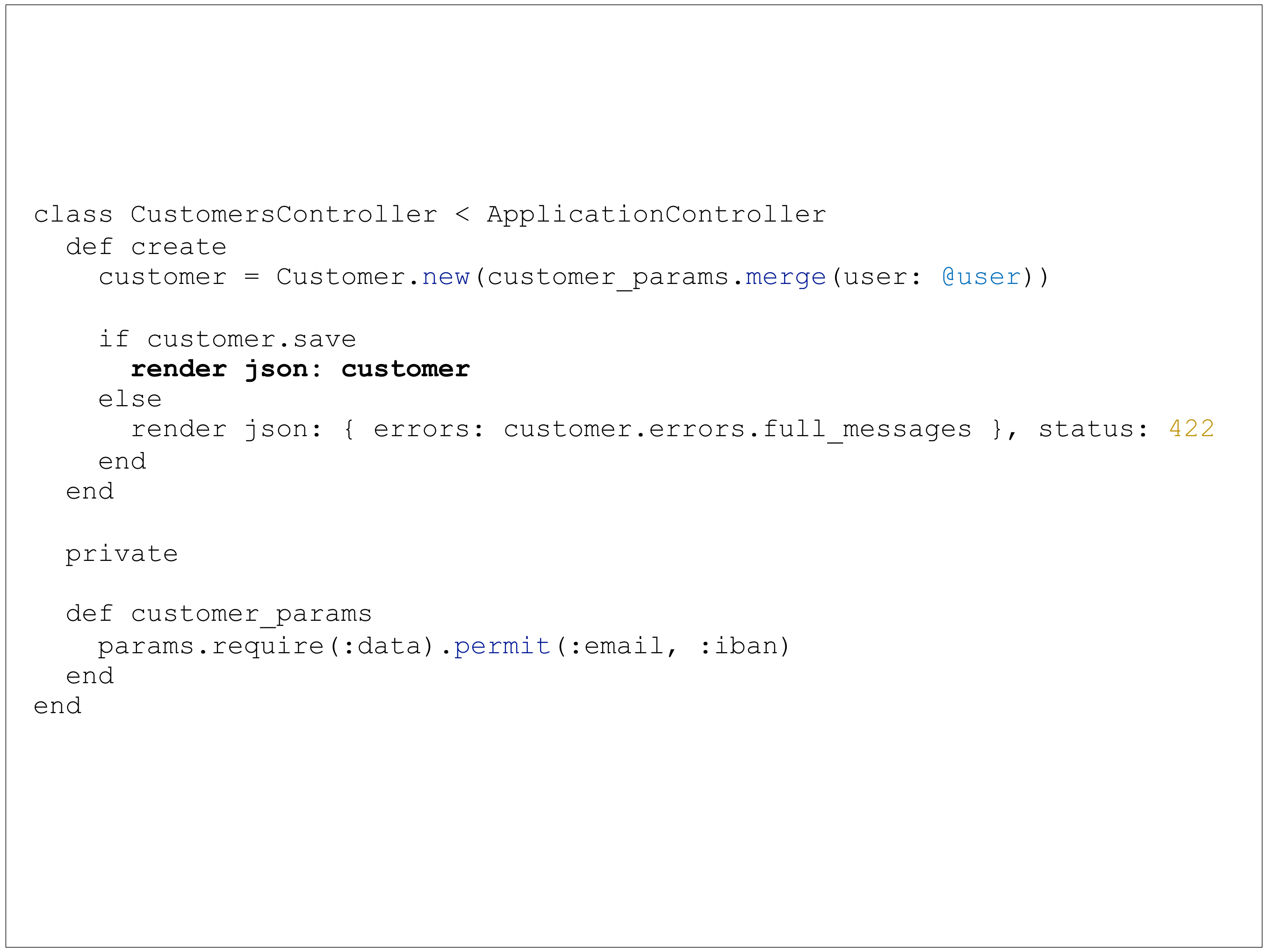


```ruby
class CustomersController < ApplicationController
  def create
    customer = Customer.new(customer_params.merge(user: @user))

    if customer.save
      render json: customer
    else
      render json: { errors: customer.errors.full_messages }, status: 422
    end
  end

  private

  def customer_params
    params.require(:data).permit(:email, :iban)
  end
end
```

It builds the customer from the incoming parameters

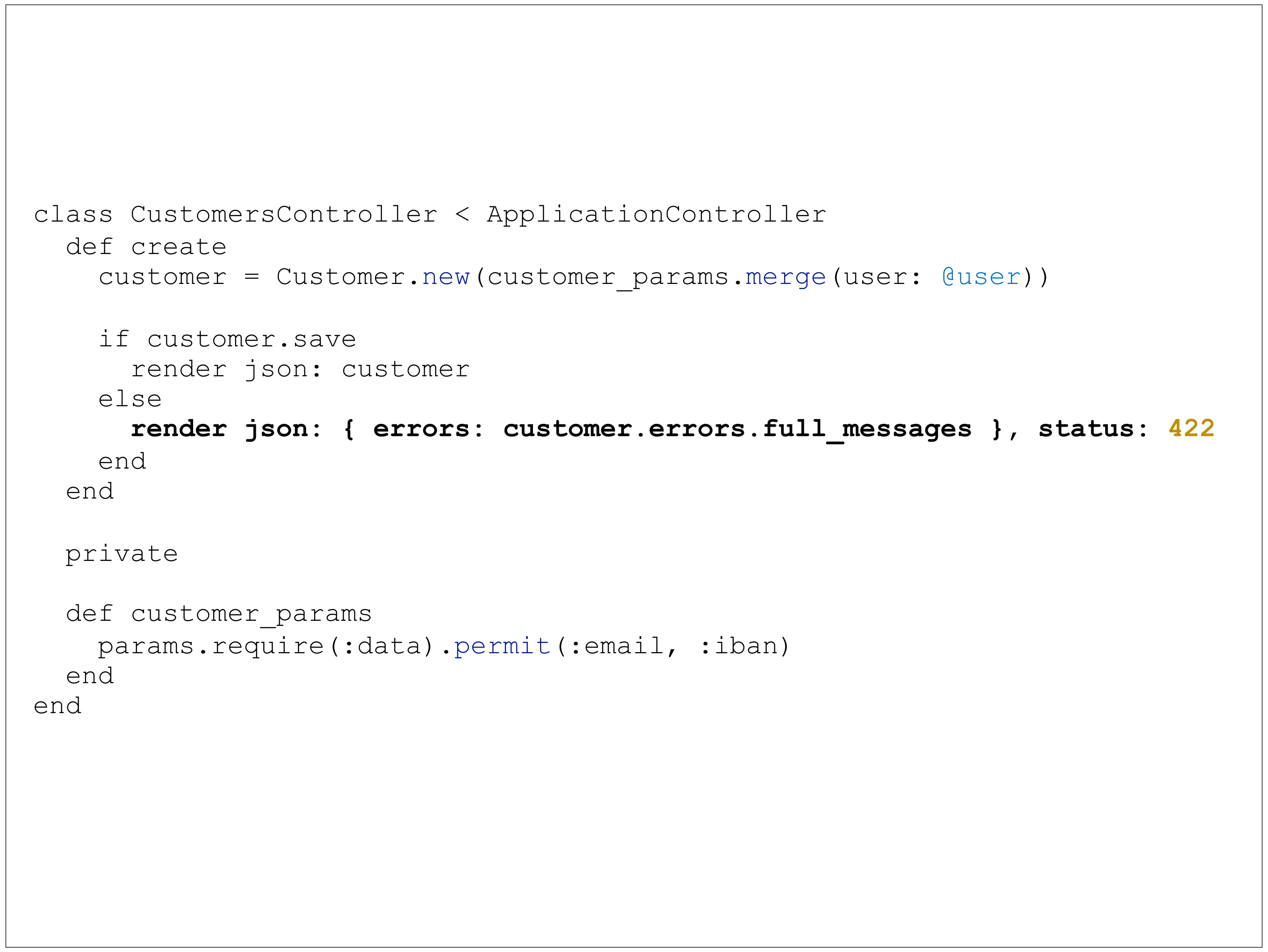
```ruby
class CustomersController < ApplicationController
  def create
    customer = Customer.new(customer_params.merge(user: @user))

    if customer.save
      render json: customer
    else
      render json: { errors: customer.errors.full_messages }, status: 422
    end
  end

  private

  def customer_params
    params.require(:data).permit(:email, :iban)
  end
end
```

And then it tries to save our customer



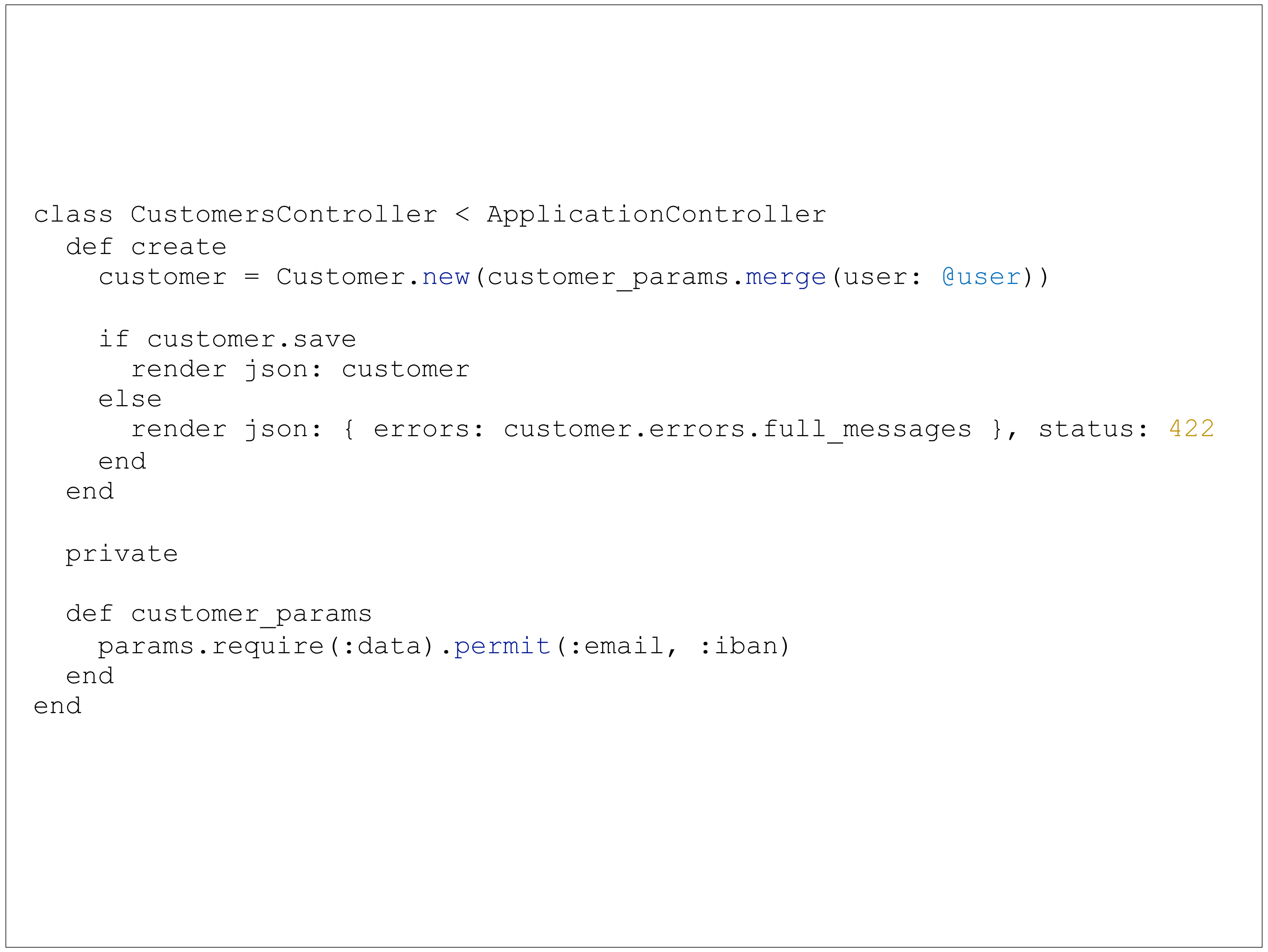
```ruby
class CustomersController < ApplicationController
  def create
    customer = Customer.new(customer_params.merge(user: @user))

    if customer.save
      render json: customer
    else
      render json: { errors: customer.errors.full_messages }, status: 422
    end
  end

  private

  def customer_params
    params.require(:data).permit(:email, :iban)
  end
end
```

If the customer saves successfully, then we render the created customer back to the user as JSON.



```ruby
class CustomersController < ApplicationController
  def create
    customer = Customer.new(customer_params.merge(user: @user))

    if customer.save
      render json: customer
    else
      render json: { errors: customer.errors.full_messages }, status: 422
    end
  end

  private

  def customer_params
    params.require(:data).permit(:email, :iban)
  end
end
```

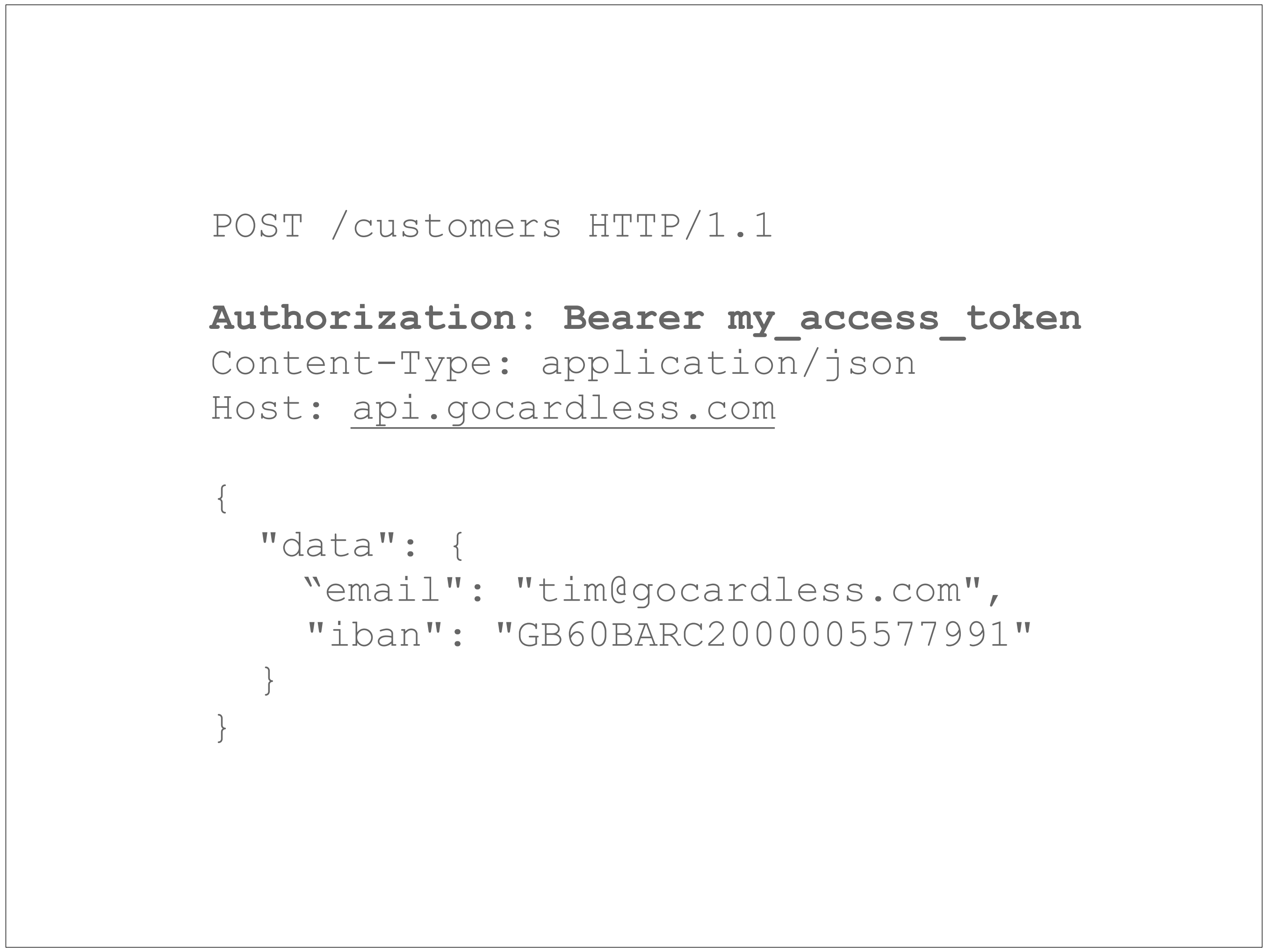
And if there are validation errors, so we can’t create the record, we return the errors back to the user.



```ruby
class CustomersController < ApplicationController
  def create
    customer = Customer.new(customer_params.merge(user: @user))

    if customer.save
      render json: customer
    else
      render json: { errors: customer.errors.full_messages }, status: 422
    end
  end

  private

  def customer_params
    params.require(:data).permit(:email, :iban)
  end
end
```

This is all very simple so far, right? Obviously, we’re missing quite a lot of important things here.

For example, our API would need some kind of authentication.



```
POST /customers HTTP/1.1

Authorization: Bearer my_access_token
Content-Type: application/json
Host: api.gocardless.com

{
  "data": {
    “email": "tim@gocardless.com",
    "iban": "GB60BARC20000055779911"
  }
}
```

The standard way to do that is to send an access token in the Authorization header.

```ruby
class CustomersController < ApplicationController
  def create
    customer = Customer.new(customer_params.merge(user: @user))

    if customer.save
      render json: customer
    else
      render json: { errors: customer.errors.full_messages }, status: 422
    end
  end

  private

  def customer_params
    params.require(:data).permit(:email, :iban)
  end
end
```

We *could* write the code to handle this straight into the create method, but that immediately makes our code less maintainable.

Understandable ❌

Reusable ❌

Testable ❌

Our code is immediately harder to understand - straight away, we’ll have a long method doing lots of jobs, with no division of labour.

Understandable ❌

Reusable ❌

Testable ❌

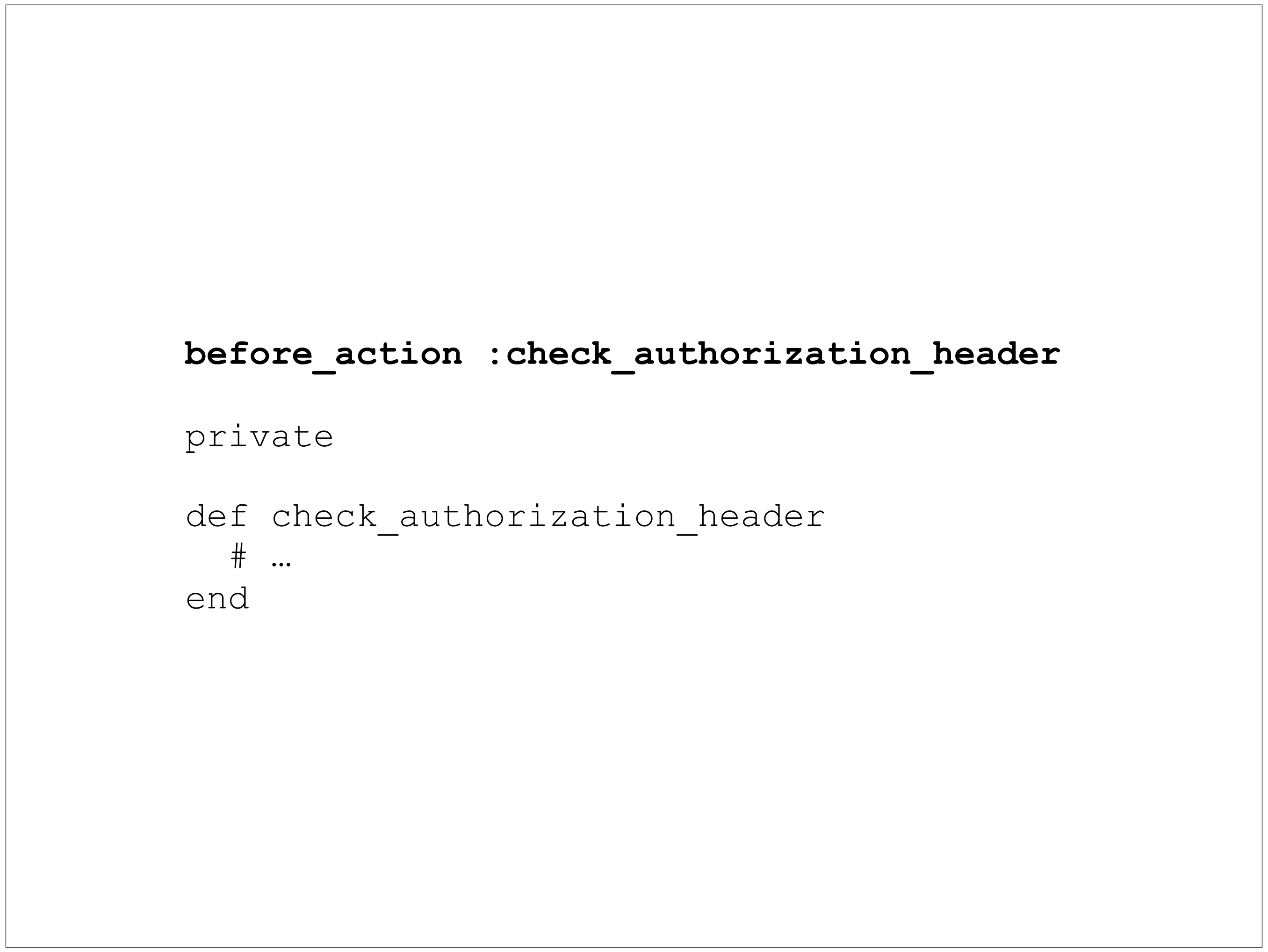
Our code will be hard to reuse, since it isn’t abstracted out, but rather lives in the “create” method.

Understandable ❌

Reusable ❌

Testable ❌

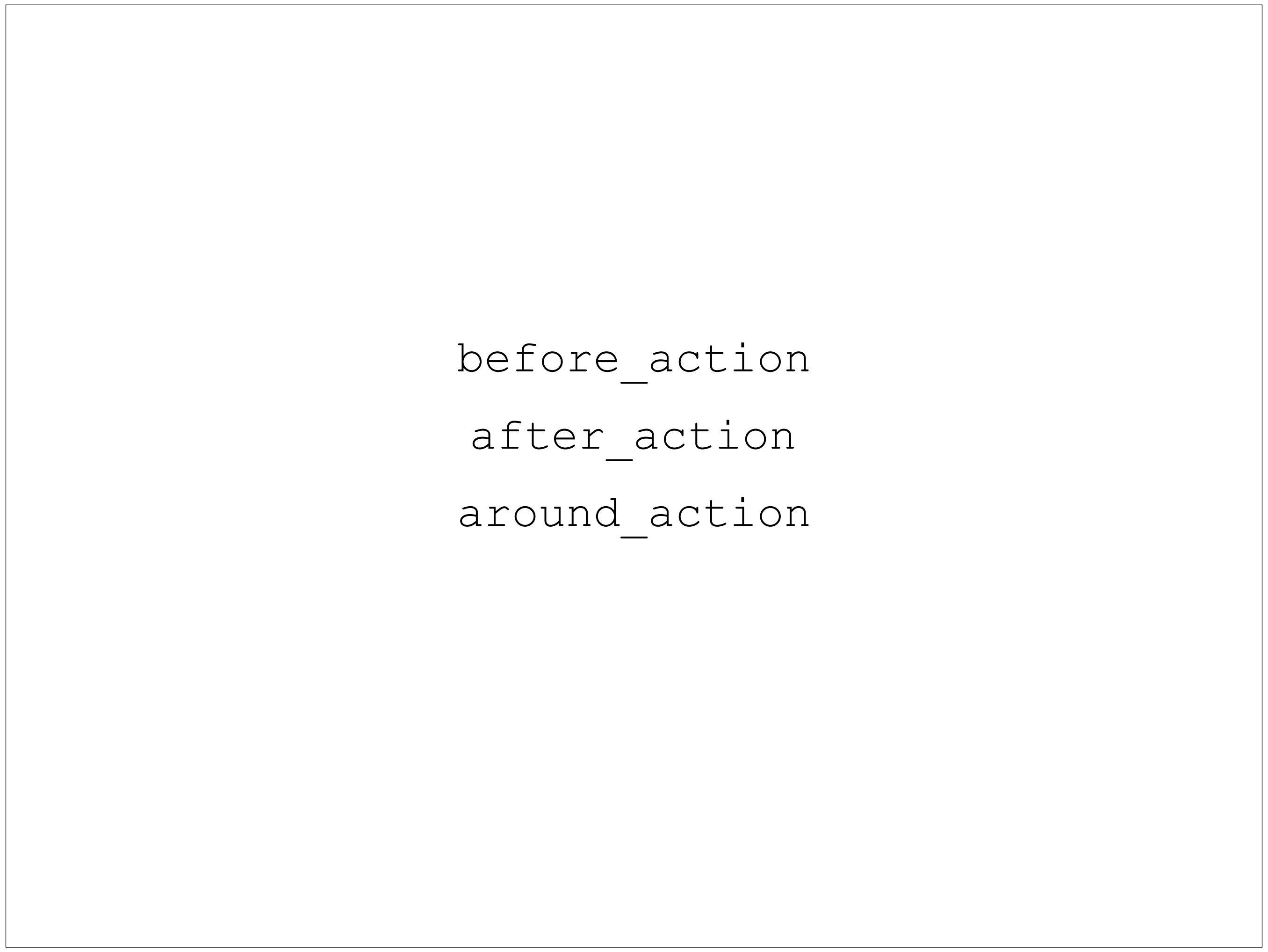
And our code will be hard to test - we can’t exercise it on its own, but rather only in the context of the controller



```
before_action :check_authorization_header

private

def check_authorization_header
  # …
end
```

The most common solution to this problem is to split out the authentication code into separate methods, using Rails’s filter functionality.

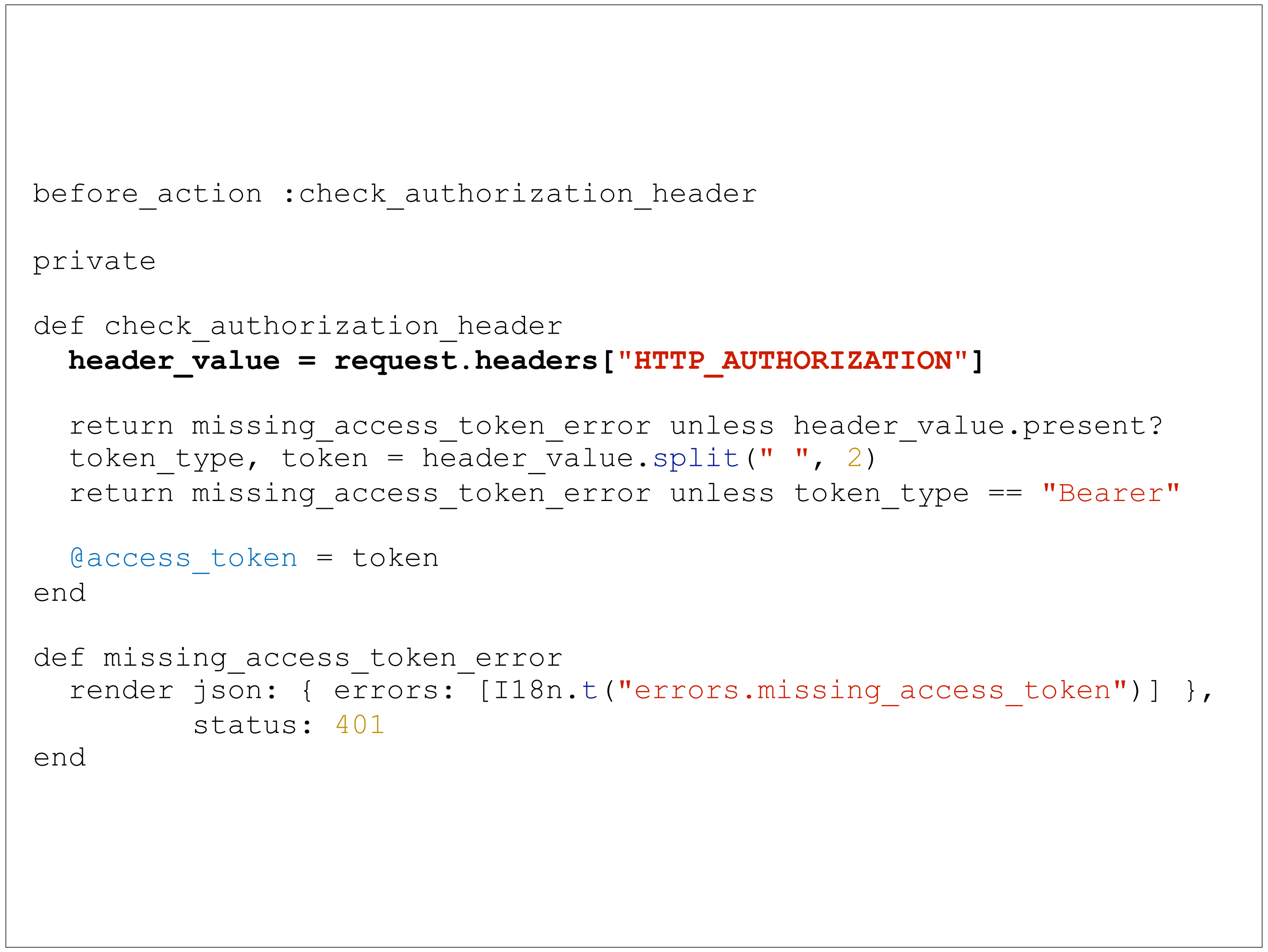
Here, we define a “check authorization header” method, and tell Rails to run it before our controller action with `before_action`.



```
before_action
after_action
around_action
```

Filters let you run code before, after or around your controller action.

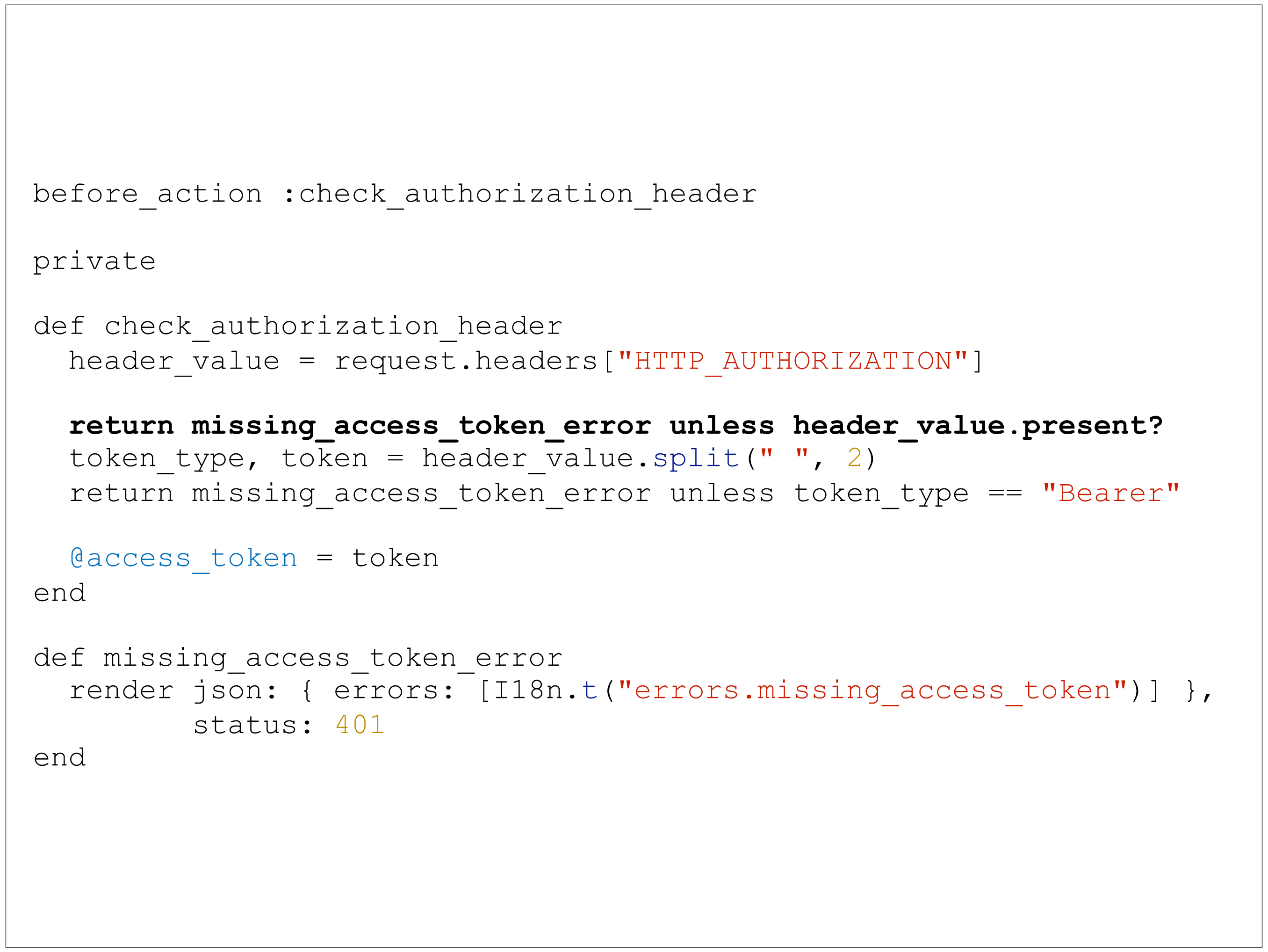
It’s worth noting that the naming of this feature is a bit confused. On Rails models, they’re called callbacks, but the Rails documentation refers to this feature in ActionController as “filters”, so we’ll stick with that.



```ruby
before_action :check_authorization_header

private

def check_authorization_header
  header_value = request.headers["HTTP_AUTHORIZATION"]

  return missing_access_token_error unless header_value.present?
  token_type, token = header_value.split(" ", 2)
  return missing_access_token_error unless token_type == "Bearer"

  @access_token = token
end

def missing_access_token_error
  render json: { errors: [I18n.t("errors.missing_access_token")] },
         status: 401
end
```

We can define a “before” filter using `before_action`, and that method will be run before our create method.

In our `#check_authorization_header` method, we pull out the value of the Authorization header.

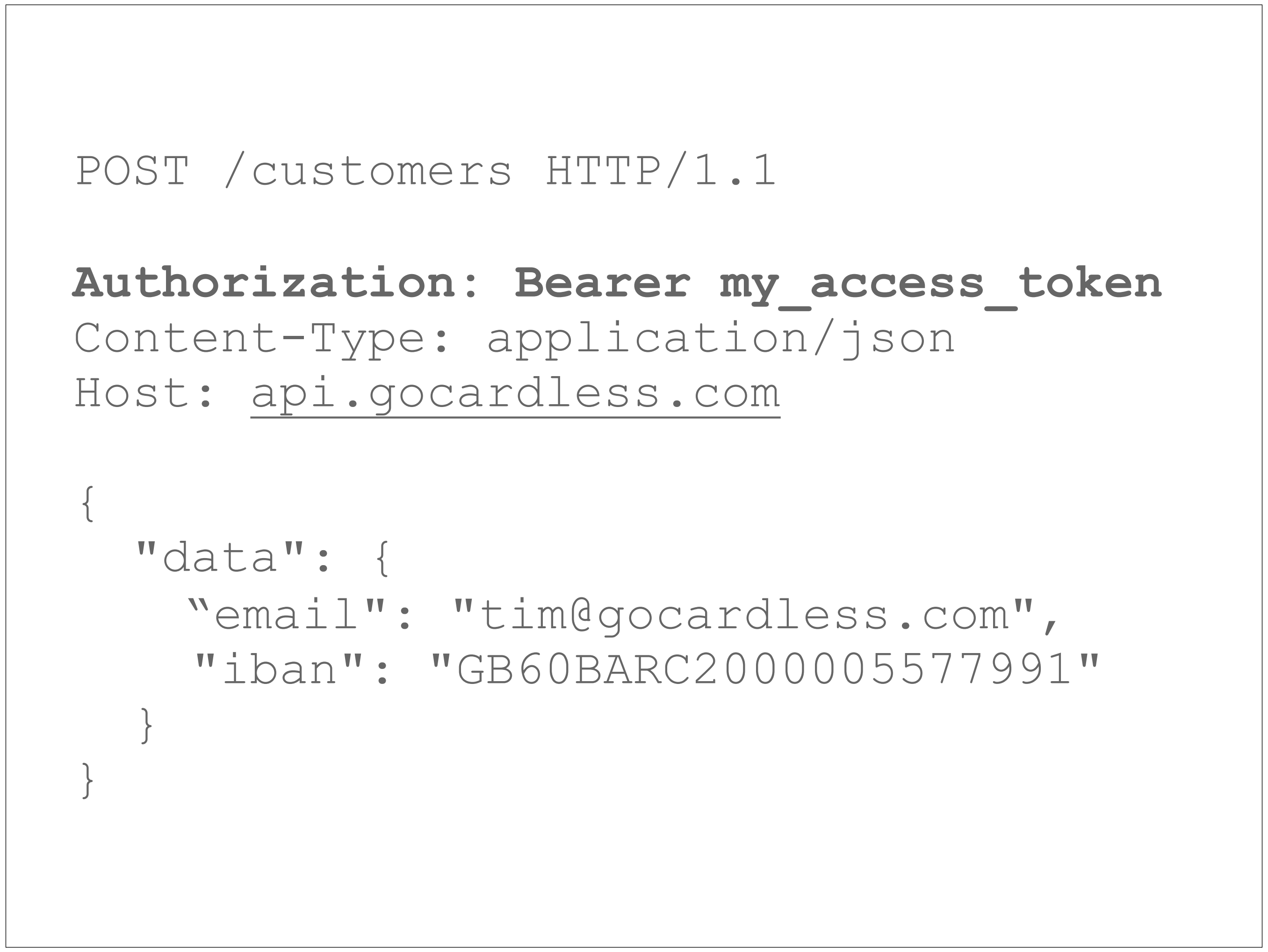


```ruby
before_action :check_authorization_header

private

def check_authorization_header
  header_value = request.headers["HTTP_AUTHORIZATION"]

  return missing_access_token_error unless header_value.present?
  token_type, token = header_value.split(" ", 2)
  return missing_access_token_error unless token_type == "Bearer"

  @access_token = token
end

def missing_access_token_error
  render json: { errors: [I18n.t("errors.missing_access_token")] },
         status: 401
end
```

If the header is missing, then we return an error.

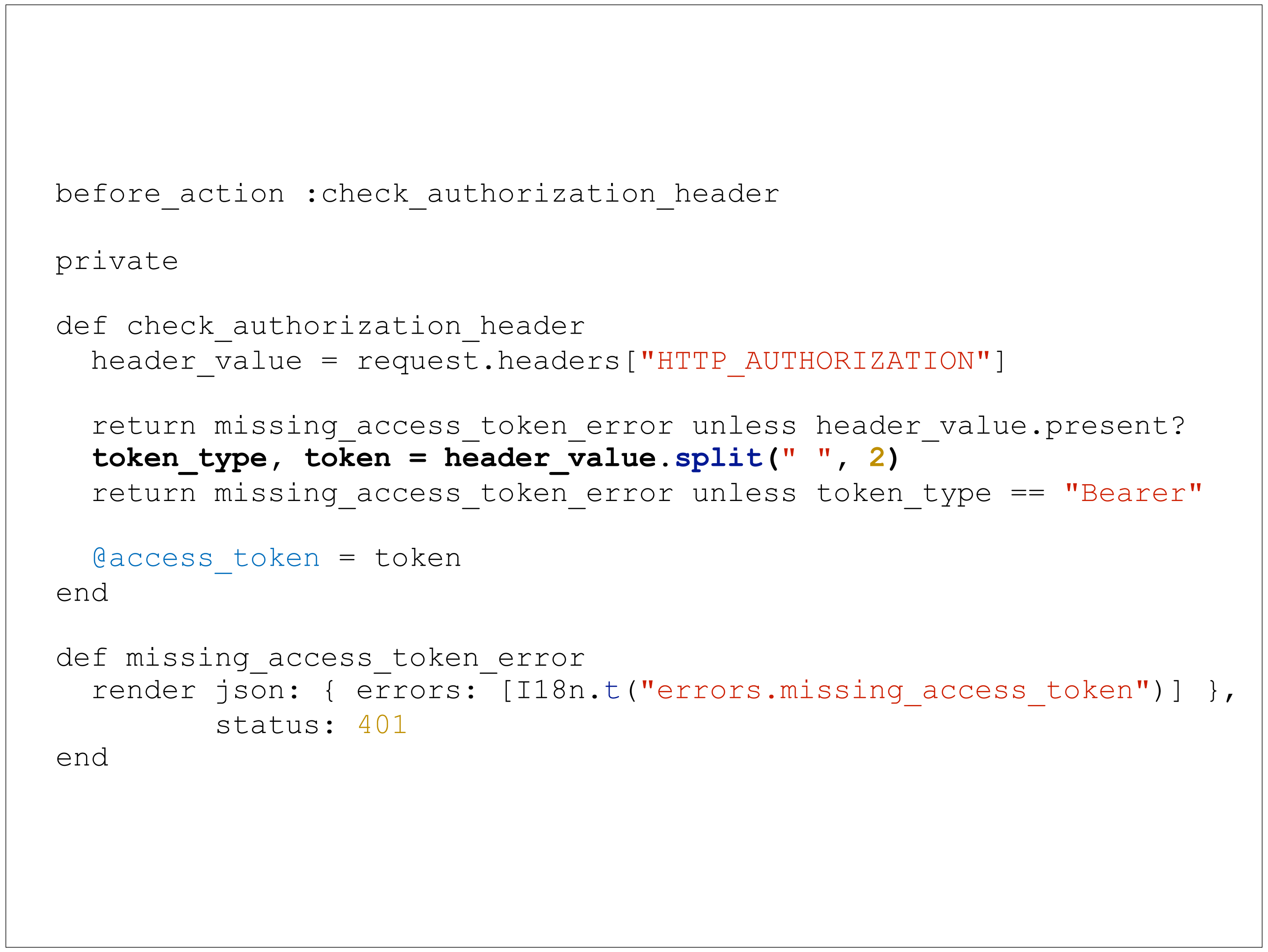
If our filter raises an exception or returns an HTTP response as we d here, execution will stop. Otherwise, Rails will move on to the next filter, and eventually on to the action itself, the “create” method.

Once we’ve checked that the header has actually been provided, we need to look at its value.



```
POST /customers HTTP/1.1

Authorization: Bearer my_access_token
Content-Type: application/json
Host: api.gocardless.com

{
  "data": {
    “email": "tim@gocardless.com",
     "iban": "GB60BARC20000055779911"
  }
}
```

Which, if you remember, looks like this: “Bearer”, followed by an access token



```ruby
before_action :check_authorization_header

private

def check_authorization_header
  header_value = request.headers["HTTP_AUTHORIZATION"]

  return missing_access_token_error unless header_value.present?
  token_type, token = header_value.split(" ", 2)
  return missing_access_token_error unless token_type == "Bearer"

  @access_token = token
end

def missing_access_token_error
  render json: { errors: [I18n.t("errors.missing_access_token")] },
         status: 401
end
```

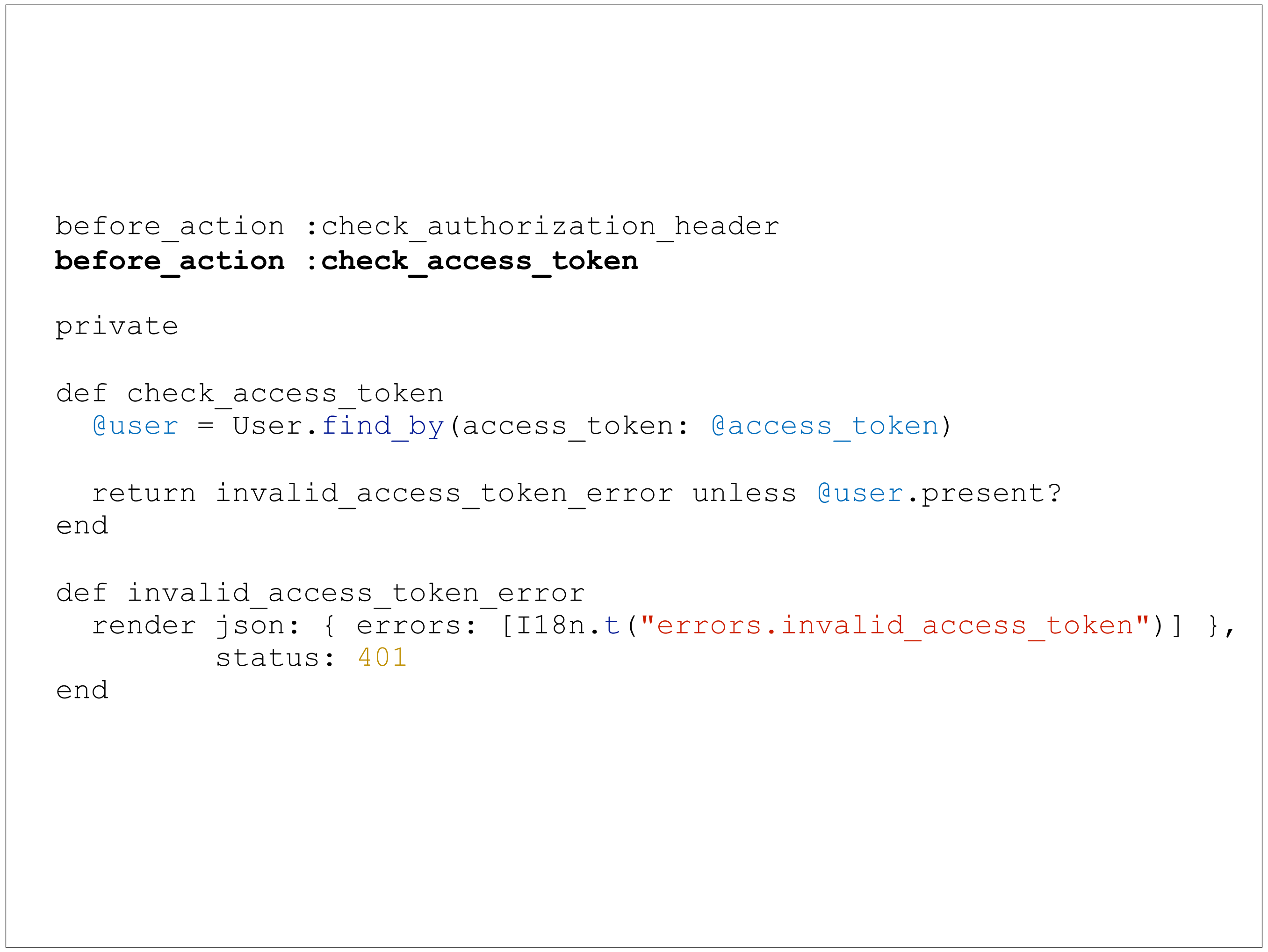
We split the header into its two parts

```ruby
before_action :check_authorization_header

private

def check_authorization_header
  header_value = request.headers["HTTP_AUTHORIZATION"]

  return missing_access_token_error unless header_value.present?
  token_type, token = header_value.split(" ", 2)
  return missing_access_token_error unless token_type == "Bearer"

  @access_token = token
end

def missing_access_token_error
  render json: { errors: [I18n.t("errors.missing_access_token")] },
         status: 401
end
```

And then we check that it starts with “Bearer”, returning an error if not

```
before_action :check_authorization_header

private

def check_authorization_header
  header_value = request.headers["HTTP_AUTHORIZATION"]

  return missing_access_token_error unless header_value.present?
  token_type, token = header_value.split(" ", 2)
  return missing_access_token_error unless token_type == "Bearer"

  @access_token = token
end

def missing_access_token_error
  render json: { errors: [I18n.t("errors.missing_access_token")] },
         status: 401
end
```

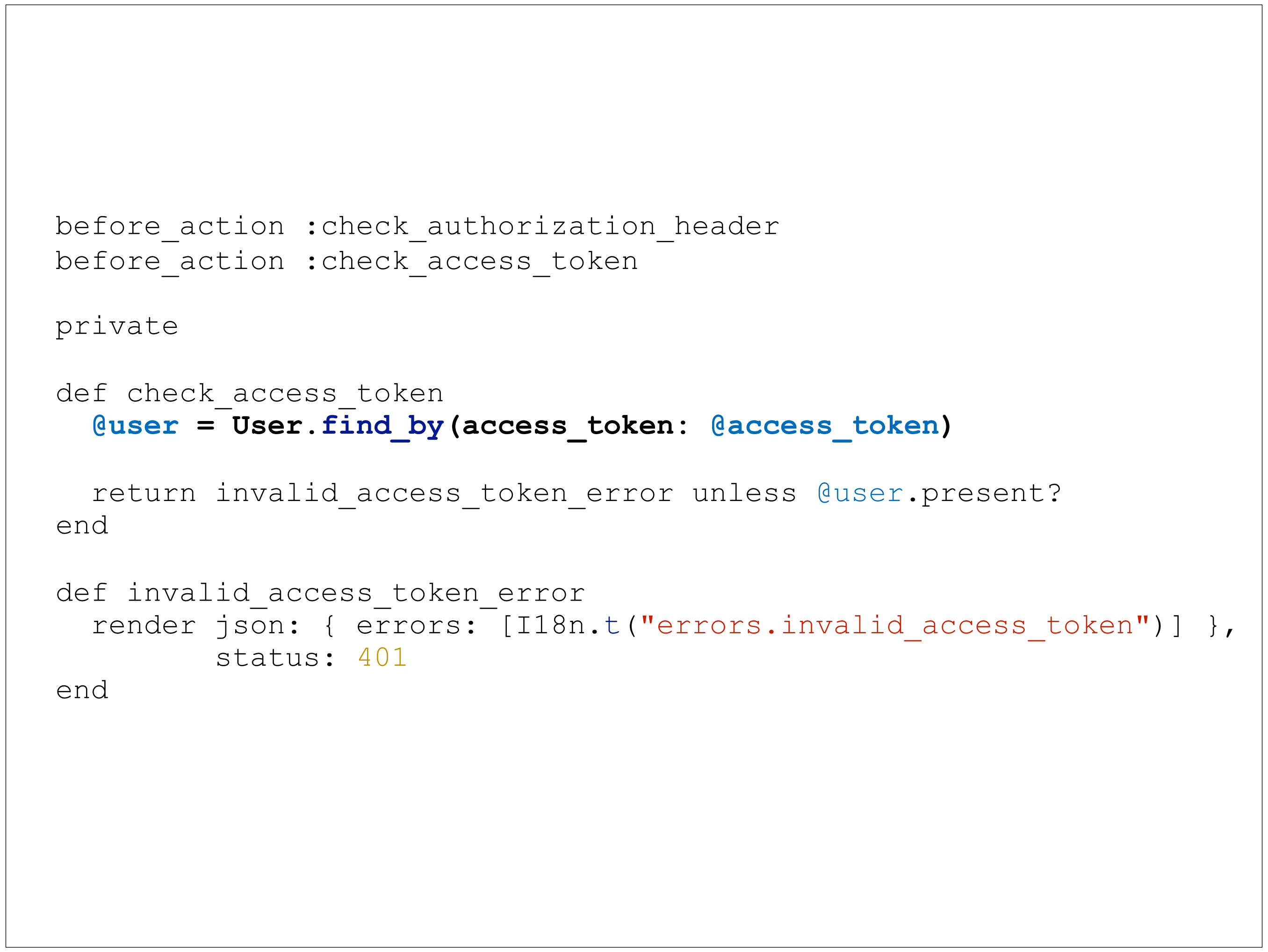
If everything looks good, we save the access token to an instance variable, @access_token.



```ruby
before_action :check_authorization_header
before_action :check_access_token

private

def check_access_token
  @user = User.find_by(access_token: @access_token)

  return invalid_access_token_error unless @user.present?
end

def invalid_access_token_error
  render json: { errors: [I18n.t("errors.invalid_access_token")] },
         status: 401
end
```

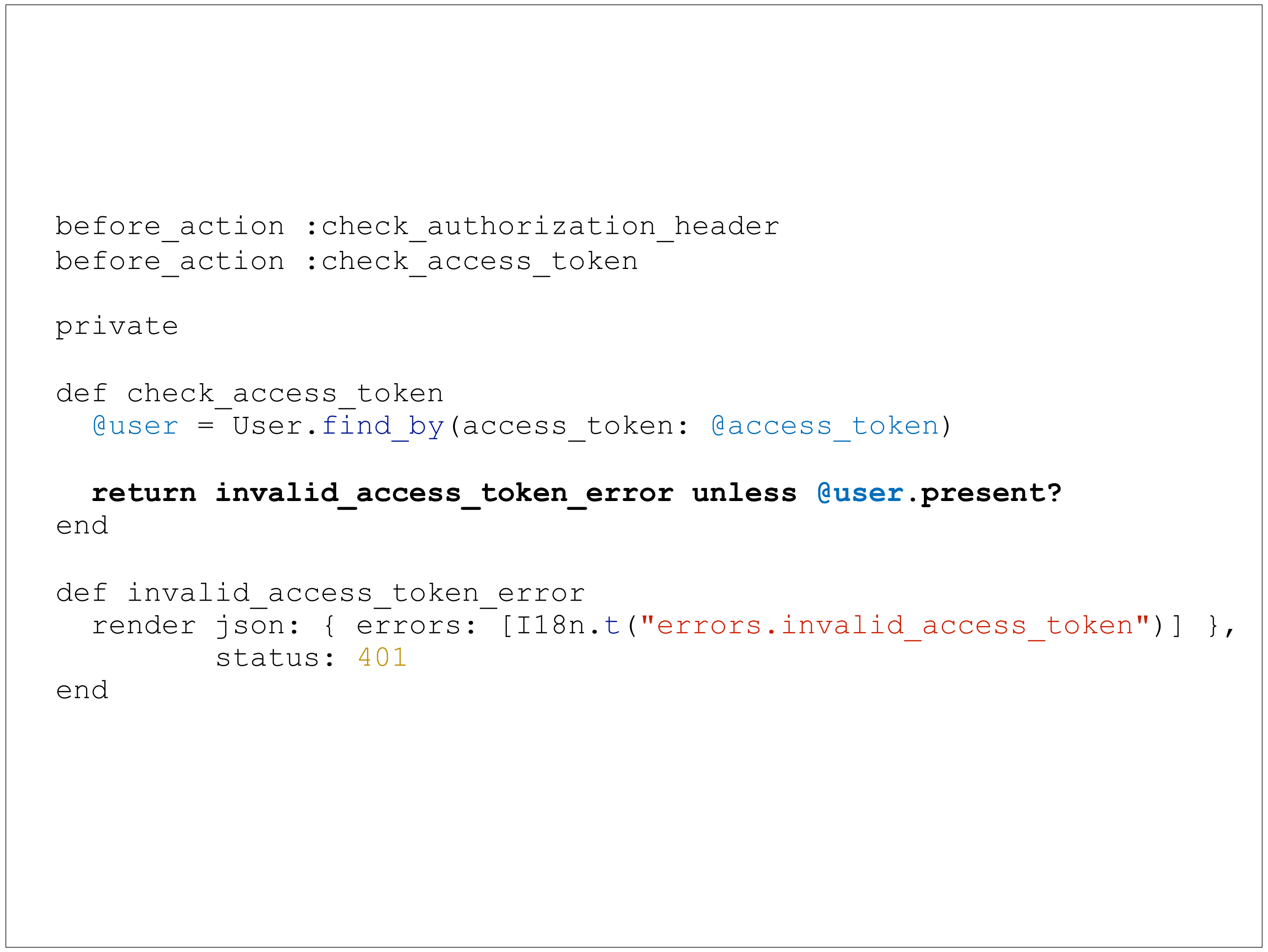
Now, having extracted the access token, we want to check if it’s valid.

So we add another filter, `#check_access_token`.



```
before_action :check_authorization_header
before_action :check_access_token

private

def check_access_token
  @user = User.find_by(access_token: @access_token)

  return invalid_access_token_error unless @user.present?
end

def invalid_access_token_error
  render json: { errors: [I18n.t("errors.invalid_access_token")] },
         status: 401
end
```

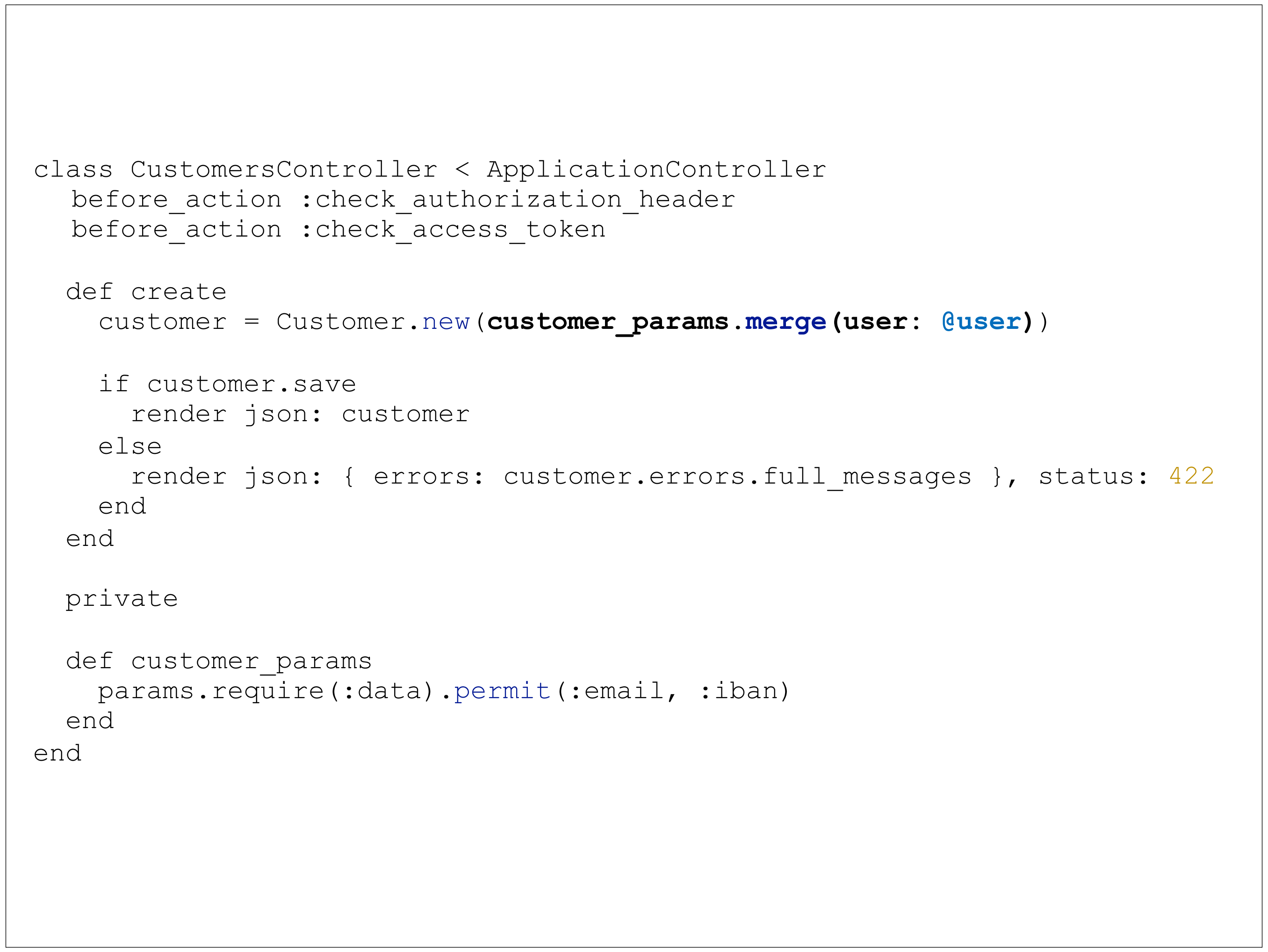
In there, we look up the access token which we've already saved to an instance variable, and store the authenticating user in an instance variable, @user.



```
before_action :check_authorization_header
before_action :check_access_token

private

def check_access_token
  @user = User.find_by(access_token: @access_token)

  return invalid_access_token_error unless @user.present?
end

def invalid_access_token_error
  render json: { errors: [I18n.t("errors.invalid_access_token")] },
         status: 401
end
```

If there’s no match, we return an error.

Otherwise, we’ll reach the end of our filters, and Rails will move on to our controller action.



```ruby
class CustomersController < ApplicationController
  before_action :check_authorization_header
  before_action :check_access_token

  def create
    customer = Customer.new(customer_params.merge(user: @user))

    if customer.save
      render json: customer
    else
      render json: { errors: customer.errors.full_messages }, status: 422
    end
  end

  private

  def customer_params
    params.require(:data).permit(:email, :iban)
  end
end
```

Now, in the controller action, we can make use of the user stored in the @user instance variable.

There are tonnes of other things like this which we might want to do in filters - for example, validating input against our API schema, recording statistics about who is using which API or enforcing users' permissions.

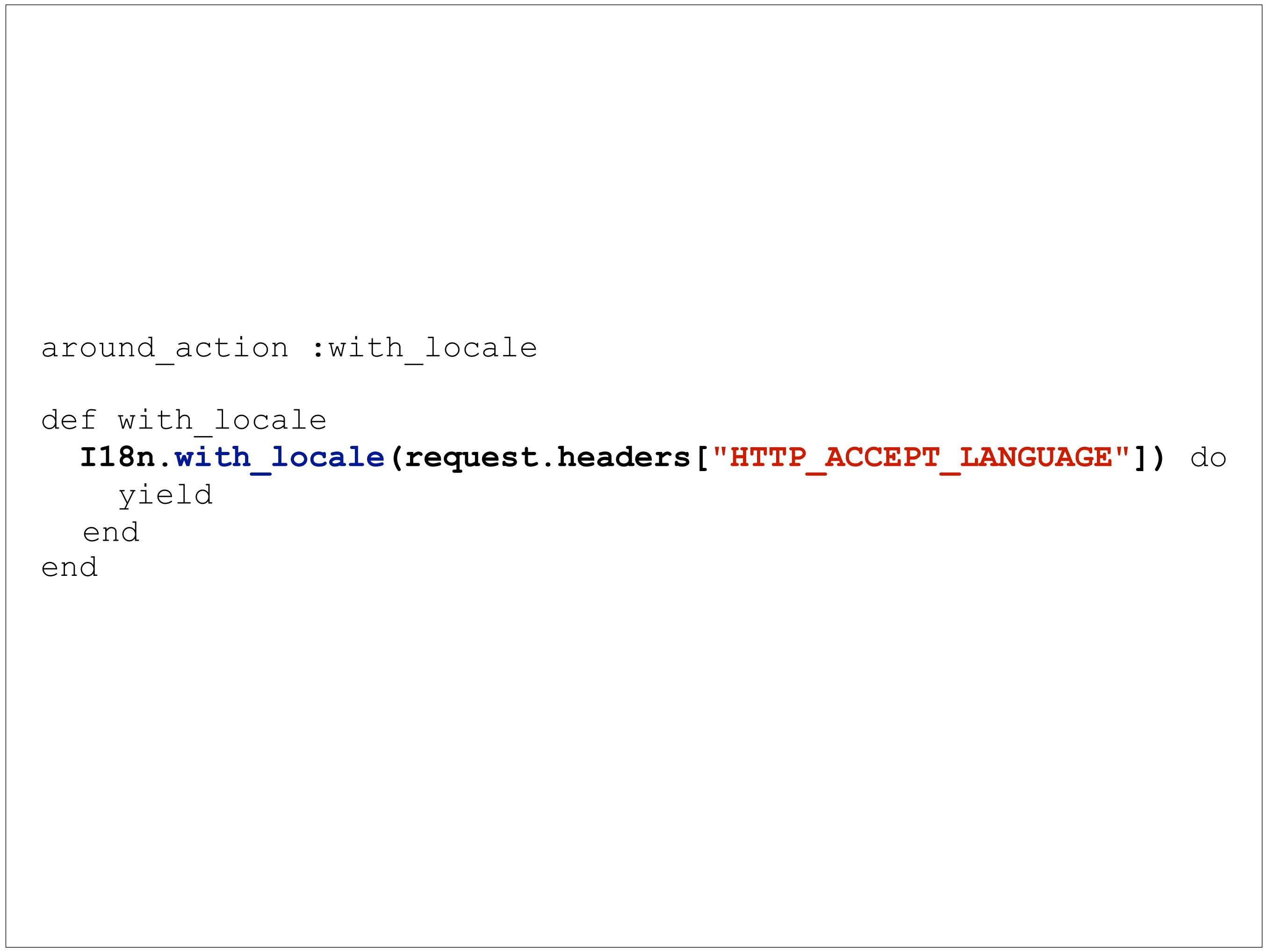
We'll look at just one more real-world example from GoCardless. Let's say that we also want to translate the API's error messages into the user's language.

```
POST /customers HTTP/1.1

Accept-Language: uk-UA
Authorization: Bearer my_access_token
Content-Type: application/json
Host: api.gocardless.com

{
  "data": {
    “email": "tim@gocardless.com",
     "iban": “GB60BARC20000055779911"
  }
}
```

There’s a standard header called Accept-Language where you can specify what language you want the response to be in. So, let’s say we want our response in Ukrainian.

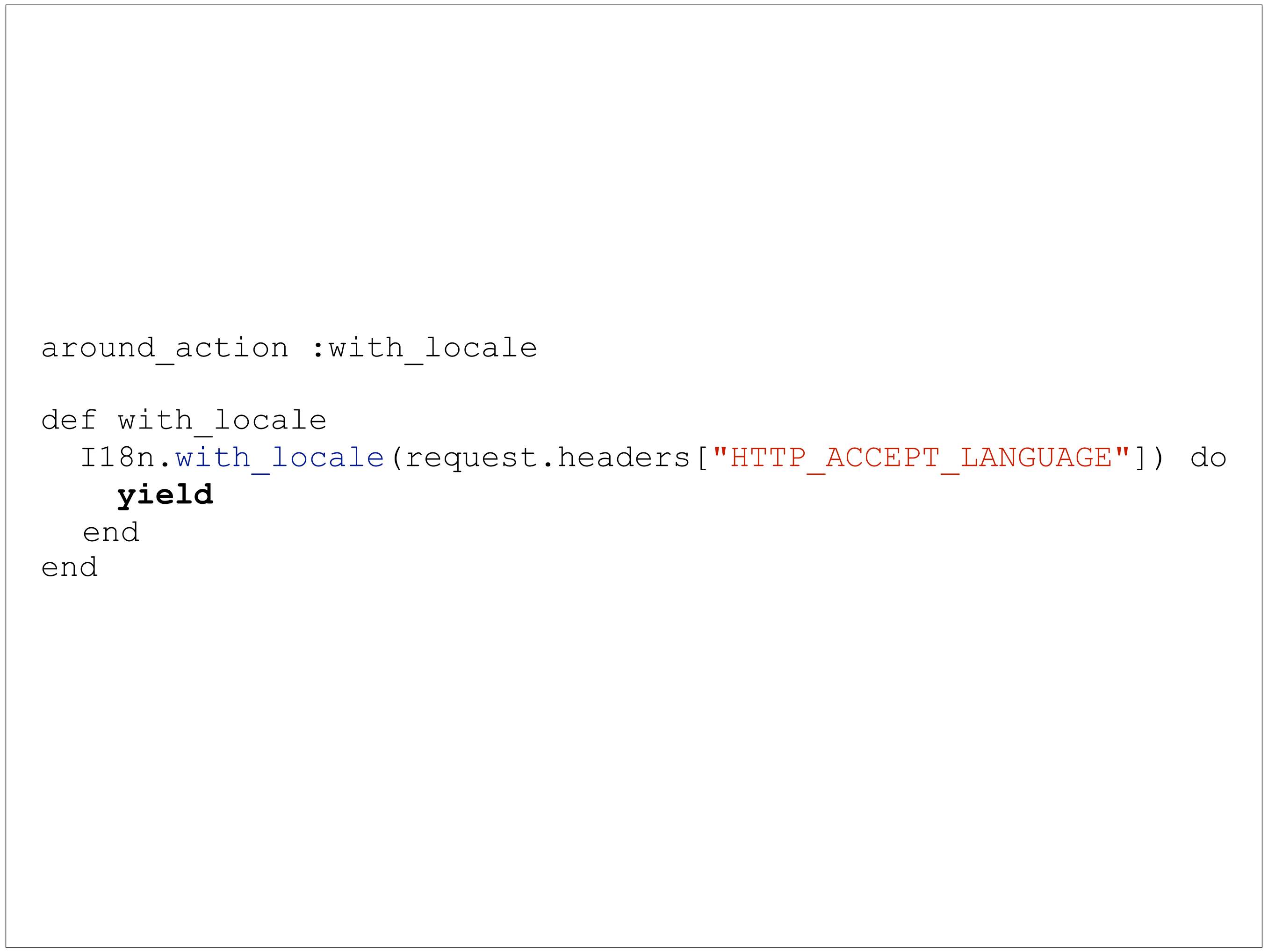
```
around_action :with_locale

def with_locale
  I18n.with_locale(request.headers["HTTP_ACCEPT_LANGUAGE"]) do
    yield
  end
end
```

We’ll add another filter to our controller to handle this, this time using `around_action`, which lets us, as the same suggests, run some code “around” our action.



```
around_action :with_locale

def with_locale
  I18n.with_locale(request.headers["HTTP_ACCEPT_LANGUAGE"]) do
    yield
  end
end
```

We’ll set Rails’s built-in I18n functionality to use the language specified in the request



```
around_action :with_locale

def with_locale
  I18n.with_locale(request.headers["HTTP_ACCEPT_LANGUAGE"]) do
    yield
  end
end
```

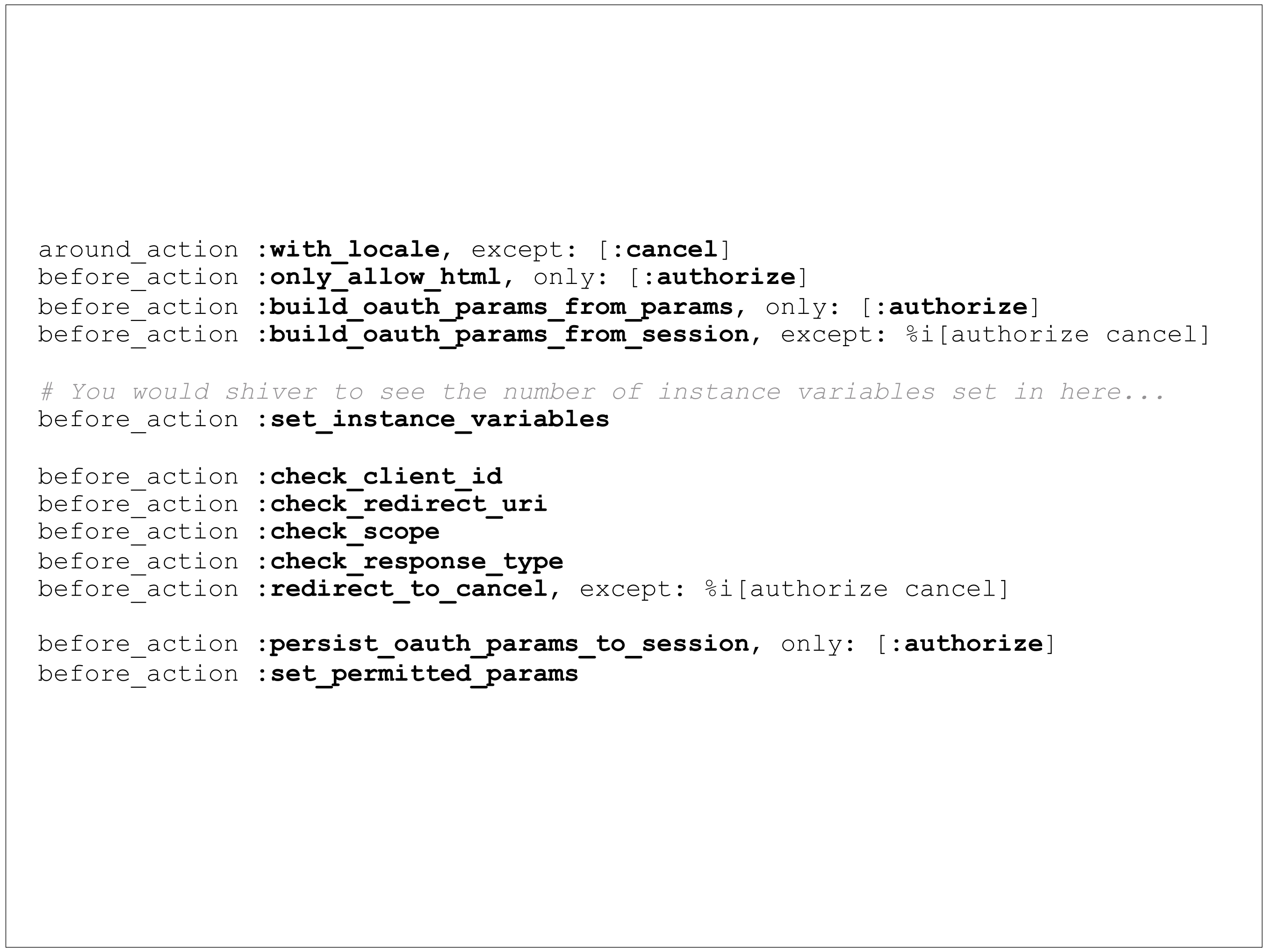
And with that setting configured, we’ll continue on by calling `yield`.

Welcome to

`before_action`

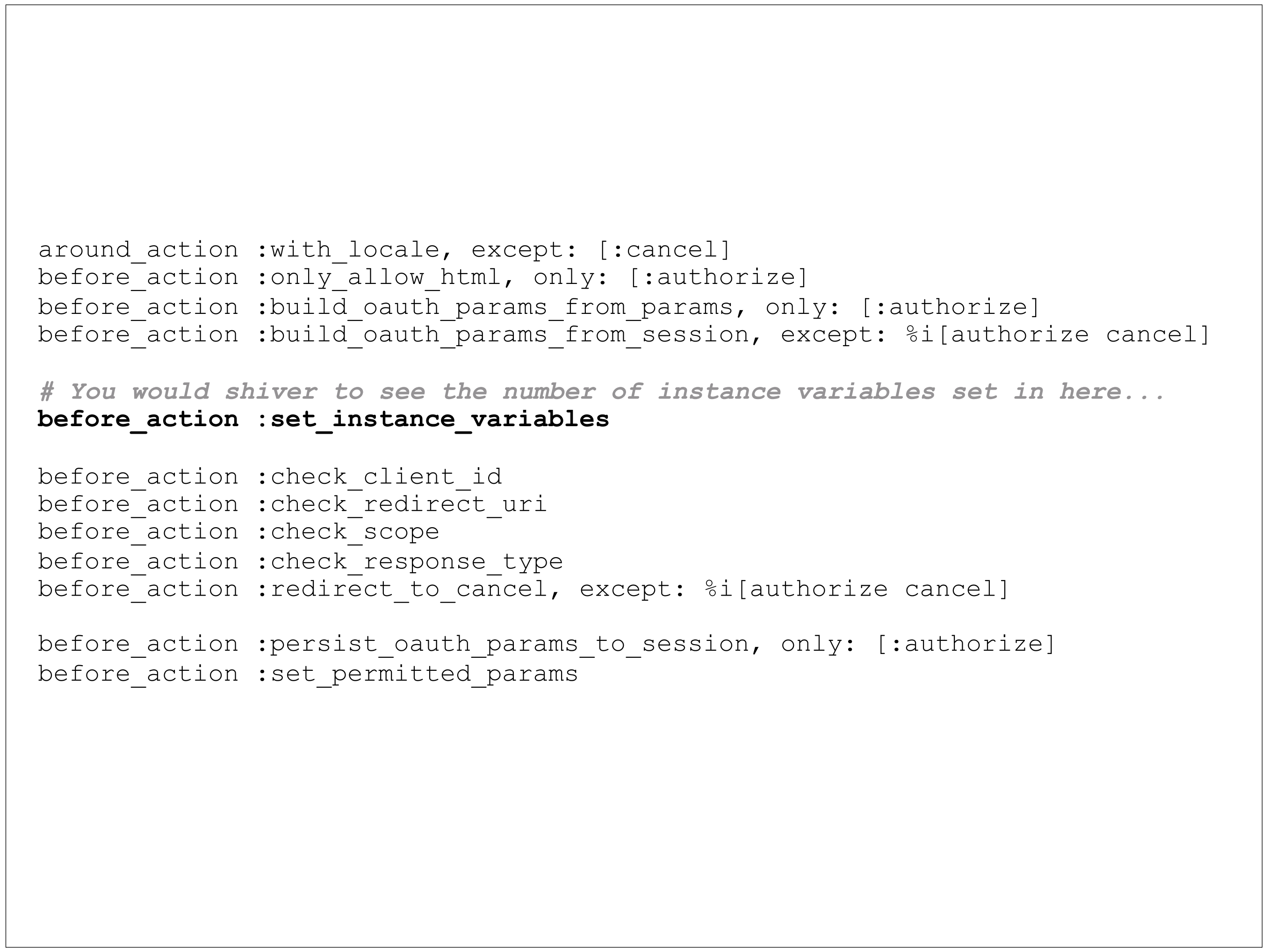
hell

Very quickly, this approach of adding more and more filters gets out of control, leading to a world of pain and unmaintainability.



```
around_action :with_locale, except: [:cancel]
before_action :only_allow_html, only: [:authorize]
before_action :build_oauth_params_from_params, only: [:authorize]
before_action :build_oauth_params_from_session, except: %i[authorize cancel]

# You would shiver to see the number of instance variables set in here...
before_action :set_instance_variables

before_action :check_client_id
before_action :check_redirect_uri
before_action :check_scope
before_action :check_response_type
before_action :redirect_to_cancel, except: %i[authorize cancel]

before_action :persist_oauth_params_to_session, only: [:authorize]
before_action :set_permitted_params
```

Here’s a real example from a controller at GoCardless which used to power our OAuth flow, with no less than 12 filters in the worst case.



```ruby
around_action :with_locale, except: [:cancel]
before_action :only_allow_html, only: [:authorize]
before_action :build_oauth_params_from_params, only: [:authorize]
before_action :build_oauth_params_from_session, except: %i[authorize cancel]

# You would shiver to see the number of instance variables set in here...
before_action :set_instance_variables

before_action :check_client_id
before_action :check_redirect_uri
before_action :check_scope
before_action :check_response_type
before_action :redirect_to_cancel, except: %i[authorize cancel]

before_action :persist_oauth_params_to_session, only: [:authorize]
before_action :set_permitted_params
```

Our code gets really painful, really quickly.

Endpoint-specific logic
vs.
Reusable logic

What’s happened here? Why does our code get out of control so quickly? Where do all of these filters come from?

The `#create` method in our controller contains the endpoint-specific logic - the code, which you don’t need to reuse, to create a customer based on the incoming request.

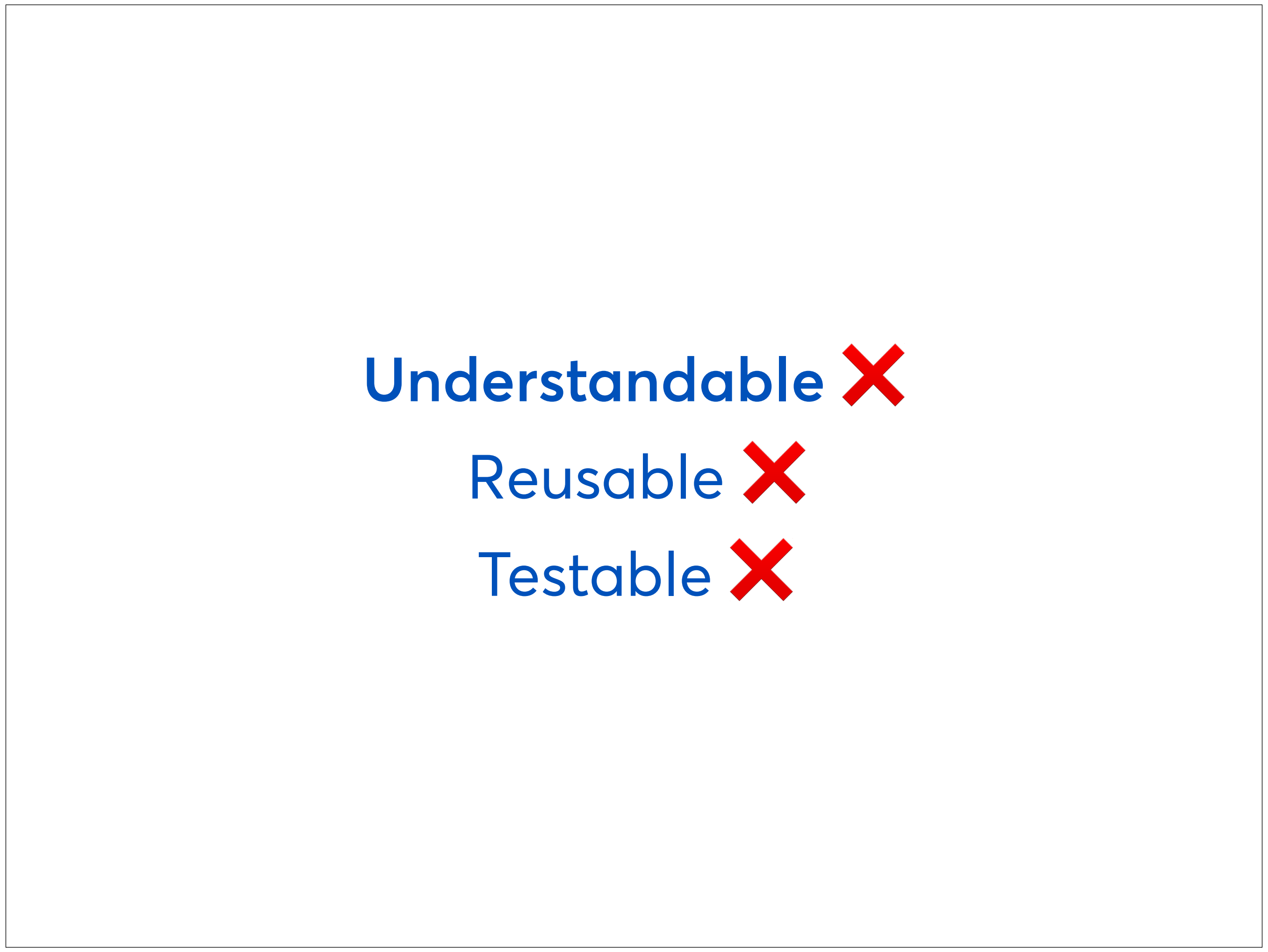
This code tends to stay fairly simple, and there are simple and well-known solutions if it gets more complicated, for example moving complex logic to service objects.

What we’ve been adding is reusable code which we’d use across our API - things like authentication and internationalisation.

APIs are designed to provide a rigid, consistent interface for users, which means that you’re likely to have more shared, reusable logic which you use to provide that interface, from authentication to validation.

What’s the problem? 🤔

Even just by adding a few of these filters, we’ve already hit three problems which make our application less maintainable:



Understandable ❌

Reusable ❌

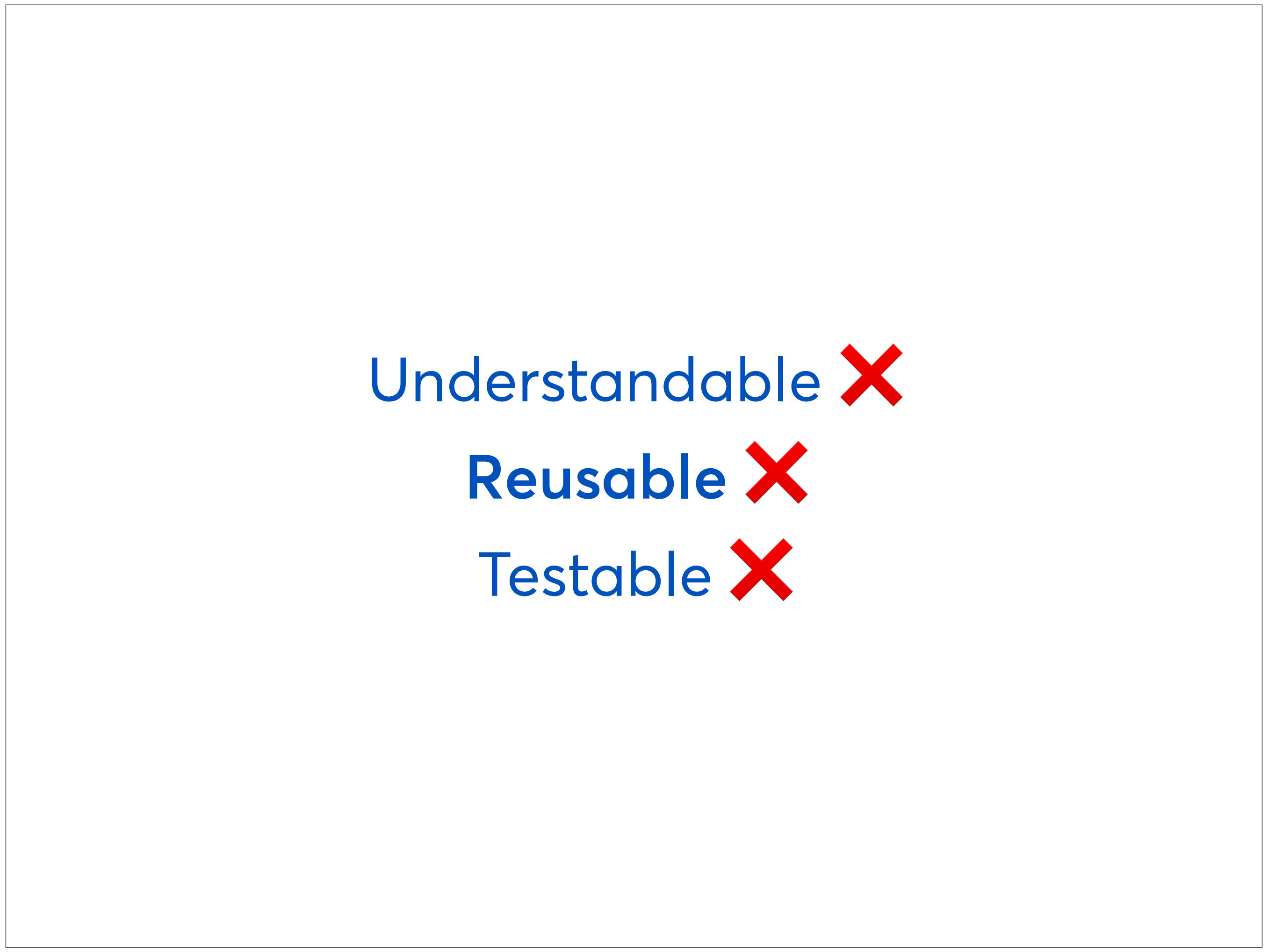
Testable ❌

Our code is hard to understand, particularly in the way it uses state.

We naturally want to split our logic into different methods to make our code as clear as possible. For example, we split parsing the access token from checking if it exists.

We need to share data between those steps, and make data available from the filters to the controller action itself - in this case, the authenticated user.

We end up using instance variables, which aren't easy to reason about. The flow of data isn't clear, it isn't define in one place, and dependencies aren't enforced. Once you have lots of filters, you find yourself scrolling through your code constantly trying to understand where data comes from.



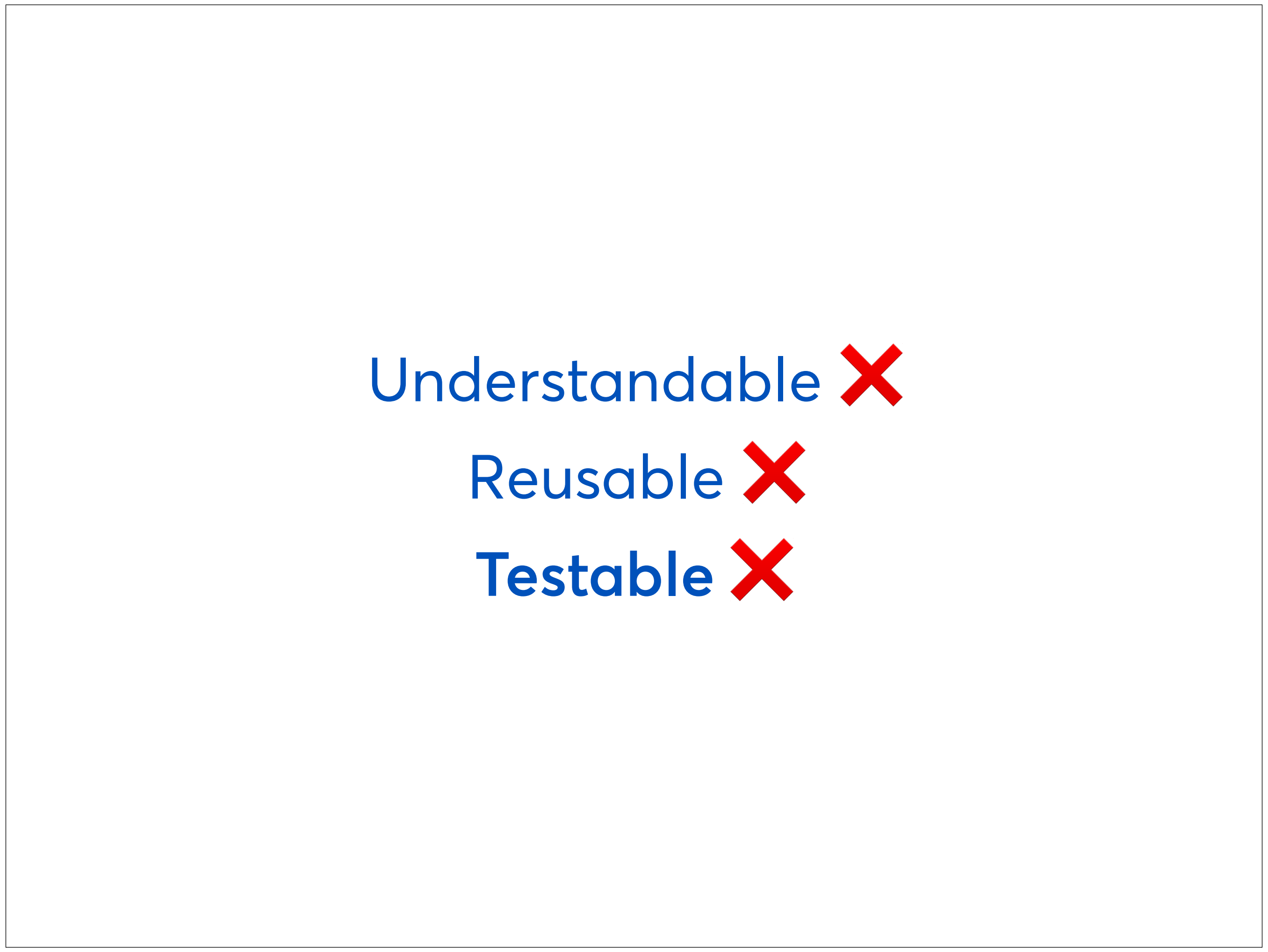
Understandable ❌

Reusable ❌

Testable ❌

This code lives in our controller. It’s difficult to share it, reuse it or adapt it.

We’ll just end up duplicating this code when we want to add another API - for example, an API to collect a payment from a customer.


Understandable ✗

Reusable ✗

Testable ✗

And our code will be hard to test.

We want to be able to test all the edge cases in our logic. For example, what happens if the Authorization header is invalid? What happens if a language we don't support is specified?

We want to be able to test the state that is constructed (i.e. what data do we fetch and store in instance variables to use later), as well as any response sent to the user.

This approach forces us to test that in the form of an integration test. We can't exercise our authentication or internationalisation logic on its own since it's just part of the controller, tied to an action.

The single responsibility principle

The SOLID principles are a series of software engineering design principles which help to make our code more understandable, more reusable and more testable.

One of these principles is the “single responsibility principle”, which is at the heart of the middleware pattern.

Understandable ✅

Reusable ✅

Testable ✅

The single responsibility principle helps us to make our code more understandable, more testable and more reusable.

"Every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class"

The single responsibility principle says that “every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class”.

In short, we want to turn each of our filters into a single class with a single responsibility. This makes our code much easier to test, understand and reuse.

Is there a Rails way?

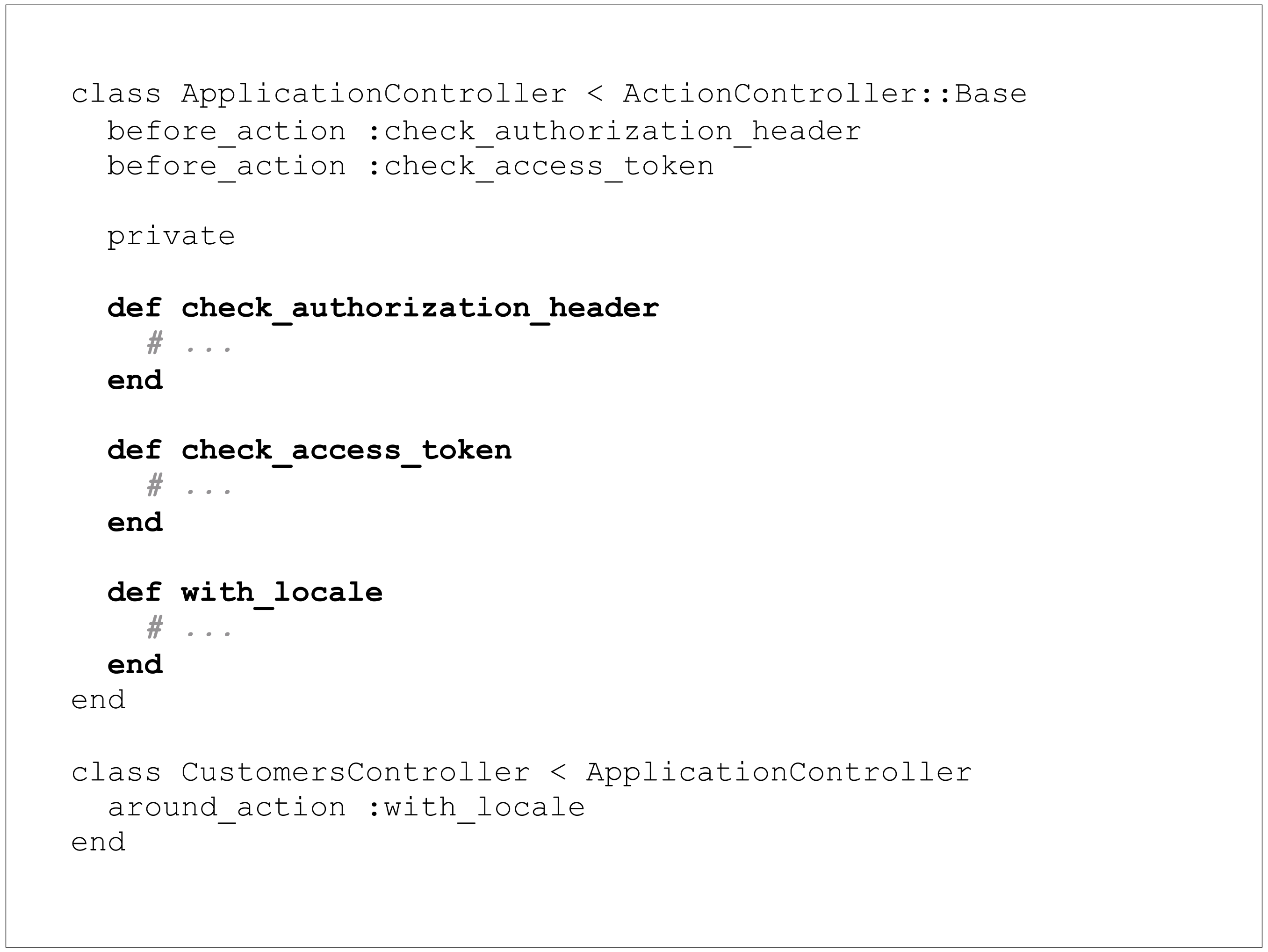
Before we dive into the middleware pattern, let’s look at the Rails way.

So far, we’ve been using filters defined within our controller. This is certainly better than adding our reusable logic to each controller action, but as we’ve seen, there are still some maintainability challenges.

The Rails documentation offers a few options if you want to use filters that get you closer to the single responsibility principle. Let’s take a look.

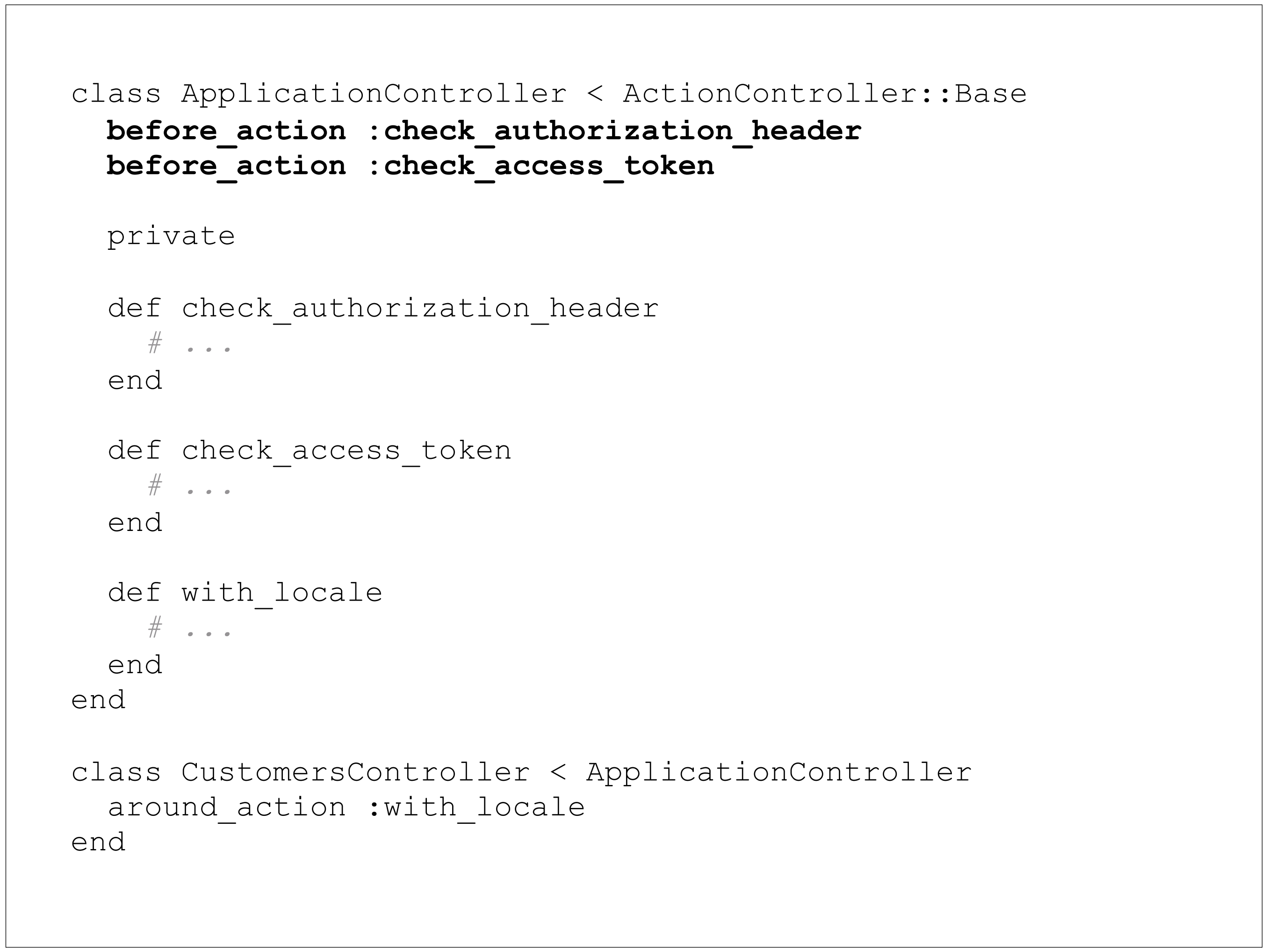
Move our filters from `CustomersController` to `ApplicationController`

First off, we can make our filters more reusable by moving them to `ApplicationController`.



```ruby
class ApplicationController < ActionController::Base
  before_action :check_authorization_header
  before_action :check_access_token

  private

  def check_authorization_header
    # ...
  end

  def check_access_token
    # ...
  end

  def with_locale
    # ...
  end
end

class CustomersController < ApplicationController
  around_action :with_locale
end
```

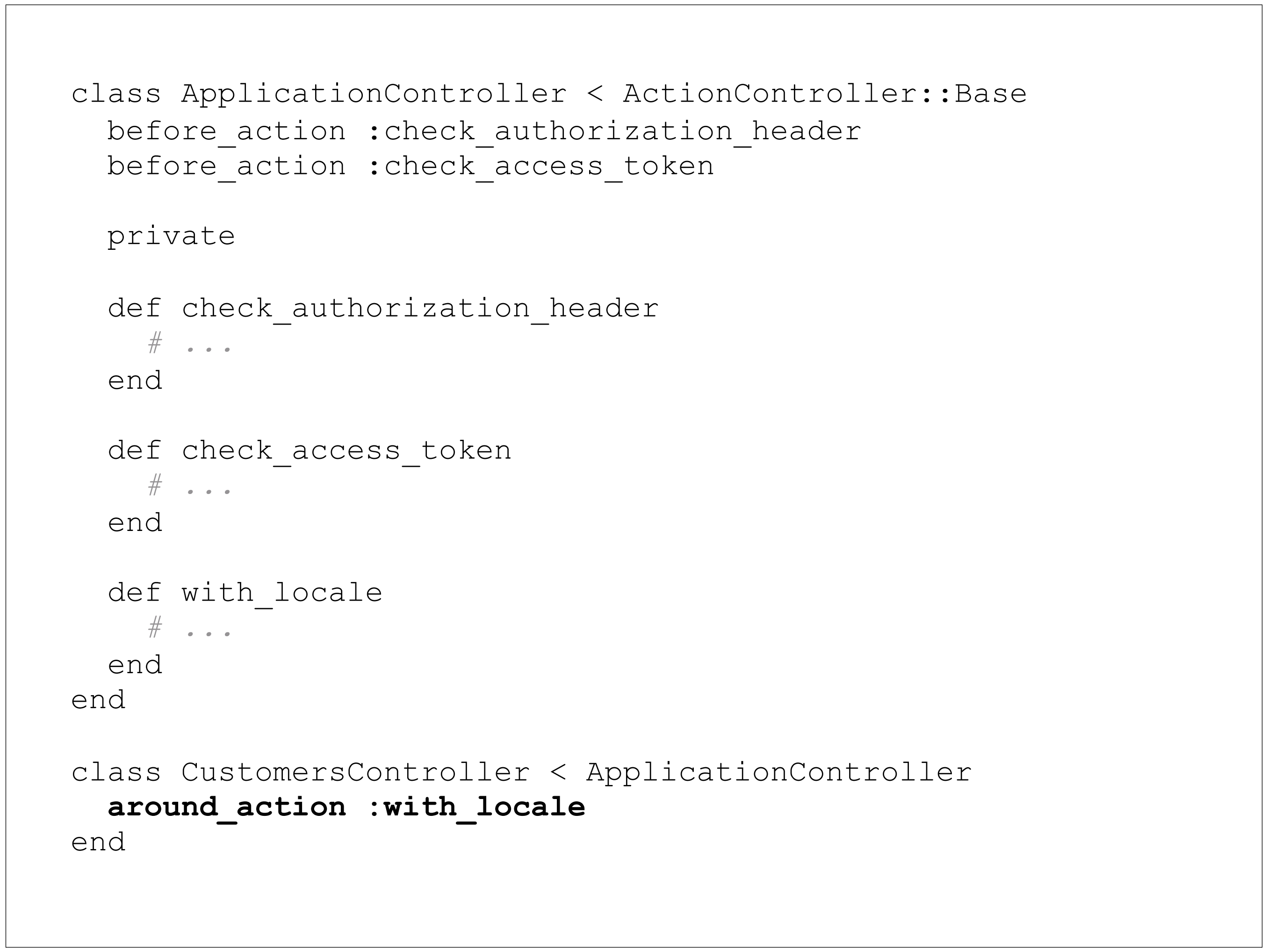
We can move our filter methods to ApplicationController.



```
class ApplicationController < ActionController::Base
  before_action :check_authorization_header
  before_action :check_access_token

  private

  def check_authorization_header
    # ...
  end

  def check_access_token
    # ...
  end

  def with_locale
    # ...
  end
end

class CustomersController < ApplicationController
  around_action :with_locale
end
```

We then have two options for actually using the filters.

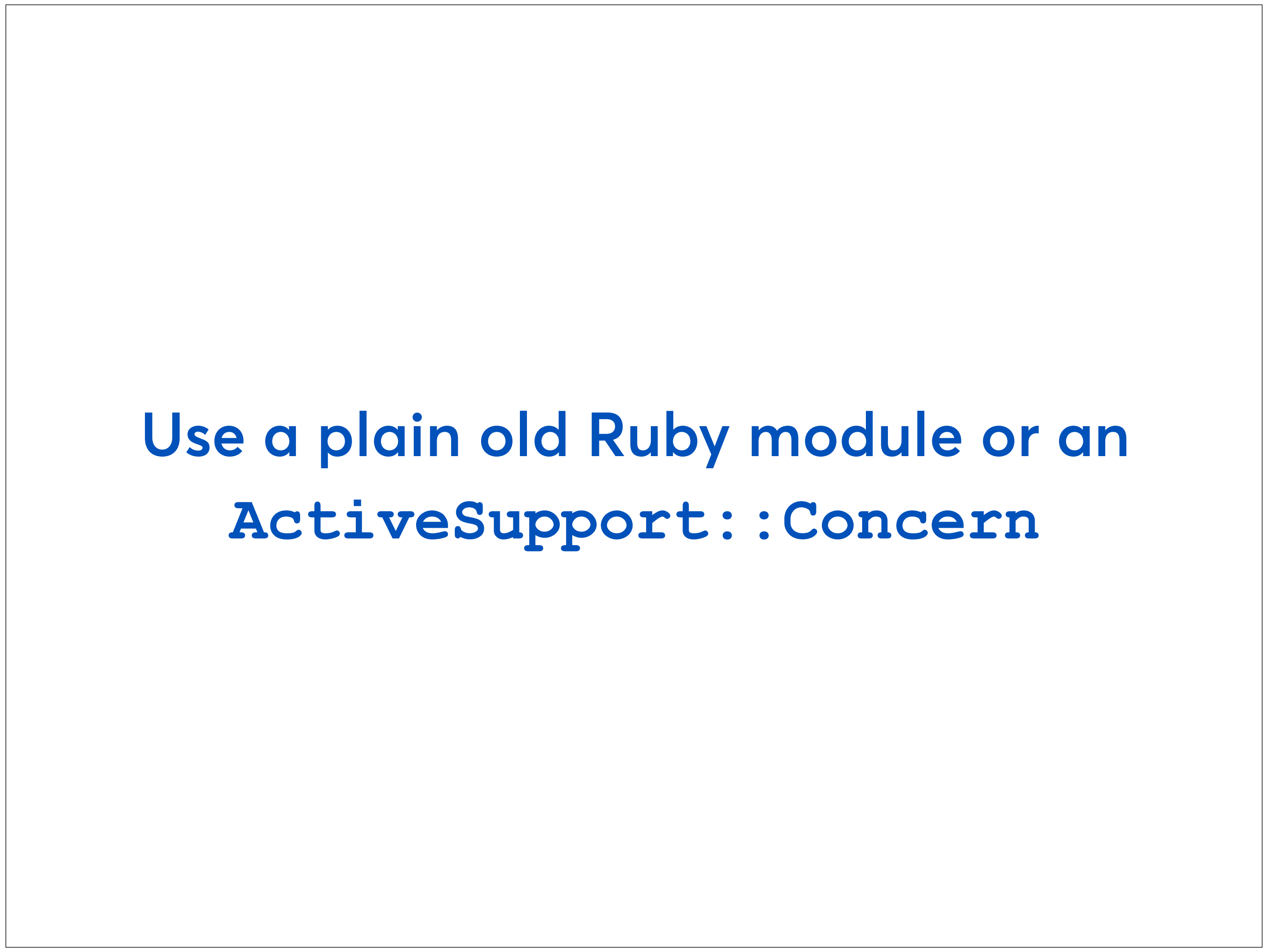
We can use `before_action` in ApplicationController if we’re really sure we want these filters everywhere.



```ruby
class ApplicationController < ActionController::Base
  before_action :check_authorization_header
  before_action :check_access_token

  private

  def check_authorization_header
    # ...
  end

  def check_access_token
    # ...
  end

  def with_locale
    # ...
  end
end

class CustomersController < ApplicationController
  around_action :with_locale
end
```

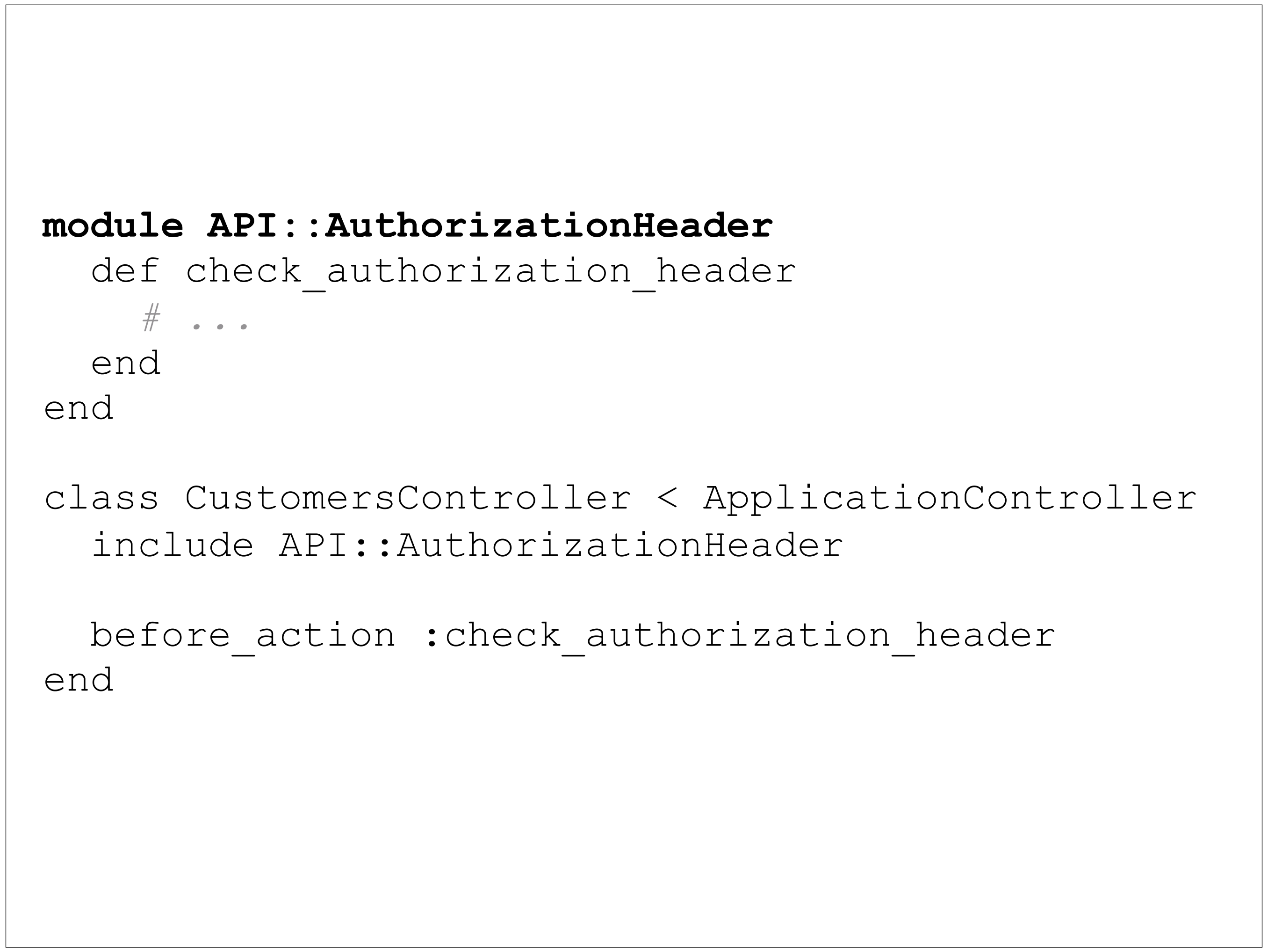
Or if we want a bit more control, we can add them in CustomersController.

I would recommend avoiding doing a mix of both - you'll be left very confused and unsure what order your filters will run in.



Use a plain old Ruby module or an `ActiveSupport::Concern`

Our second option is to move the filter methods to their own modules, and then include those modules into our controller.



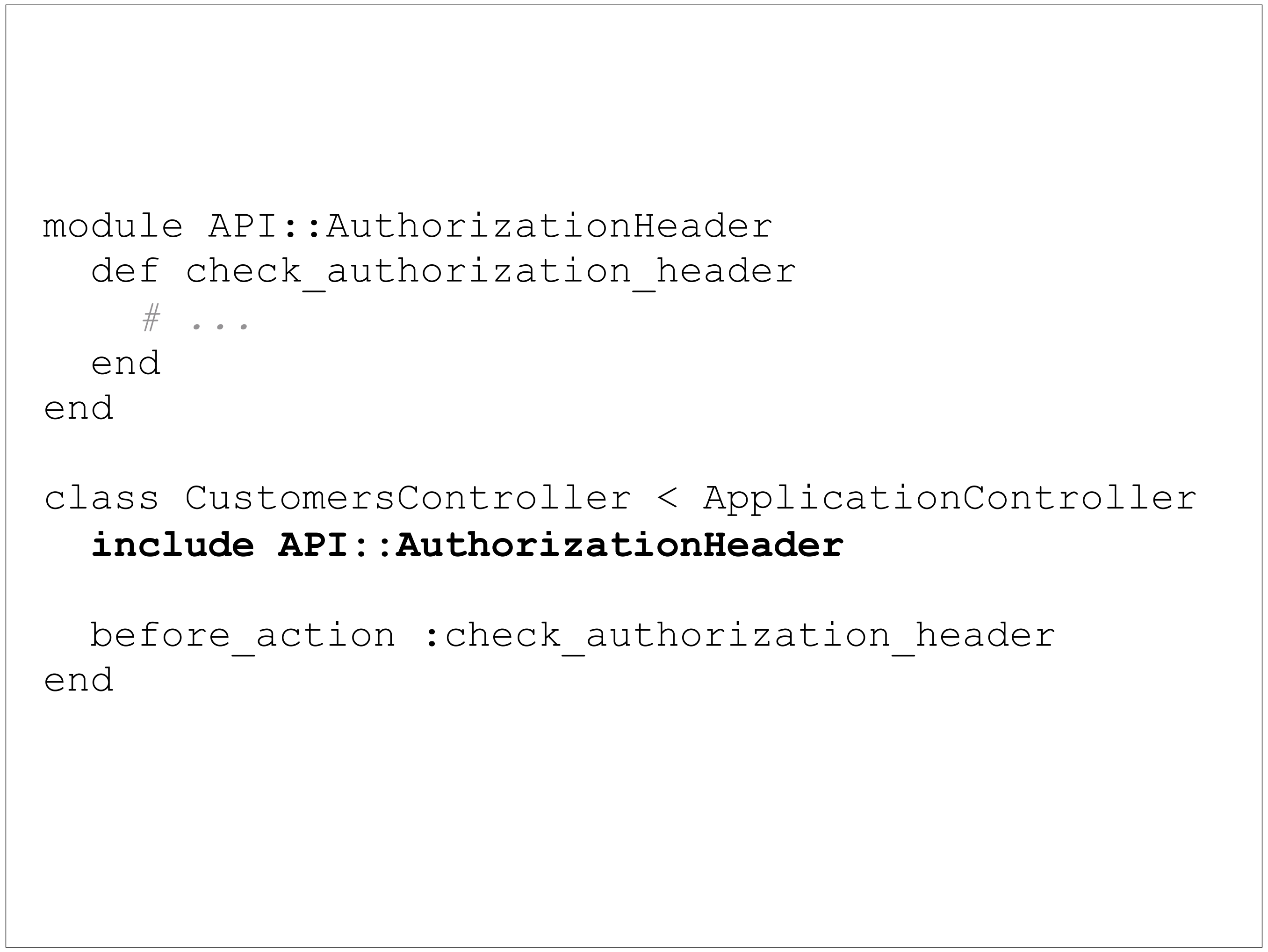
```ruby
module API::AuthorizationHeader
  def check_authorization_header
    # ...
  end
end

class CustomersController < ApplicationController
  include API::AuthorizationHeader

  before_action :check_authorization_header
end
```

This is the standard way in Ruby to share code between classes.

We can create a separate module

```
module API::AuthorizationHeader
  def check_authorization_header
    # ...
  end
end

class CustomersController < ApplicationController
  include API::AuthorizationHeader

  before_action :check_authorization_header
end
```

And define our method there which we want to be shared.

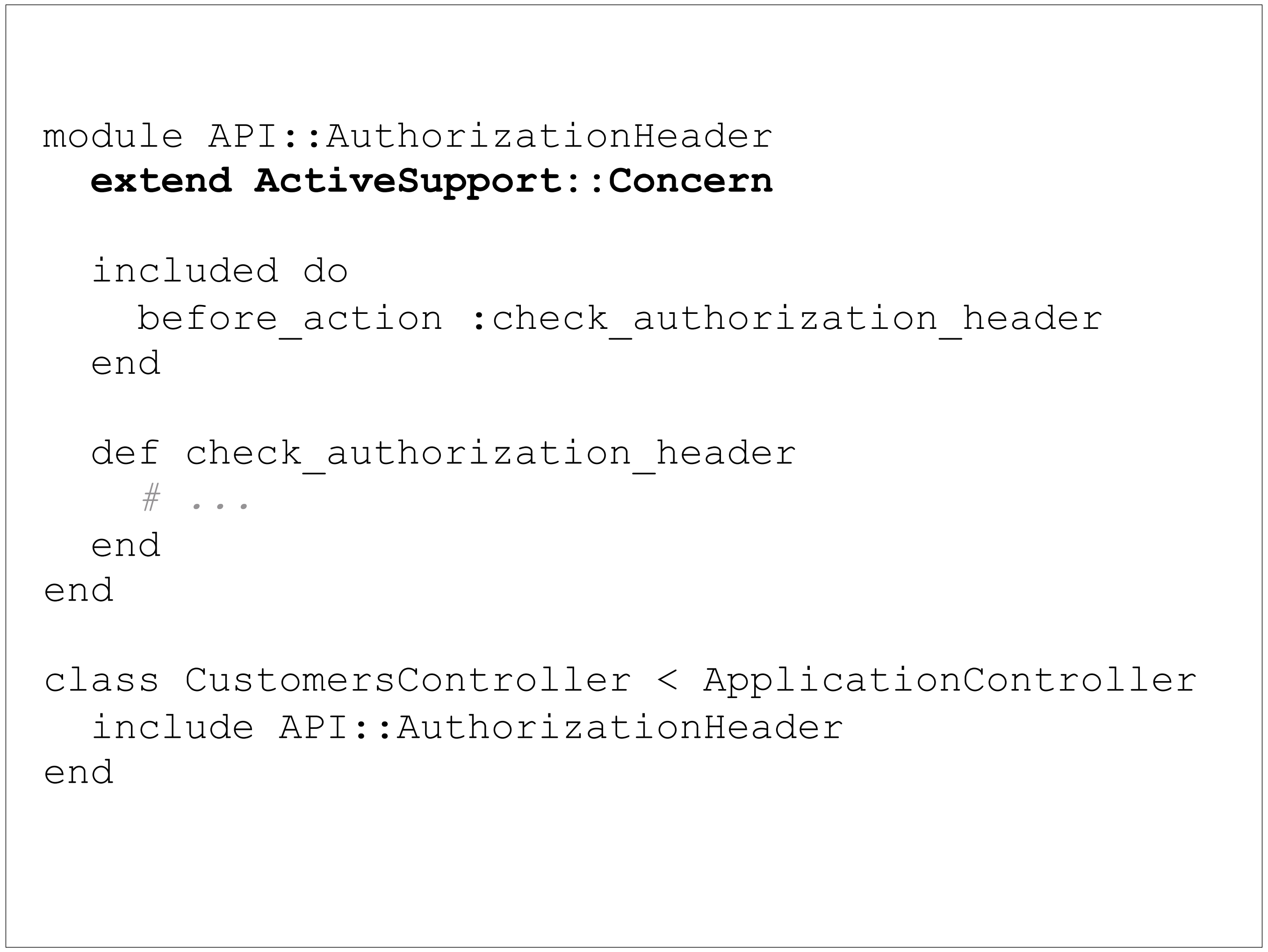


```ruby
module API::AuthorizationHeader
  def check_authorization_header
    # ...
  end
end

class CustomersController < ApplicationController
  include API::AuthorizationHeader

  before_action :check_authorization_header
end
```

Then, we just include our module in our controller, and it'll behave just like we define the method in the controller itself — so we can just use `before_action`.

```
module API::AuthorizationHeader
  def check_authorization_header
    # ...
  end
end

class CustomersController < ApplicationController
  include API::AuthorizationHeader

  before_action :check_authorization_header
end
```

so we can just use `before_action`.



```ruby
module API::AuthorizationHeader
  extend ActiveSupport::Concern

  included do
    before_action :check_authorization_header
  end

  def check_authorization_header
    # ...
  end
end

class CustomersController < ApplicationController
  include API::AuthorizationHeader
end
```

Alternatively if we want to be extra clever, we can use `ActiveSupport::Concern` which lets you automatically add the filter when you include the module.

```ruby
module API::AuthorizationHeader
  extend ActiveSupport::Concern

  included do
    before_action :check_authorization_header
  end

  def check_authorization_header
    # ...
  end
end

class CustomersController < ApplicationController
  include API::AuthorizationHeader
end
```

We include the module in our controller as usual.

```ruby
module API::AuthorizationHeader
  extend ActiveSupport::Concern

  included do
    before_action :check_authorization_header
  end

  def check_authorization_header
    # ...
  end
end

class CustomersController < ApplicationController
  include API::AuthorizationHeader
end
```

But here, we can add an `included` block to the module. This allows us to automatically set up the filter whenever this module is included in our controller.

Now, we’ve separated our logic into its own class, but it’s still not easy to test.

Use a filter module

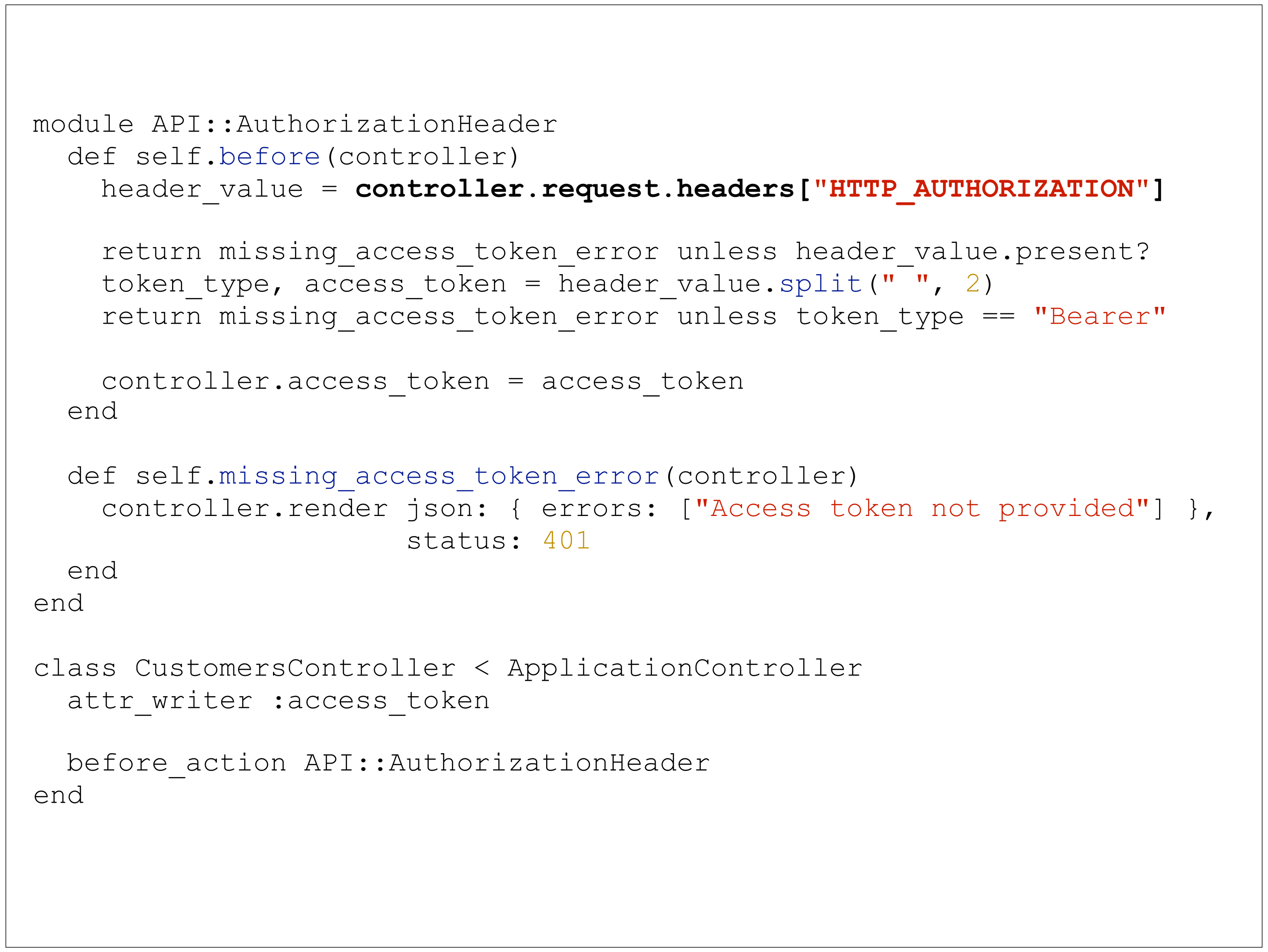
Rails’s final option, a filter class, gets us a bit closer, with the logic separated out and a more easily-testable interface.

```ruby
module API::AuthorizationHeader
  def self.before(controller)
    header_value = controller.request.headers["HTTP_AUTHORIZATION"]

    return missing_access_token_error unless header_value.present?
    token_type, access_token = header_value.split(" ", 2)
    return missing_access_token_error unless token_type == "Bearer"

    controller.access_token = access_token
  end

  def self.missing_access_token_error(controller)
    controller.render json: { errors: ["Access token not provided"] },
                      status: 401
  end
end

class CustomersController < ApplicationController
  attr_writer :access_token

  before_action API::AuthorizationHeader
end
```

We start by defining a module

```
module API::AuthorizationHeader
  def self.before(controller)
    header_value = controller.request.headers["HTTP_AUTHORIZATION"]

    return missing_access_token_error unless header_value.present?
    token_type, access_token = header_value.split(" ", 2)
    return missing_access_token_error unless token_type == "Bearer"

    controller.access_token = access_token
  end

  def self.missing_access_token_error(controller)
    controller.render json: { errors: ["Access token not provided"] },
                      status: 401
  end
end

class CustomersController < ApplicationController
  attr_writer :access_token

  before_action API::AuthorizationHeader
end
```

And then we give that module a `before` method, which gets passed a controller. We could do the same for an “after” or “around” filter, by using the obvious method name.



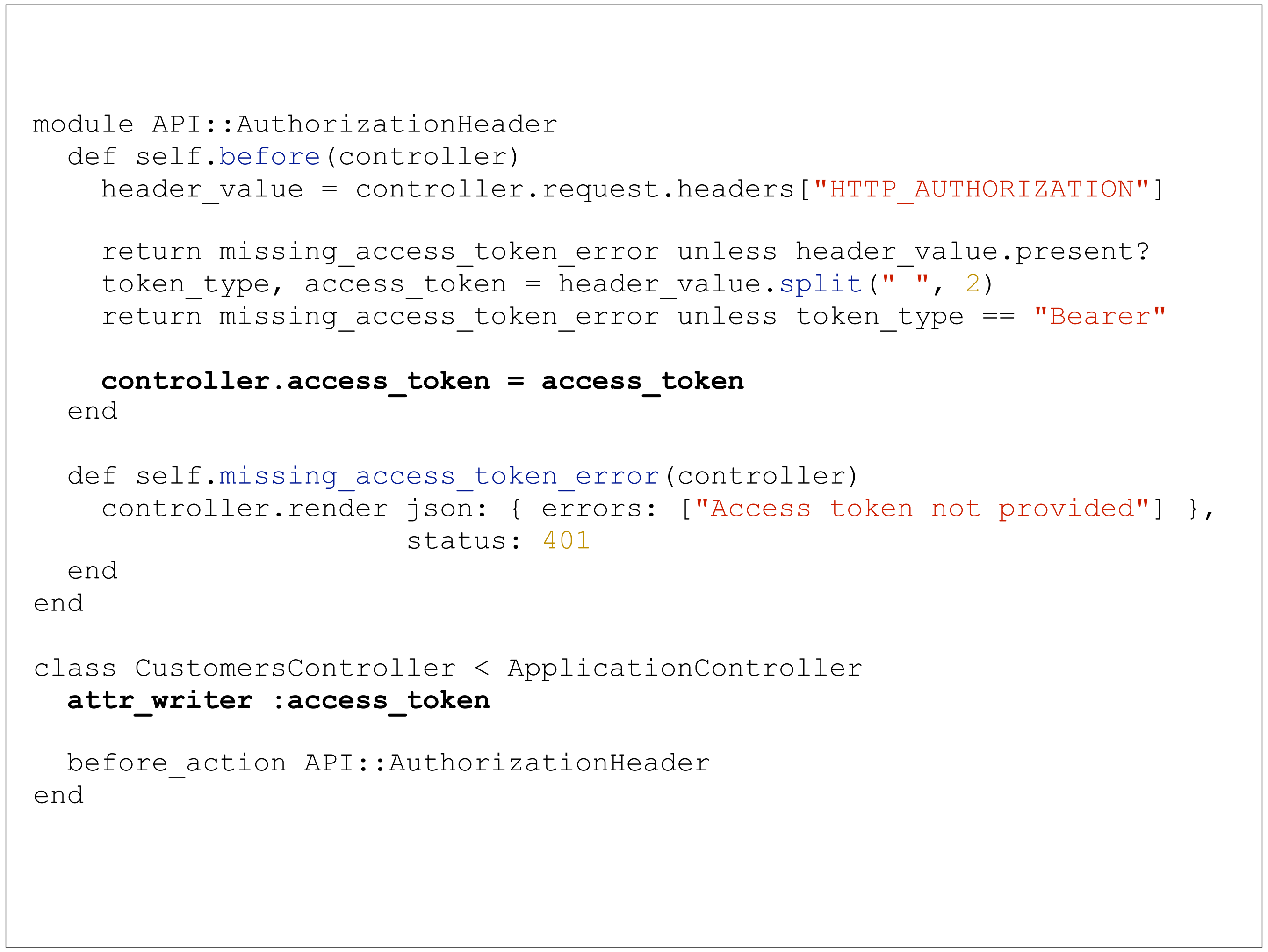
```ruby
module API::AuthorizationHeader
  def self.before(controller)
    header_value = controller.request.headers["HTTP_AUTHORIZATION"]

    return missing_access_token_error unless header_value.present?
    token_type, access_token = header_value.split(" ", 2)
    return missing_access_token_error unless token_type == "Bearer"

    controller.access_token = access_token
  end

  def self.missing_access_token_error(controller)
    controller.render json: { errors: ["Access token not provided"] },
                      status: 401
  end
end

class CustomersController < ApplicationController
  attr_writer :access_token

  before_action API::AuthorizationHeader
end
```

This means we can do everything that we could do with our previous patterns.

We can see the request - for example, we can inspect the headers

```
module API::AuthorizationHeader
  def self.before(controller)
    header_value = controller.request.headers["HTTP_AUTHORIZATION"]

    return missing_access_token_error unless header_value.present?
    token_type, access_token = header_value.split(" ", 2)
    return missing_access_token_error unless token_type == "Bearer"

    controller.access_token = access_token
  end

  def self.missing_access_token_error(controller)
    controller.render json: { errors: ["Access token not provided"] },
                      status: 401
  end
end

class CustomersController < ApplicationController
  attr_writer :access_token

  before_action API::AuthorizationHeader
end
```

We can send a response, using the controller’s `#render` method



```ruby
module API::AuthorizationHeader
  def self.before(controller)
    header_value = controller.request.headers["HTTP_AUTHORIZATION"]

    return missing_access_token_error unless header_value.present?
    token_type, access_token = header_value.split(" ", 2)
    return missing_access_token_error unless token_type == "Bearer"

    controller.access_token = access_token
  end

  def self.missing_access_token_error(controller)
    controller.render json: { errors: ["Access token not provided"] },
                      status: 401
  end
end

class CustomersController < ApplicationController
  attr_writer :access_token

  before_action API::AuthorizationHeader
end
```

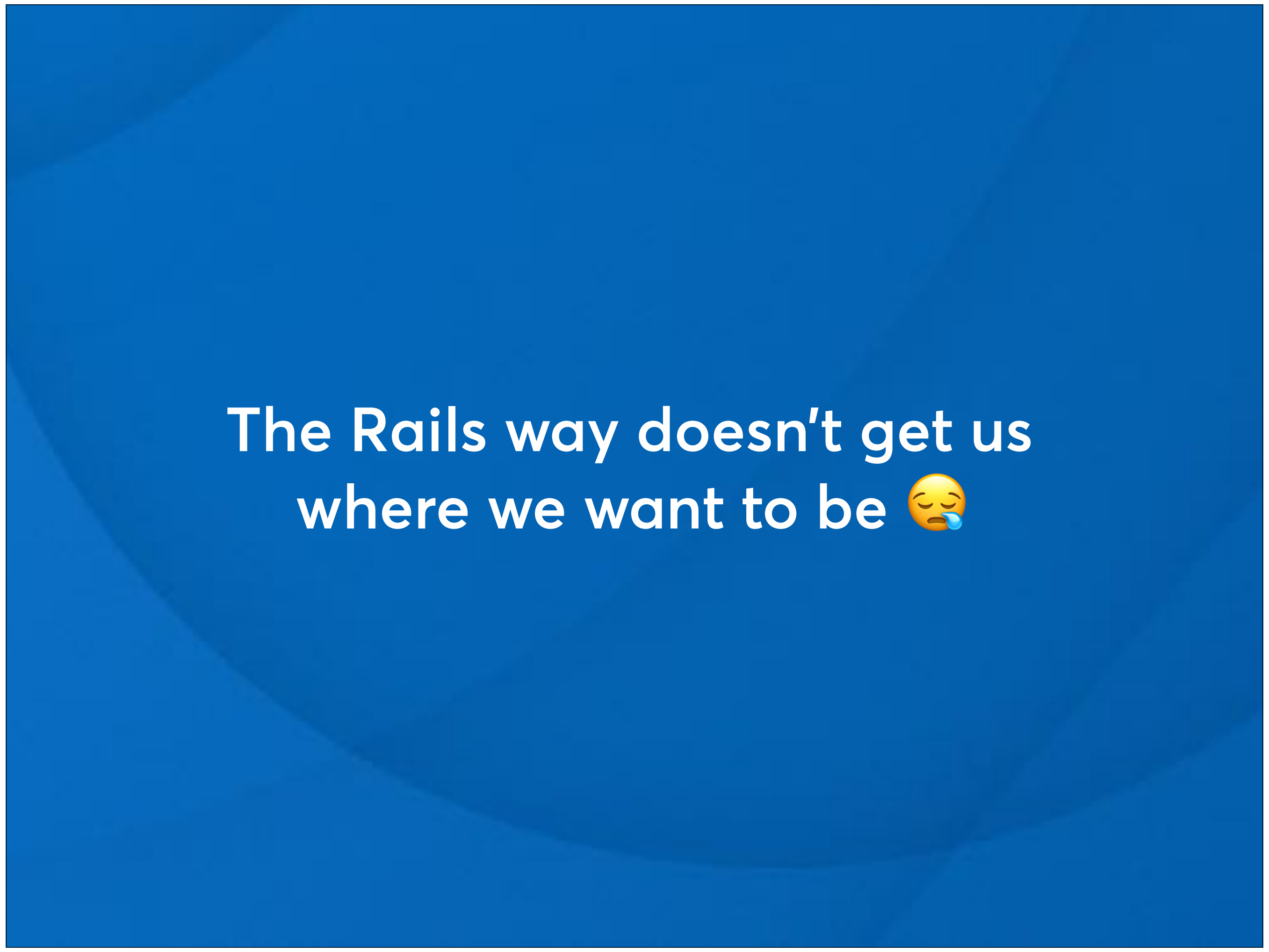
And we can even store state on the controller

```ruby
module API::AuthorizationHeader
  def self.before(controller)
    header_value = controller.request.headers["HTTP_AUTHORIZATION"]

    return missing_access_token_error unless header_value.present?
    token_type, access_token = header_value.split(" ", 2)
    return missing_access_token_error unless token_type == "Bearer"

    controller.access_token = access_token
  end

  def self.missing_access_token_error(controller)
    controller.render json: { errors: ["Access token not provided"] },
                      status: 401
  end
end

class CustomersController < ApplicationController
  attr_writer :access_token

  before_action API::AuthorizationHeader
end
```

We use this filter class by passing its name as the argument to `before_action`.

Our filter is now abstracted out and fairly simple to test, since we can pass it a fake controller and see what happens (e.g. what is returned? What state is stored on the controller?)

But using instances variables for state still leaves things quite confusing, it isn’t *that* easy to test and using a fake controller isn’t too nice.



The Rails way doesn't get us where we want to be 😪

We've now seen the maintainability problems we hit with Rails, and we haven't been able to find a good solution which follows *the Rails way*.

Let’s see how the middleware pattern can help us 🌟

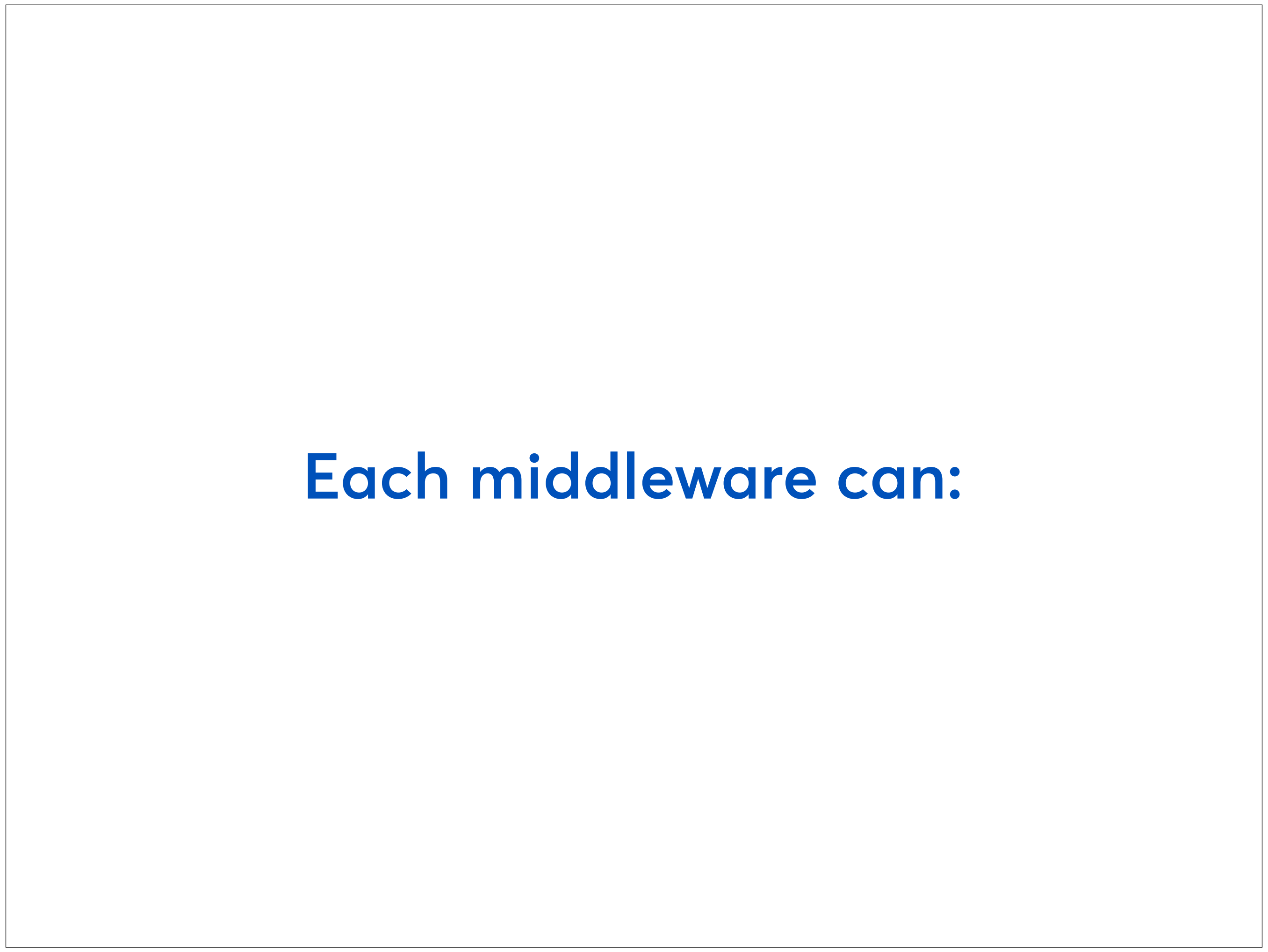
Let’s start talking about the middleware solution. Using the middleware pattern lets us write easy to understand, easy to test, easy to reuse, maintainable code. So what is it?

"Middleware is software that's assembled into an application pipeline to handle requests and responses"

Surprisingly, Microsoft's ASP.NET documentation actually has one of the best descriptions of how middleware work.

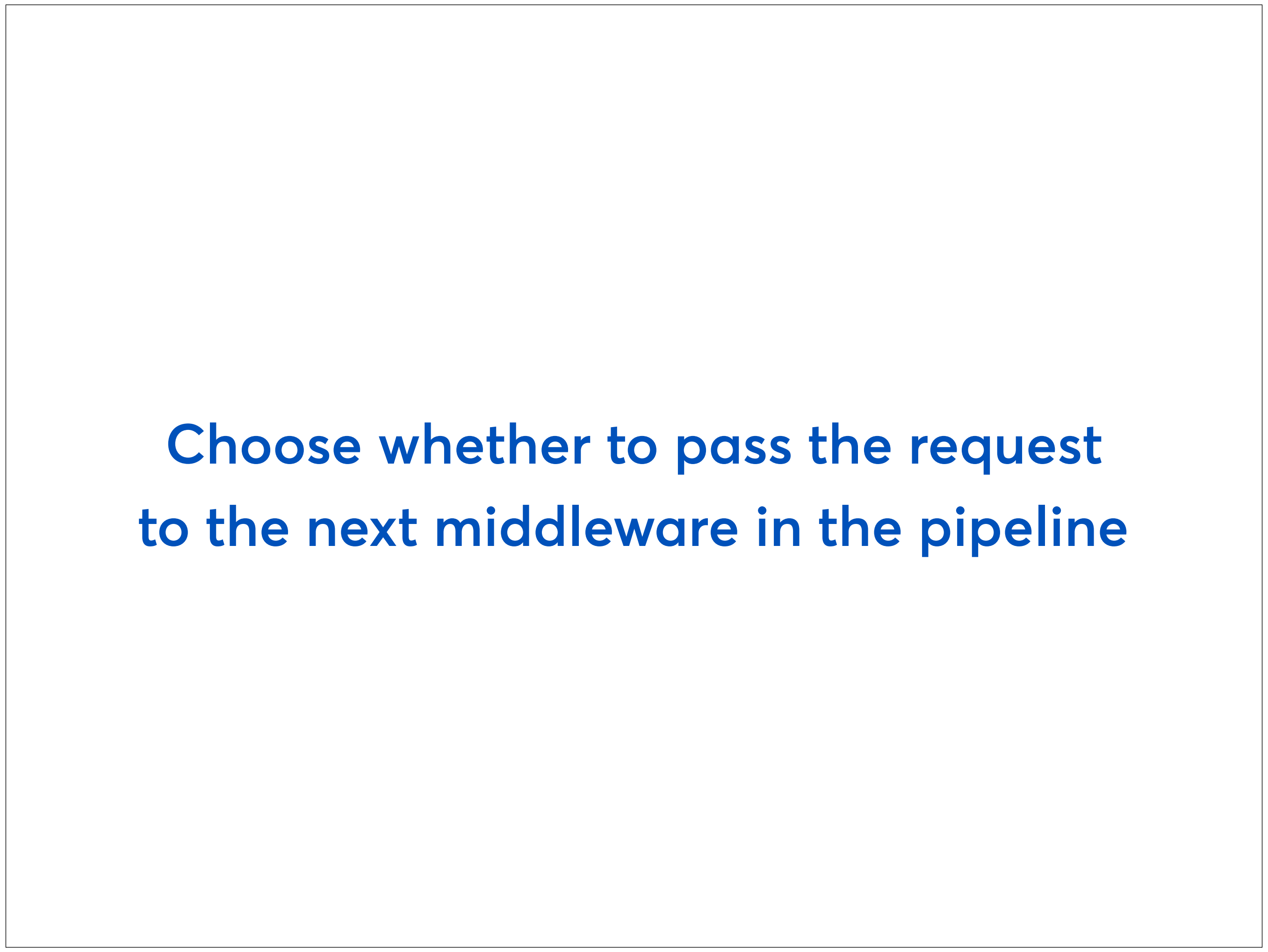
Middleware is software that's assembled into an application pipeline to handle requests and responses.

We chain together middleware into a pipeline, and the middleware in the pipeline work together to handle the request and build a response.



Each middleware can:

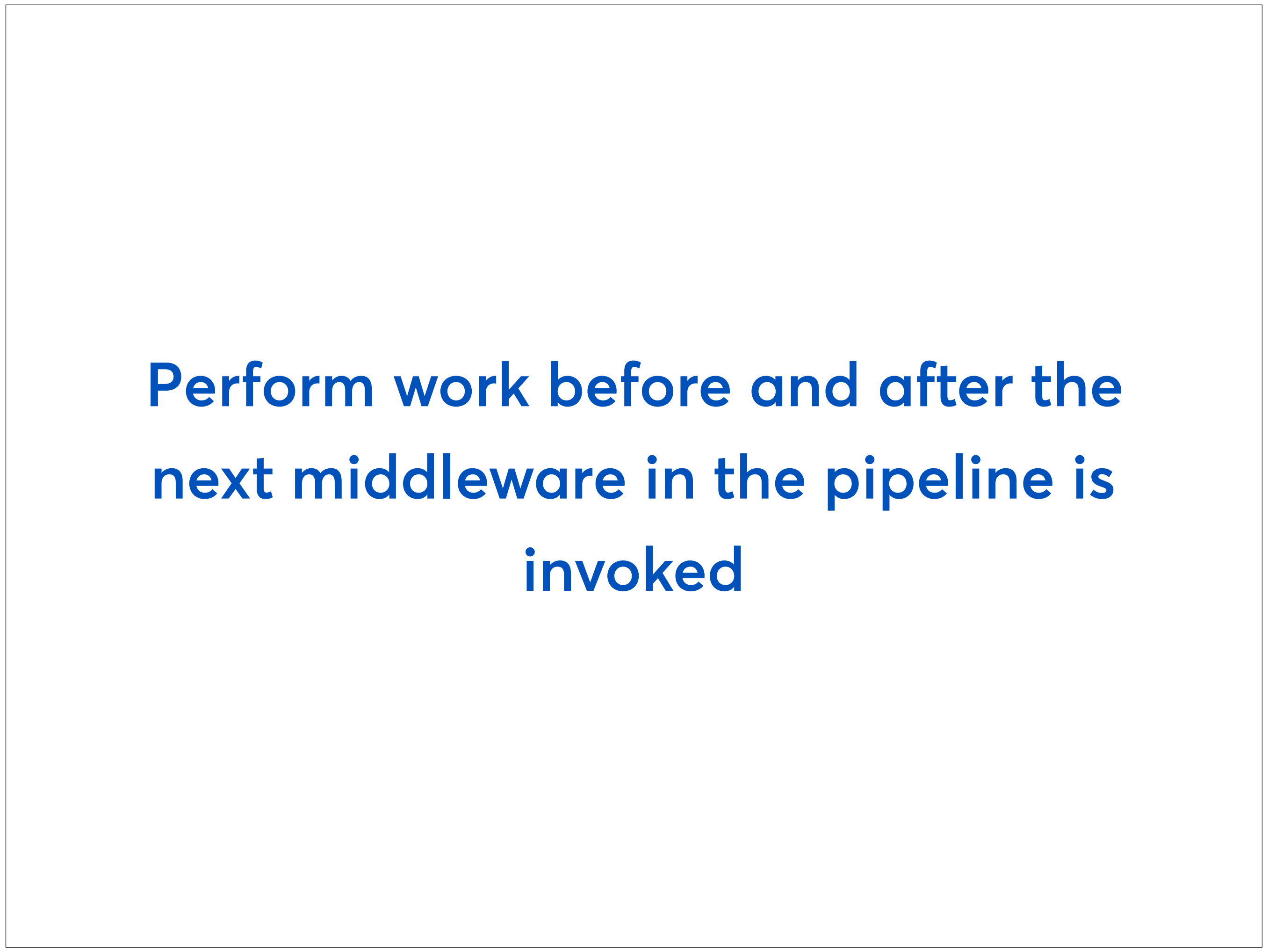
Each middleware in the pipeline can:


Choose whether to pass the request to the next middleware in the pipeline

Choose whether to pass the request to the next middleware in the pipeline

A middleware can do two things: it can either return a result, or it can call on the next middleware in the pipeline to do its job.

So, for example, a middleware which checks a user's access token could return early if the access token isn't valid, or if it is, it can move on to the next middleware in the pipeline.



Perform work before and after the next middleware in the pipeline is invoked

Perform work before and after the next middleware in the pipeline is invoked

For example, a middleware can fetch data and pass it to the next middleware in the pipeline, or it can run the next middleware in the pipeline, and then write something to our logs afterwards.

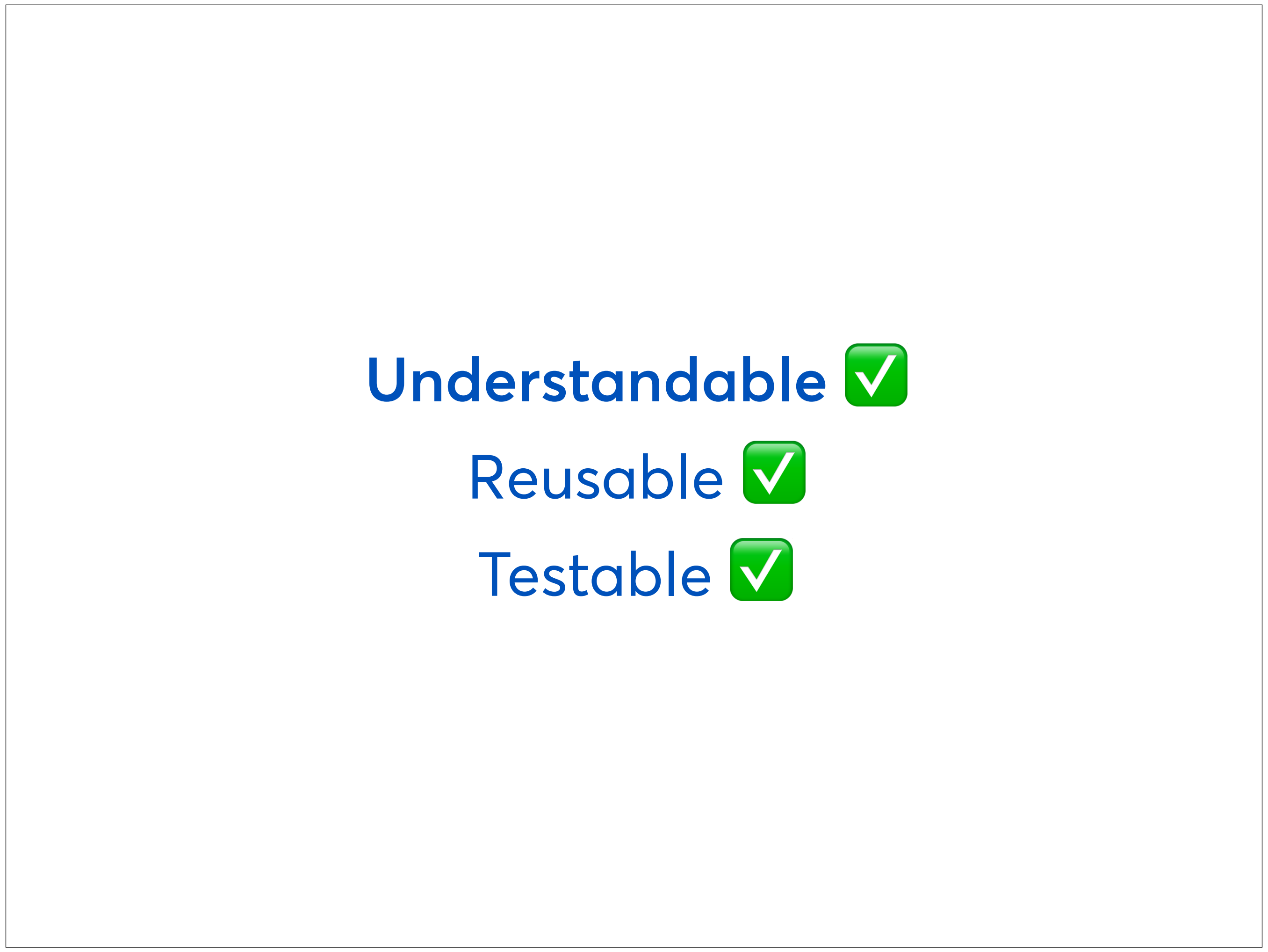
The middleware pattern has been particularly popularised by JavaScript frameworks like Express, as well as in a variety of other frameworks, for example Elixir’s Phoenix.

What’s so great about the middleware pattern?

So, what’s so great about this pattern? Why does building our application as a pipeline of middleware help us to make our code more maintainable?

The single responsibility principle

At the heart of this is the single responsibility principle: instead of big, unwieldy pieces of code doing many things, we have simple classes with a single job.



Understandable ✅

Reusable ✅

Testable ✅

Our code is easy to understand, because each middleware does one simple thing.

We then assemble lots of simple middleware together into a pipeline, and they work together to produce the final result, which ends up being pretty complex.

Understandable ✅

Reusable ✅

Testable ✅

This code is easy to reuse, because middleware have one job, and we can easily compose them together to build more complex functionality. We can even make them configurable.

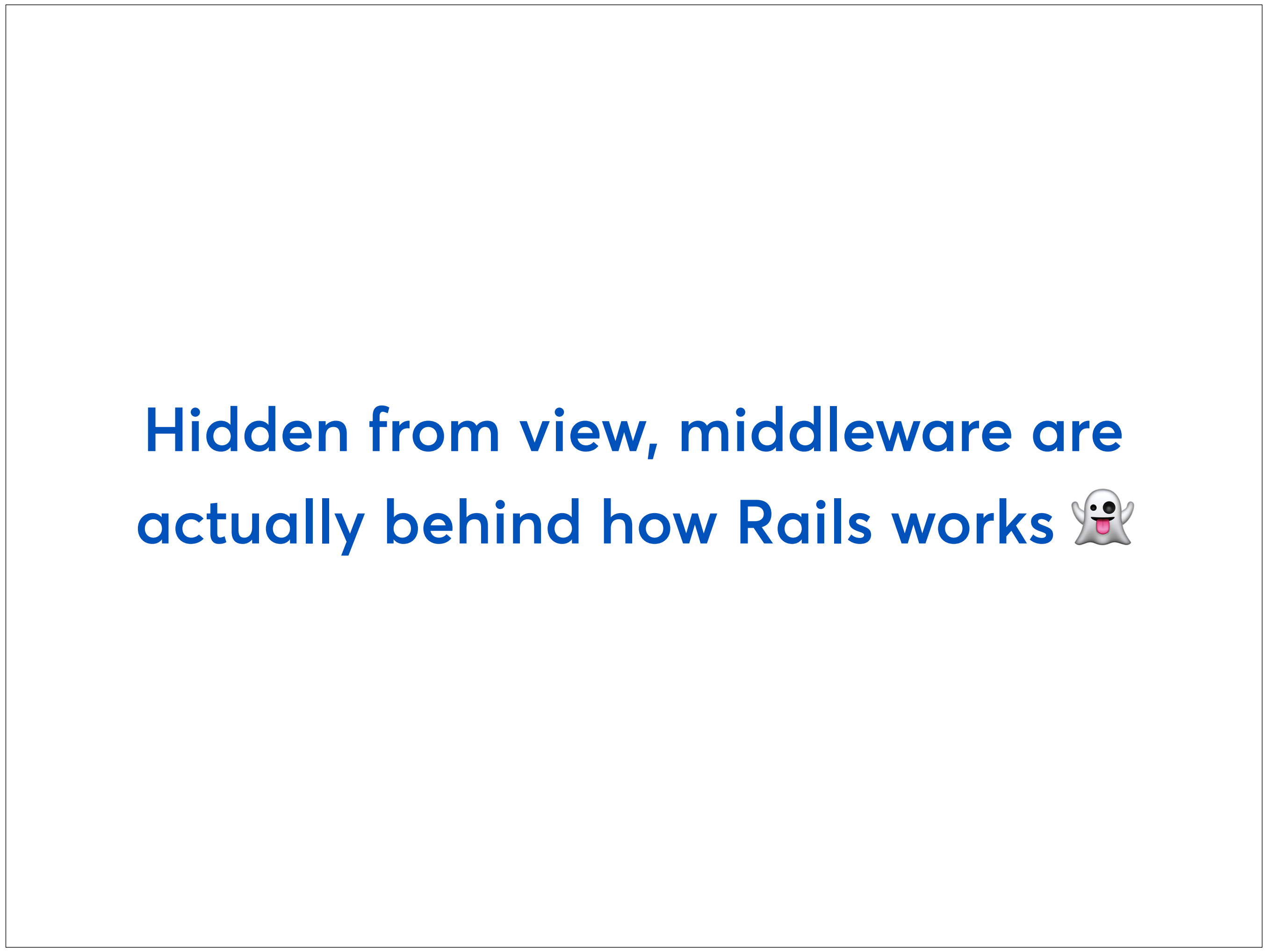
Understandable ✅

Reusable ✅

Testable ✅

Our middleware are easy to test. You can call the middleware, passing in some input, and see what happens.

You can see whether a response is returned - and if so, what - or if the next middleware in the chain is called.



Hidden from view, middleware are actually behind how Rails works 👻

Middleware are actually, unknown to many of us, at the core of how Rails works.

rack
powers web applications

Rails is built on Rack, a Ruby framework for building simple, modular web applications.

Before the code you’ve written gets run, and even before Rails gets run, a bunch of middleware get things ready.

```ruby
require "securerandom"

module ActionDispatch
  class RequestId
    X_REQUEST_ID = "X-Request-Id".freeze

    def call(env)
      req = ActionDispatch::Request.new env
      req.request_id = internal_request_id
      @app.call(env).tap do |_status, headers, _body|
        headers[X_REQUEST_ID] = req.request_id
      end
    end

    private

    def internal_request_id
      SecureRandom.uuid
    end
  end
end
```

For example, there's a middleware included in Rails called `ActionDispatch::RequestId`, which by default runs on every request.

You might have spotted that responses from your Rails application include an X-Request-Id header, making it easy to track requests in your log. Well, that's powered by this middleware.

On the screen, you can see a simplified version.

```
require "securerandom"

module ActionDispatch
  class RequestId
    X_REQUEST_ID = "X-Request-Id".freeze

    def call(env)
      req = ActionDispatch::Request.new env
      req.request_id = internal_request_id
      @app.call(env).tap do |_status, headers, _body|
        headers[X_REQUEST_ID] = req.request_id
      end
    end

    private

    def internal_request_id
      SecureRandom.uuid
    end
  end
end
```

It can take a look at the incoming request, represented by `env`.

```ruby
require "securerandom"

module ActionDispatch
  class RequestId
    X_REQUEST_ID = "X-Request-Id".freeze

    def call(env)
      req = ActionDispatch::Request.new env
      req.request_id = internal_request_id
      @app.call(env).tap do |_status, headers, _body|
        headers[X_REQUEST_ID] = req.request_id
      end
    end

    private

    def internal_request_id
      SecureRandom.uuid
    end
  end
end
```

It then generates a request ID and stores it in the request's state, so it can be used by later middleware in the pipeline - and by your application code!

```ruby
require "securerandom"

module ActionDispatch
  class RequestId
    X_REQUEST_ID = "X-Request-Id".freeze

    def call(env)
      req = ActionDispatch::Request.new env
      req.request_id = internal_request_id
      @app.call(env).tap do |_status, headers, _body|
        headers[X_REQUEST_ID] = req.request_id
      end
    end

    private

    def internal_request_id
      SecureRandom.uuid
    end
  end
end
```

Then it calls the next middleware in the pipeline

```
require "securerandom"

module ActionDispatch
  class RequestId
    X_REQUEST_ID = "X-Request-Id".freeze

    def call(env)
      req = ActionDispatch::Request.new env
      req.request_id = internal_request_id
      @app.call(env).tap do |_status, headers, _body|
        headers[X_REQUEST_ID] = req.request_id
      end
    end

    private

    def internal_request_id
      SecureRandom.uuid
    end
  end
end
```

And, at the end, it adds the request ID it generated to the response headers.

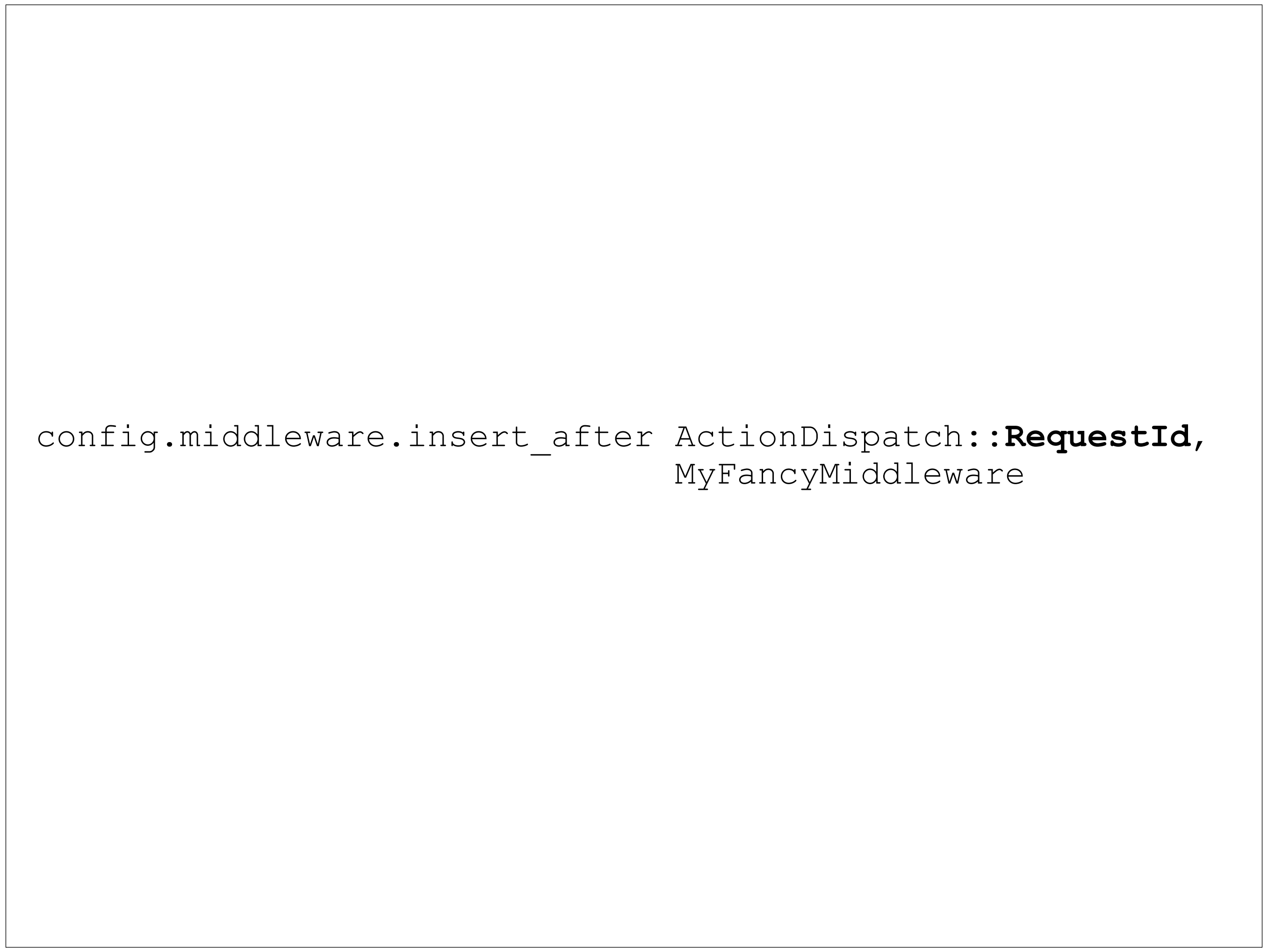
rack

powers web applications

Rack middleware do roughly what we want

Rails doesn't make it easy for us to plug middleware in on a per-request basis :(

But Rails doesn’t make it easy for us to plug middleware in on a per-request basis, so they’re a bit of a non-starter for building an API :(



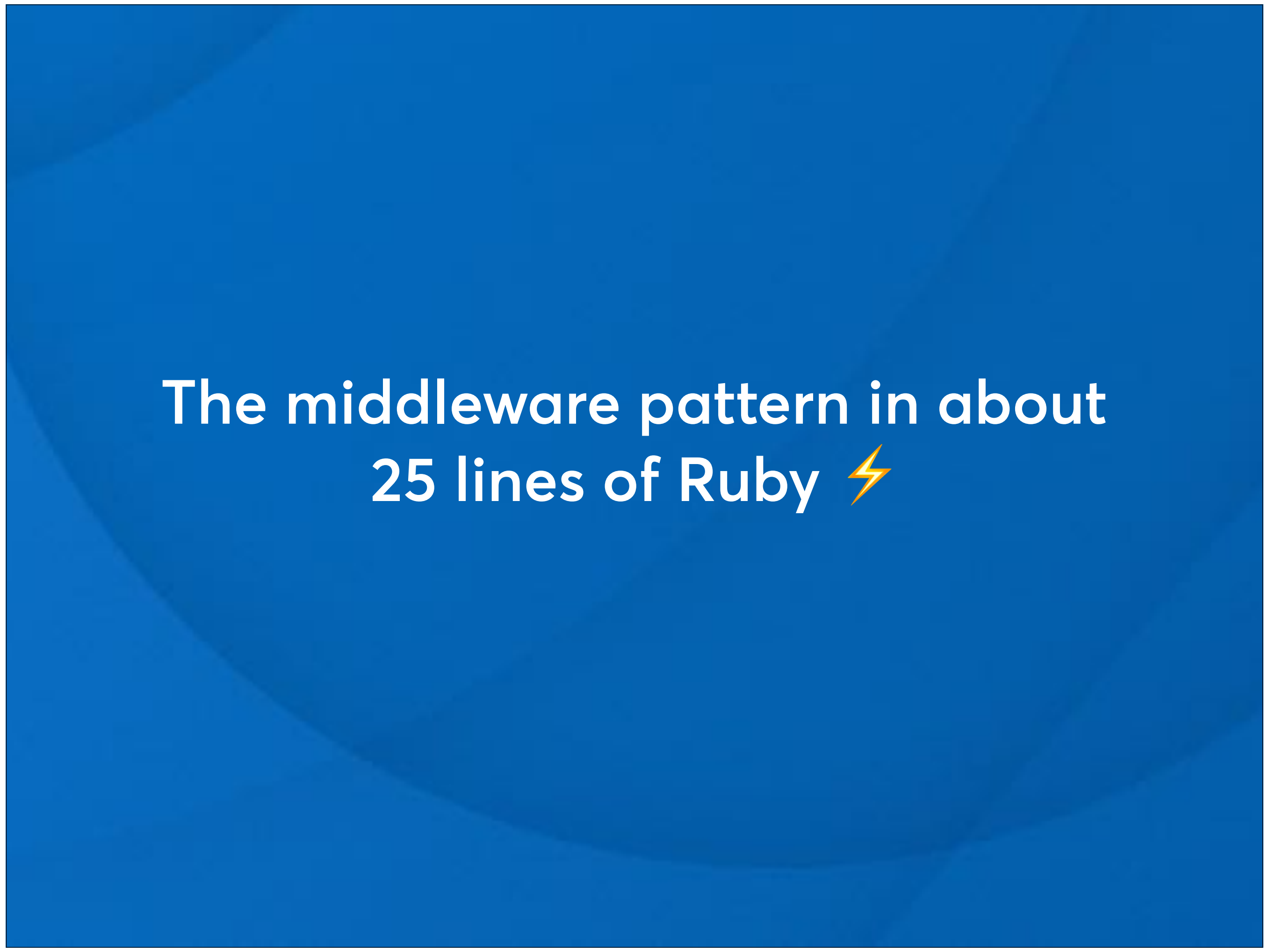
```
config.middleware.insert_after ActionDispatch::RequestId,
                               MyFancyMiddleware
```

You just have to set them up in `config/application.rb`.

We really want to be able to have the middleware pattern in our controllers to give us more flexibility.

Let's see how we can refactor our API using middleware

Let's see how we can refactor our API using middleware, making it miles more maintainable.

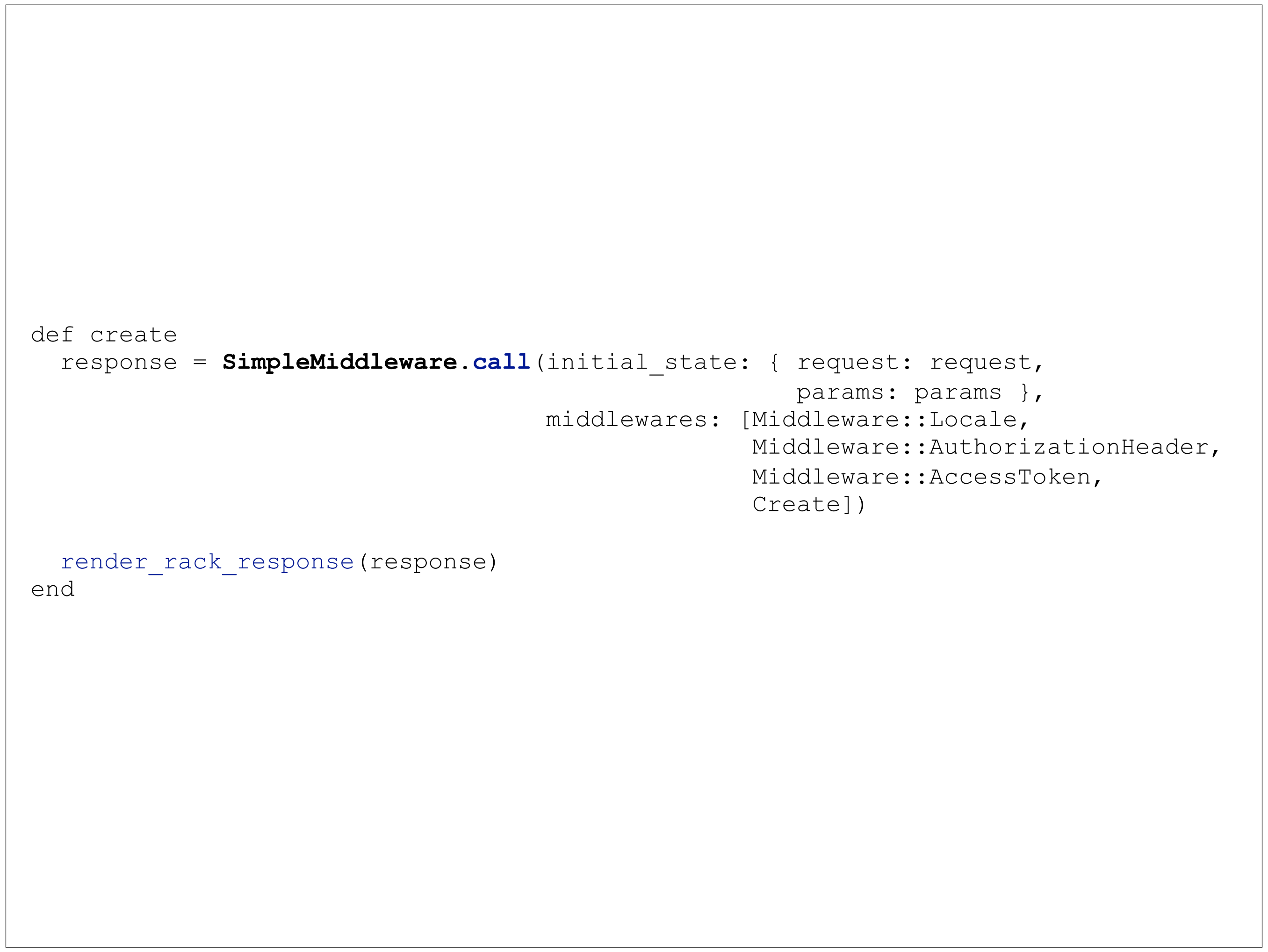


The middleware pattern in about 25 lines of Ruby ⚡

For this talk, I've built a simple implementation of the middleware pattern in about 25 lines of code.

https://github.com/timrogers/simple_middleware

I call it Simple Middleware. Let’s go through it together, and then reimplement our API using it.

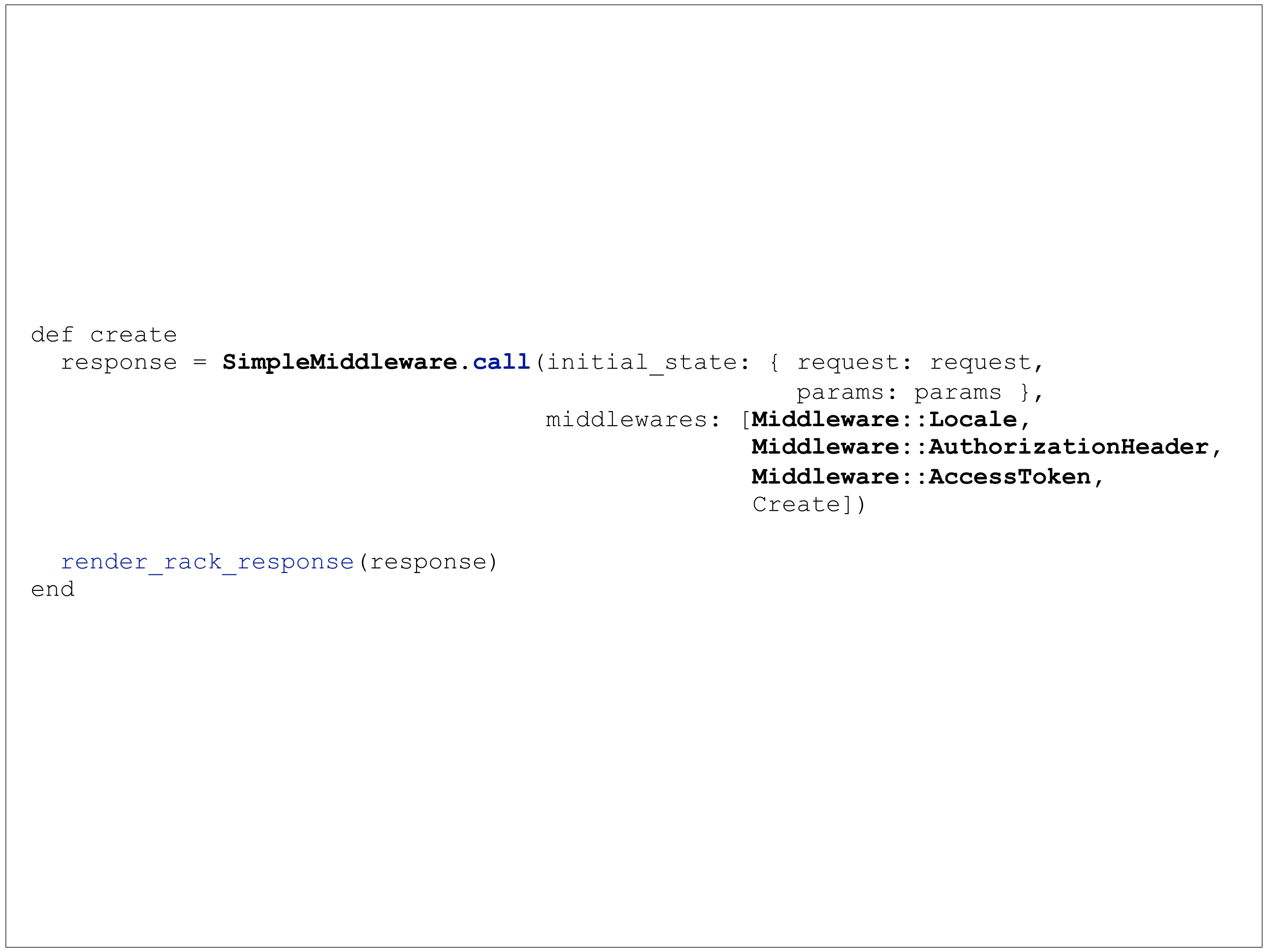


```ruby
def create
  response = SimpleMiddleware.call(initial_state: { request: request,
                                                    params: params },
                                   middlewares: [Middleware::Locale,
                                                 Middleware::AuthorizationHeader,
                                                 Middleware::AccessToken,
                                                 Create])

  render_rack_response(response)
end
```

But first, let’s look at the interface: how we’ll use our middleware from the controller.

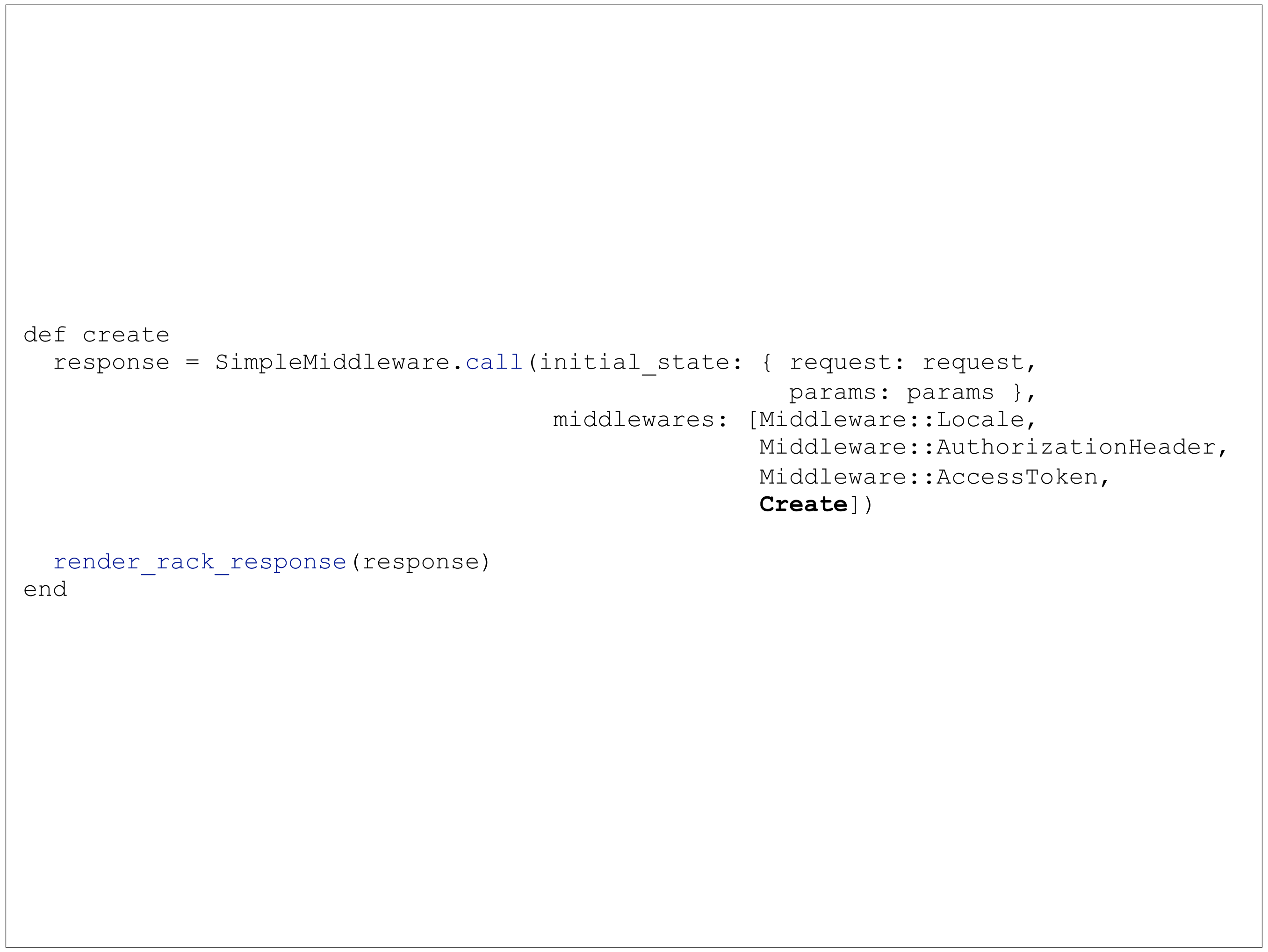
The `#create` method in the CustomersController ends up looking like this.

We call SimpleMiddleware with a list of middleware we want to run through, and specify the data we want to pass in - here, Rails’s request and params objects. We pass those objects in because we want to be able to use them from our middleware, for example to look at the Authorization header.



```
def create
  response = SimpleMiddleware.call(initial_state: { request: request,
                                                    params: params },
                                   middlewares: [Middleware::Locale,
                                                 Middleware::AuthorizationHeader,
                                                 Middleware::AccessToken,
                                                 Create])

  render_rack_response(response)
end
```

We turn out three filters from earlier into middleware.

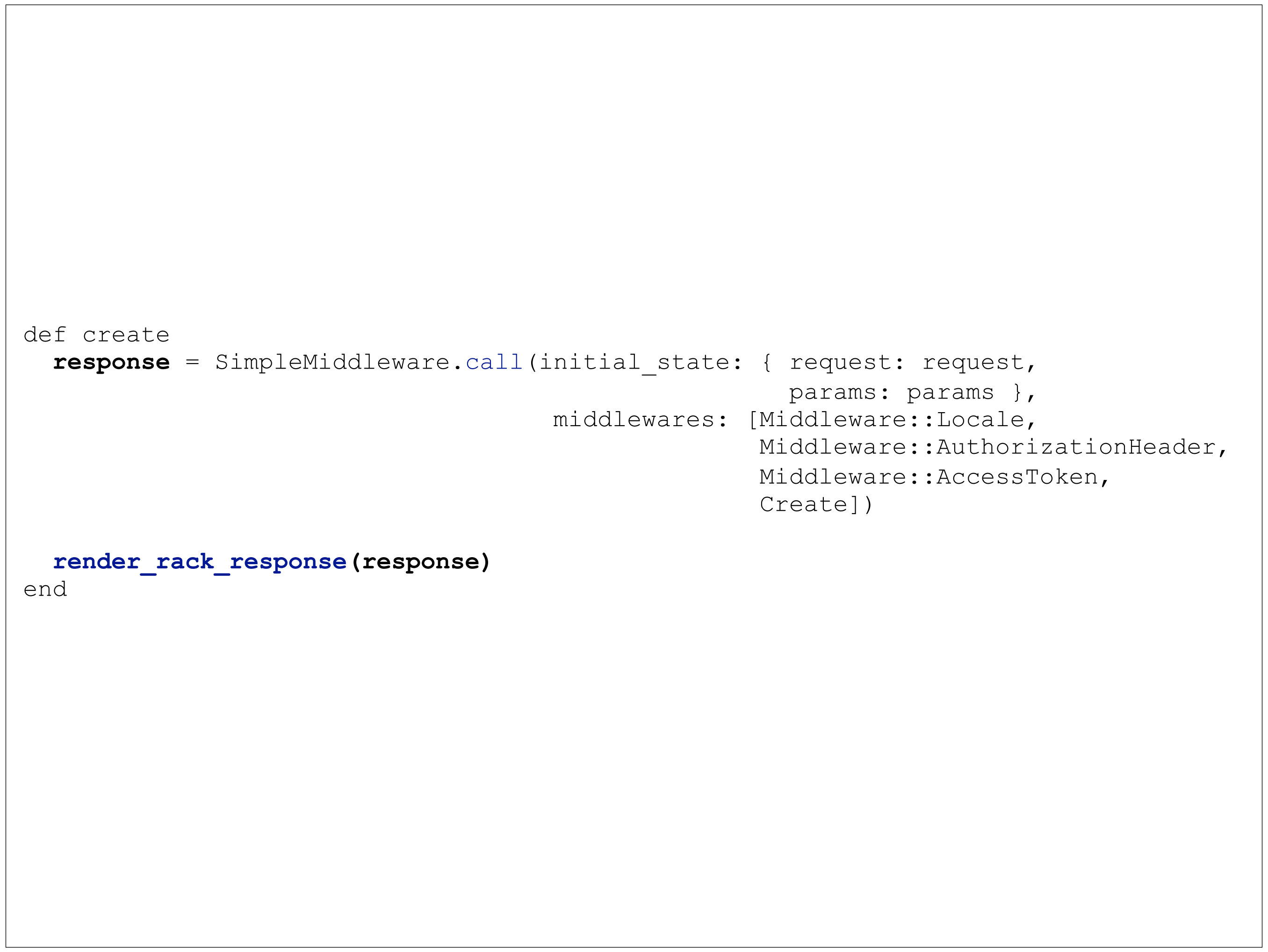


```ruby
def create
  response = SimpleMiddleware.call(initial_state: { request: request,
                                                    params: params },
                                    middlewares: [Middleware::Locale,
                                                  Middleware::AuthorizationHeader,
                                                  Middleware::AccessToken,
                                                  Create])

  render_rack_response(response)
end
```

We’ve also defined a final middleware called `Create`. In there, we’ll put the endpoint-specific logic that actually creates the customer.

We could do this a bit differently, and keep the endpoint-specific logic that creates the customer in the controller action itself.

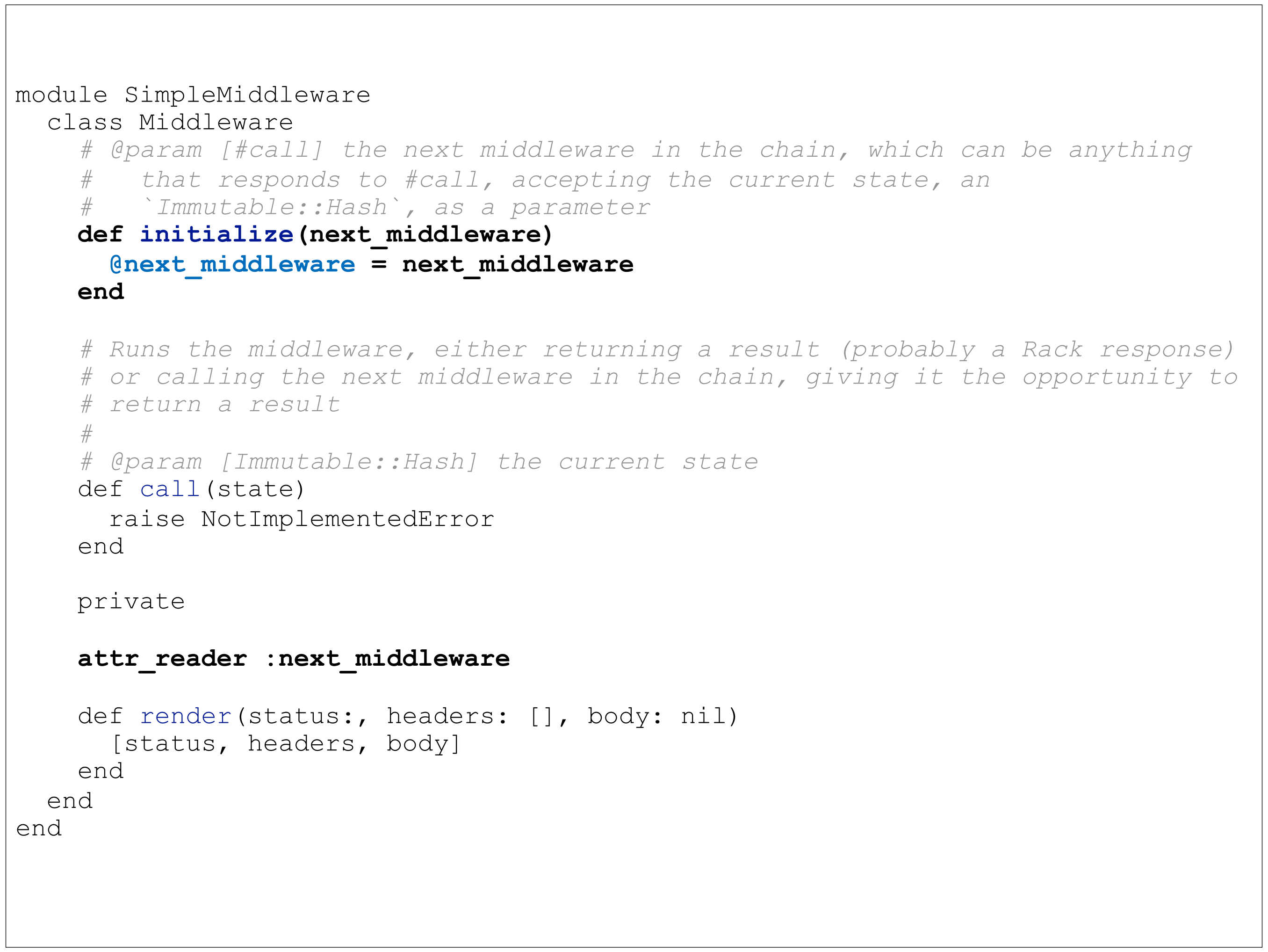
But I’ve chosen to write it as a middleware for consistency, and so we can unit test that part on its own.



```
def create
  response = SimpleMiddleware.call(initial_state: { request: request,
                                                    params: params },
                                   middlewares: [Middleware::Locale,
                                                 Middleware::AuthorizationHeader,
                                                 Middleware::AccessToken,
                                                 Create])

  render_rack_response(response)
end
```

At the end of the middleware chain, we expect to get back a Rack response, which is an array containing a status code, some response headers and a response body.

We'll have the same kind of response whether we got an error in an early middleware (e.g. due to a missing access token in the `AuthorizationHeader` middleware), or if the `Create` middleware successfully created a customer.



```
module SimpleMiddleware
  class Middleware
    # @param [#call] the next middleware in the chain, which can be anything
    #   that responds to #call, accepting the current state, an
    #   `Immutable::Hash`, as a parameter
    def initialize(next_middleware)
      @next_middleware = next_middleware
    end

    # Runs the middleware, either returning a result (probably a Rack response)
    # or calling the next middleware in the chain, giving it the opportunity to
    # return a result
    #
    # @param [Immutable::Hash] the current state
    def call(state)
      raise NotImplementedError
    end

    private

    attr_reader :next_middleware

    def render(status:, headers: [], body: nil)
      [status, headers, body]
    end
  end
end
```

So, we’ve seen what the interface looks like - that is, how we use Simple Middleware.

Now, let’s take a look at how the middleware pattern is actually implemented. The base middleware class itself is super simple.

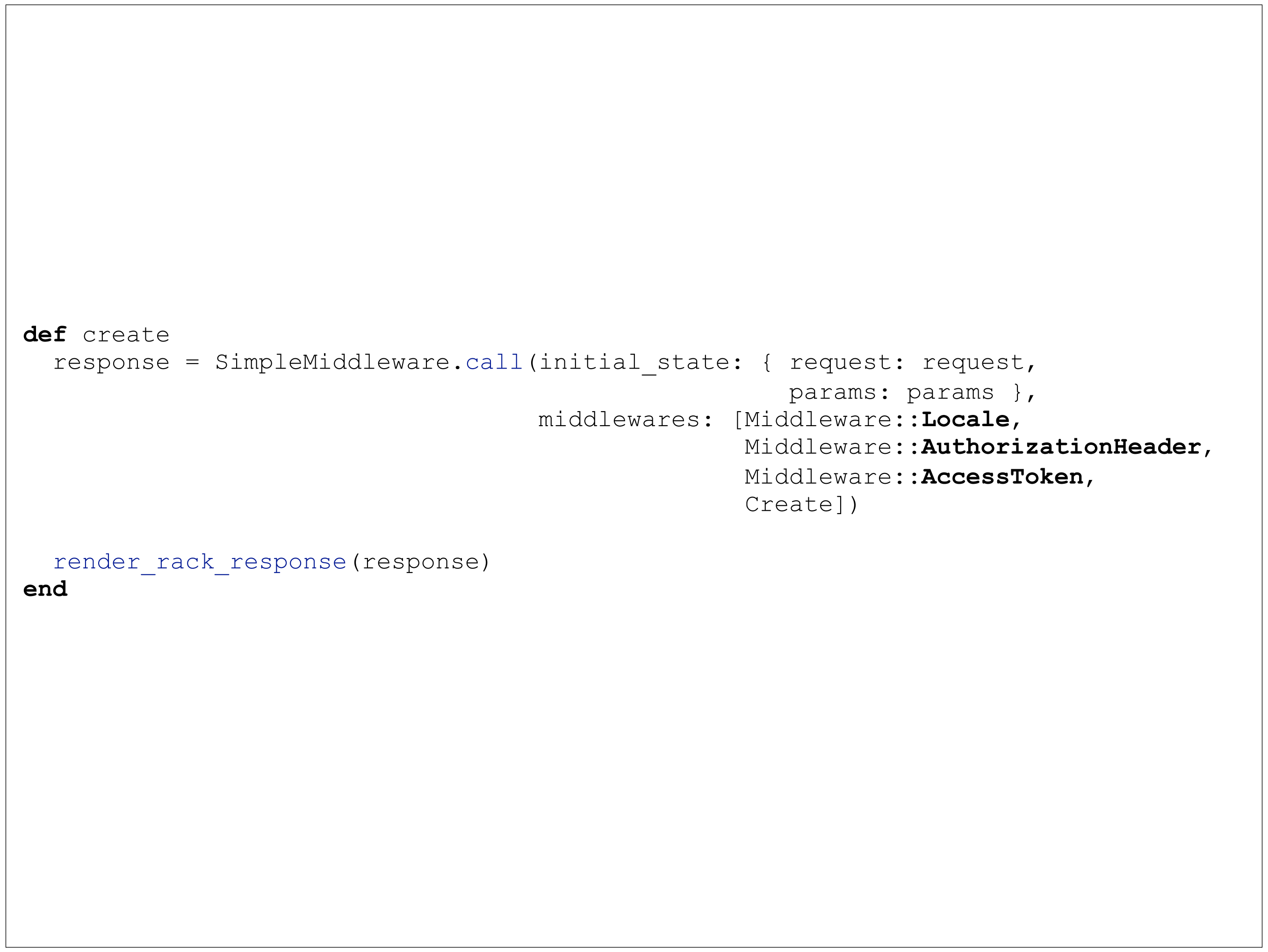
We’ve talked about how a middleware can do two things - return a response, or call the next middleware.

A middleware gets initialised with the next middleware in the chain after it, which we make accessible with an attribute reader.

```ruby
module SimpleMiddleware
  class Middleware
    # @param [#call] the next middleware in the chain, which can be anything
    #   that responds to #call, accepting the current state, an
    #   `Immutable::Hash`, as a parameter
    def initialize(next_middleware)
      @next_middleware = next_middleware
    end

    # Runs the middleware, either returning a result (probably a Rack response)
    # or calling the next middleware in the chain, giving it the opportunity to
    # return a result
    #
    # @param [Immutable::Hash] the current state
    def call(state)
      raise NotImplementedError
    end

    private

    attr_reader :next_middleware

    def render(status:, headers: [], body: nil)
      [status, headers, body]
    end
  end
end
```

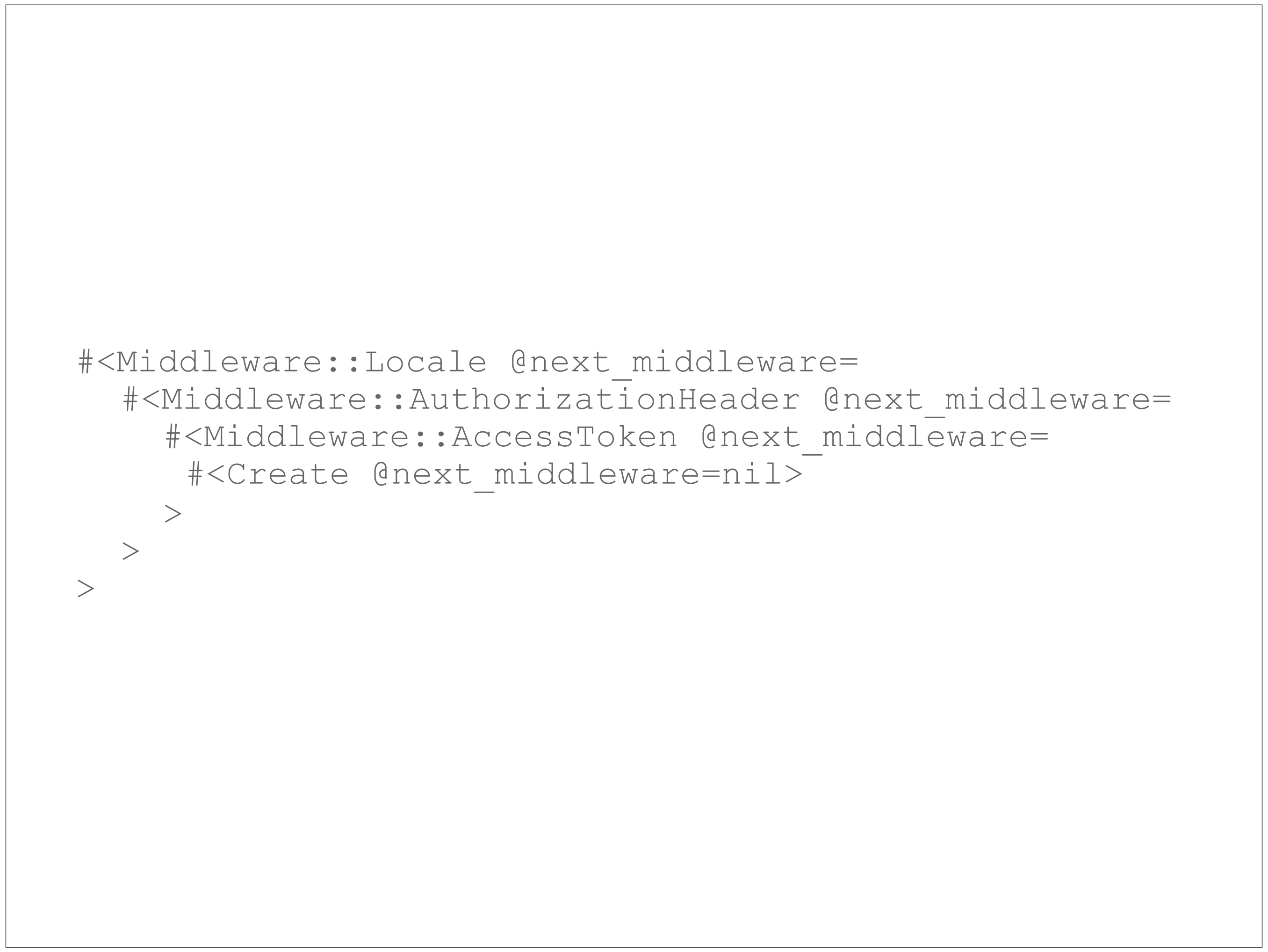
Its up to each middleware class to define a `#call` method.

That method can use the state, and then it has two options. It can either return a response, or call on the next middleware, passing it the state, which it can add something to along the way.



```ruby
def create
  response = SimpleMiddleware.call(initial_state: { request: request,
                                                    params: params },
                                   middlewares: [Middleware::Locale,
                                                 Middleware::AuthorizationHeader,
                                                 Middleware::AccessToken,
                                                 Create])

  render_rack_response(response)
end
```

Thinking back to our example, we put together a pipeline of four middleware. Each middleware can perform a single job, and they can work together to handle the request.

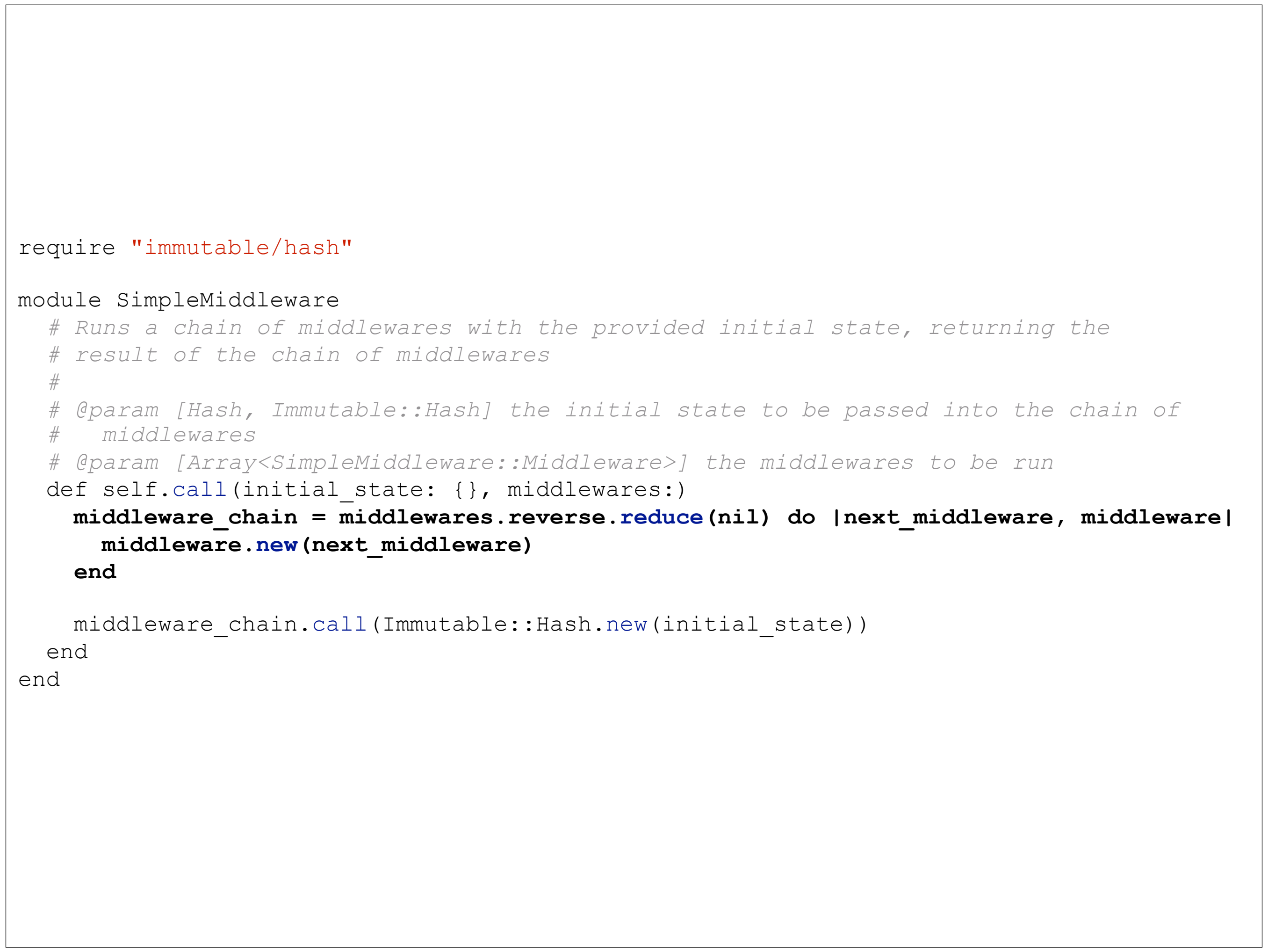


```
#<Middleware::Locale @next_middleware=
  #<Middleware::AuthorizationHeader @next_middleware=
    #<Middleware::AccessToken @next_middleware=
     #<Create @next_middleware=nil>
    >
  >
>
```

Our pipeline gets run through in the order we specify. This is important, as some middleware can depend on others. For example, our access token middleware will depend on the access token provided by the authorization header middleware.

We want the code that handles the locale to run first, then we want to parse the Authorization header, then we want to look up the access token, and then finally we want to run the endpoint-specific logic.

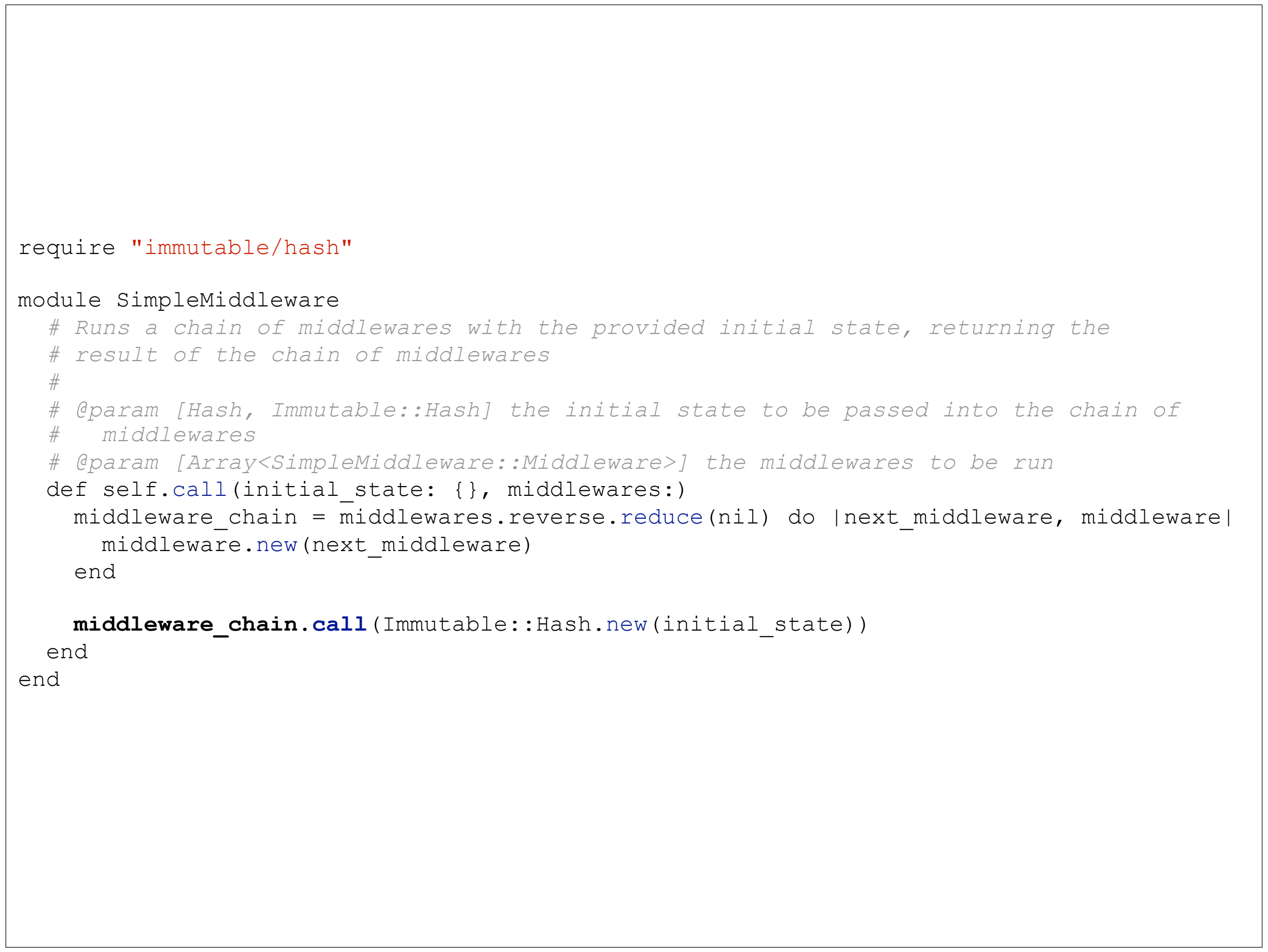
Now we're thinking in terms of middleware, that means that the locale middleware is the outer middleware. Each middleware in the chain knows which middleware is next. It does its work, generates the new state, and then calls the next middleware with it.



```ruby
require "immutable/hash"

module SimpleMiddleware
  # Runs a chain of middlewares with the provided initial state, returning the
  # result of the chain of middlewares
  #
  # @param [Hash, Immutable::Hash] the initial state to be passed into the chain of
  #   middlewares
  # @param [Array<SimpleMiddleware::Middleware>] the middlewares to be run
  def self.call(initial_state: {}, middlewares:)
    middleware_chain = middlewares.reverse.reduce(nil) do |next_middleware, middleware|
      middleware.new(next_middleware)
    end

    middleware_chain.call(Immutable::Hash.new(initial_state))
  end
end
```

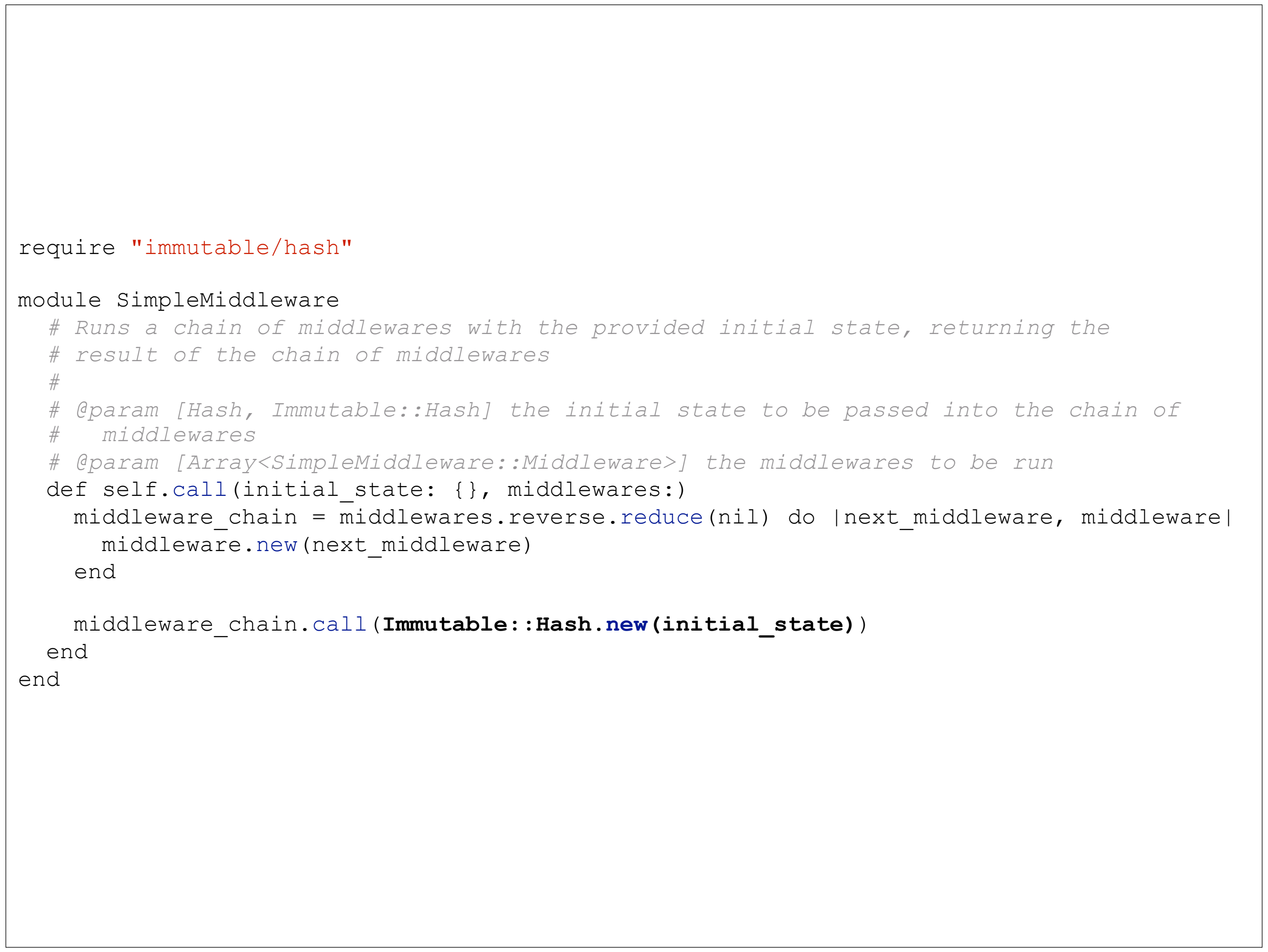
The code to put together our pipeline of middleware is simple.

It takes the list of middleware passed in, and then stacks them into a pipeline, where each middleware knows the next one in the pipeline.



```ruby
require "immutable/hash"

module SimpleMiddleware
  # Runs a chain of middlewares with the provided initial state, returning the
  # result of the chain of middlewares
  #
  # @param [Hash, Immutable::Hash] the initial state to be passed into the chain of
  #   middlewares
  # @param [Array<SimpleMiddleware::Middleware>] the middlewares to be run
  def self.call(initial_state: {}, middlewares:)
    middleware_chain = middlewares.reverse.reduce(nil) do |next_middleware, middleware|
      middleware.new(next_middleware)
    end

    middleware_chain.call(Immutable::Hash.new(initial_state))
  end
end
```

Once we’ve built our chain, we can just call the #call method on the first one, passing in our initial state,

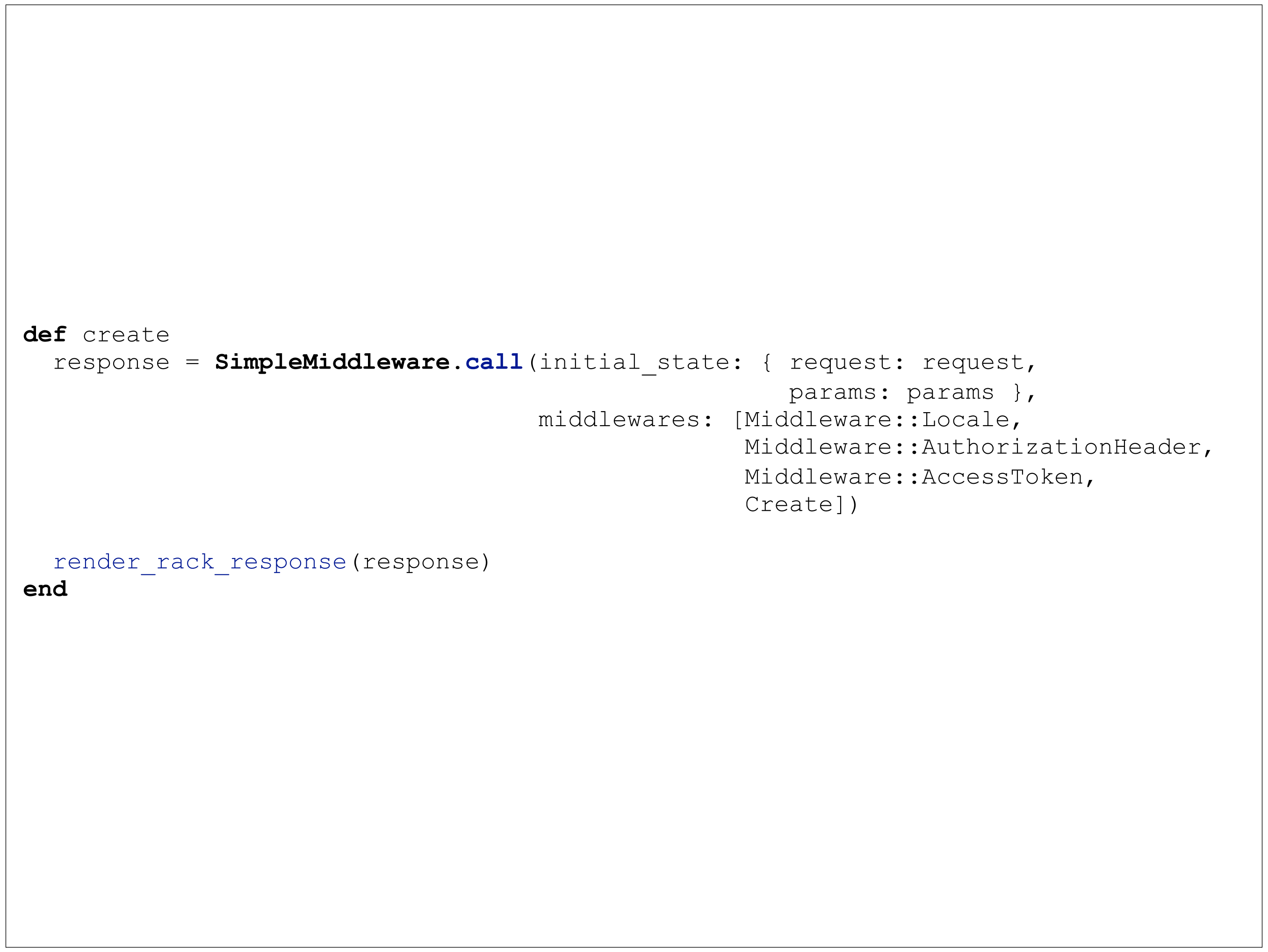


```ruby
require "immutable/hash"

module SimpleMiddleware
  # Runs a chain of middlewares with the provided initial state, returning the
  # result of the chain of middlewares
  #
  # @param [Hash, Immutable::Hash] the initial state to be passed into the chain of
  #   middlewares
  # @param [Array<SimpleMiddleware::Middleware>] the middlewares to be run
  def self.call(initial_state: {}, middlewares:)
    middleware_chain = middlewares.reverse.reduce(nil) do |next_middleware, middleware|
      middleware.new(next_middleware)
    end

    middleware_chain.call(Immutable::Hash.new(initial_state))
  end
end
```

Then that state can cascade down through the pipeline.

For the state, we’re using an “immutable hash”, provided by the gem “immutable-ruby”.

Immutable hashes can’t be mutated; they can’t be changed. All the operations you perform on them - for example, adding a new key, return a copy of the hash with the change you’ve made. Using an immutable data structures gives you more confidence in your code and helps to avoid bugs in the pipeline.



```ruby
def create
  response = SimpleMiddleware.call(initial_state: { request: request,
                                                    params: params },
                                   middlewares: [Middleware::Locale,
                                                 Middleware::AuthorizationHeader,
                                                 Middleware::AccessToken,
                                                 Create])

  render_rack_response(response)
end
```

Let’s just go through what happens when we call our pipeline of middleware.

```
#<Middleware::Locale @next_middleware=
  #<Middleware::AuthorizationHeader @next_middleware=
    #<Middleware::AccessToken @next_middleware=
     #<Create @next_middleware=nil>
    >
  >
>
```

First, the Locale middleware gets called. It sets the locale, and then calls the next middleware.

```
#<Middleware::Locale @next_middleware=
  #<Middleware::AuthorizationHeader @next_middleware=
    #<Middleware::AccessToken @next_middleware=
     #<Create @next_middleware=nil>
    >
  >
>
```

Now we're in the Authorization header middleware.

It might return an error, for example if there's no Authorization header, in which case our pipeline stops.

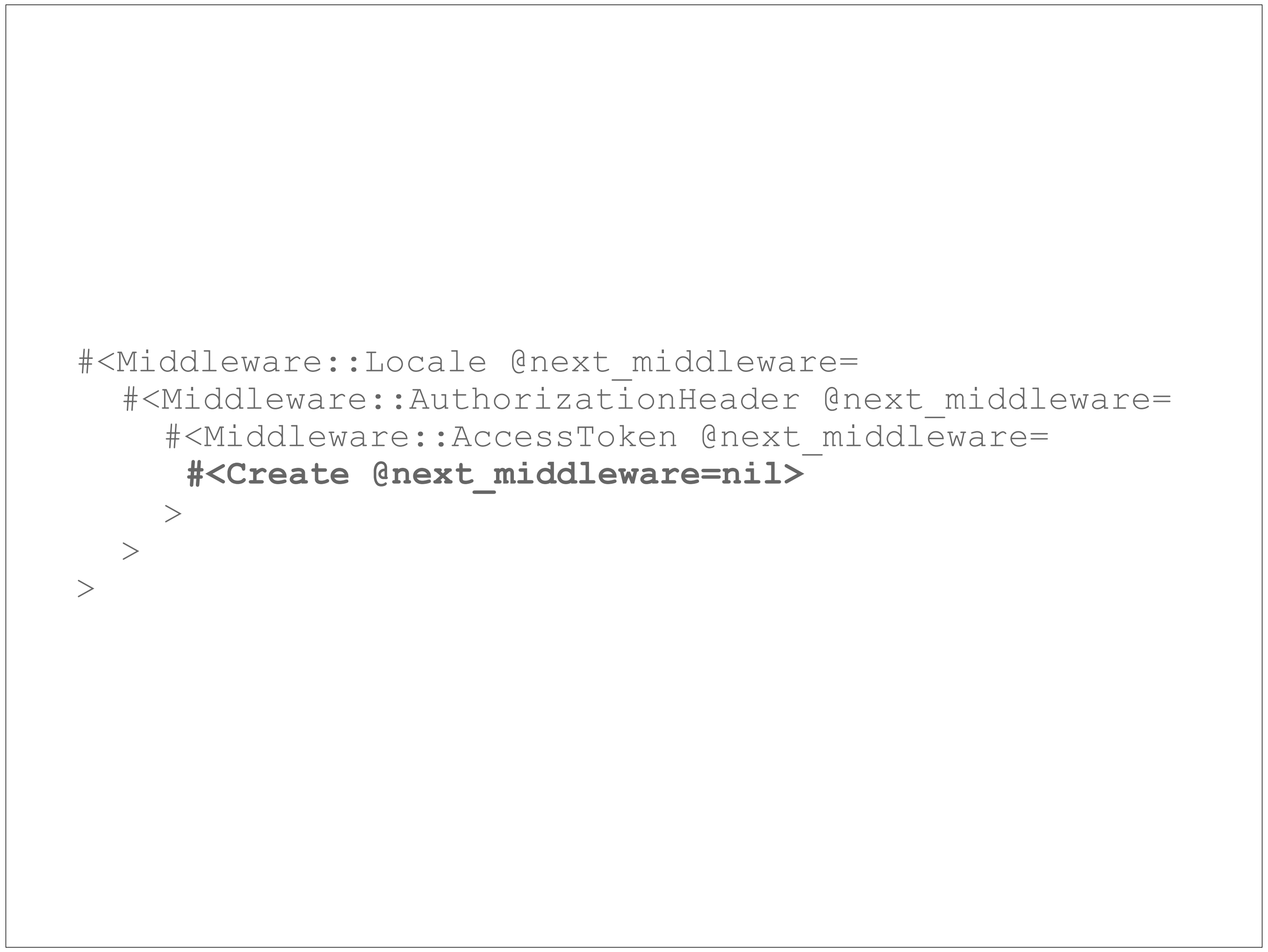
Otherwise, we'll call the next middleware, passing on the access token we've just pulled out of the header.

```
#<Middleware::Locale @next_middleware=
  #<Middleware::AuthorizationHeader @next_middleware=
    #<Middleware::AccessToken @next_middleware=
     #<Create @next_middleware=nil>
    >
  >
>
```

Now we’re in the access token middleware.

It might return an error if the access token doesn’t exist, in which case our pipeline stops.

Otherwise, we’ll call the next middleware, our endpoint-specific code, passing in the user we’ve just found.



```
#<Middleware::Locale @next_middleware=
  #<Middleware::AuthorizationHeader @next_middleware=
    #<Middleware::AccessToken @next_middleware=
     #<Create @next_middleware=nil>
    >
  >
>
```

Finally, we’ll try to create the customer attached to the authenticated user, using the request parameters. It’ll return our final HTTP response.

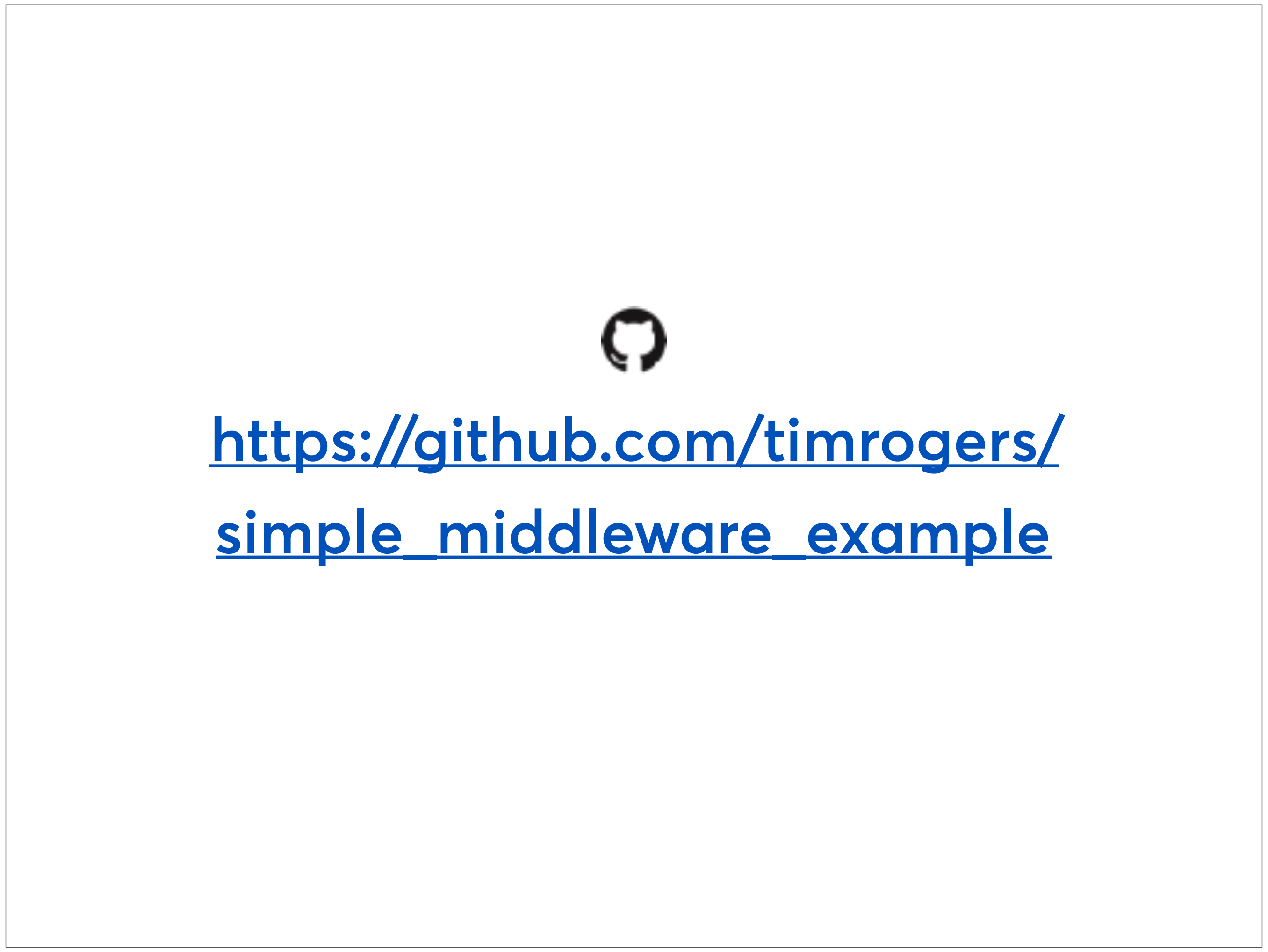
https://github.com/timrogers/simple_middleware

That’s how simple_middleware works - as you can see, it’s pretty simple and there’s very little code. You’ll find the full source on GitHub.

Time to refactor our API using middleware 🎉

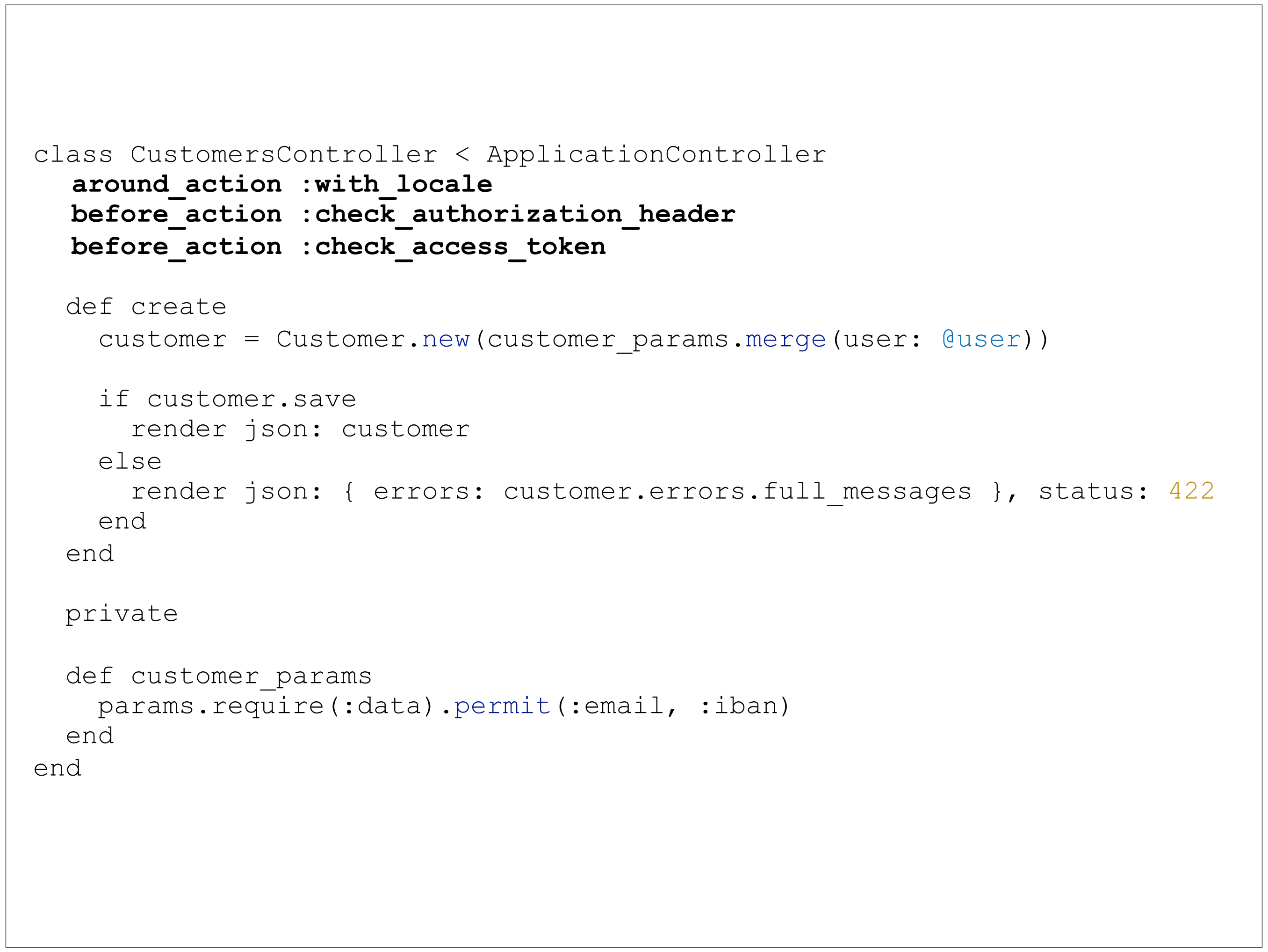
Now it's time for what we've all been waiting for - let's refactor our API using middleware.

We'll write our new middleware, plus unit tests.



https://github.com/timrogers/simple_middleware_example

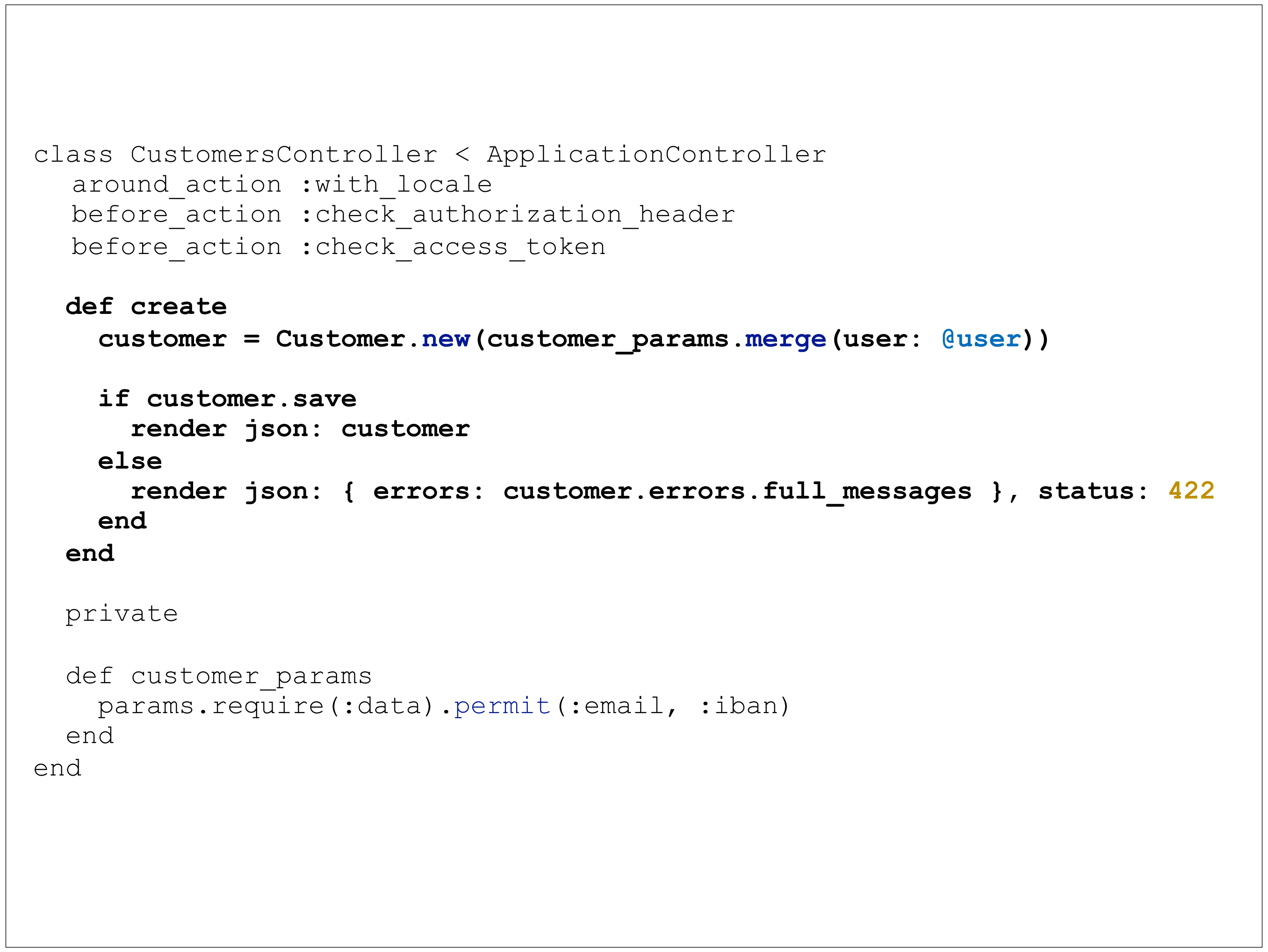
You'll be able to download the code for this at the end, so don't worry about following along too closely.



```ruby
class CustomersController < ApplicationController
  around_action :with_locale
  before_action :check_authorization_header
  before_action :check_access_token

  def create
    customer = Customer.new(customer_params.merge(user: @user))

    if customer.save
      render json: customer
    else
      render json: { errors: customer.errors.full_messages }, status: 422
    end
  end

  private

  def customer_params
    params.require(:data).permit(:email, :iban)
  end
end
```

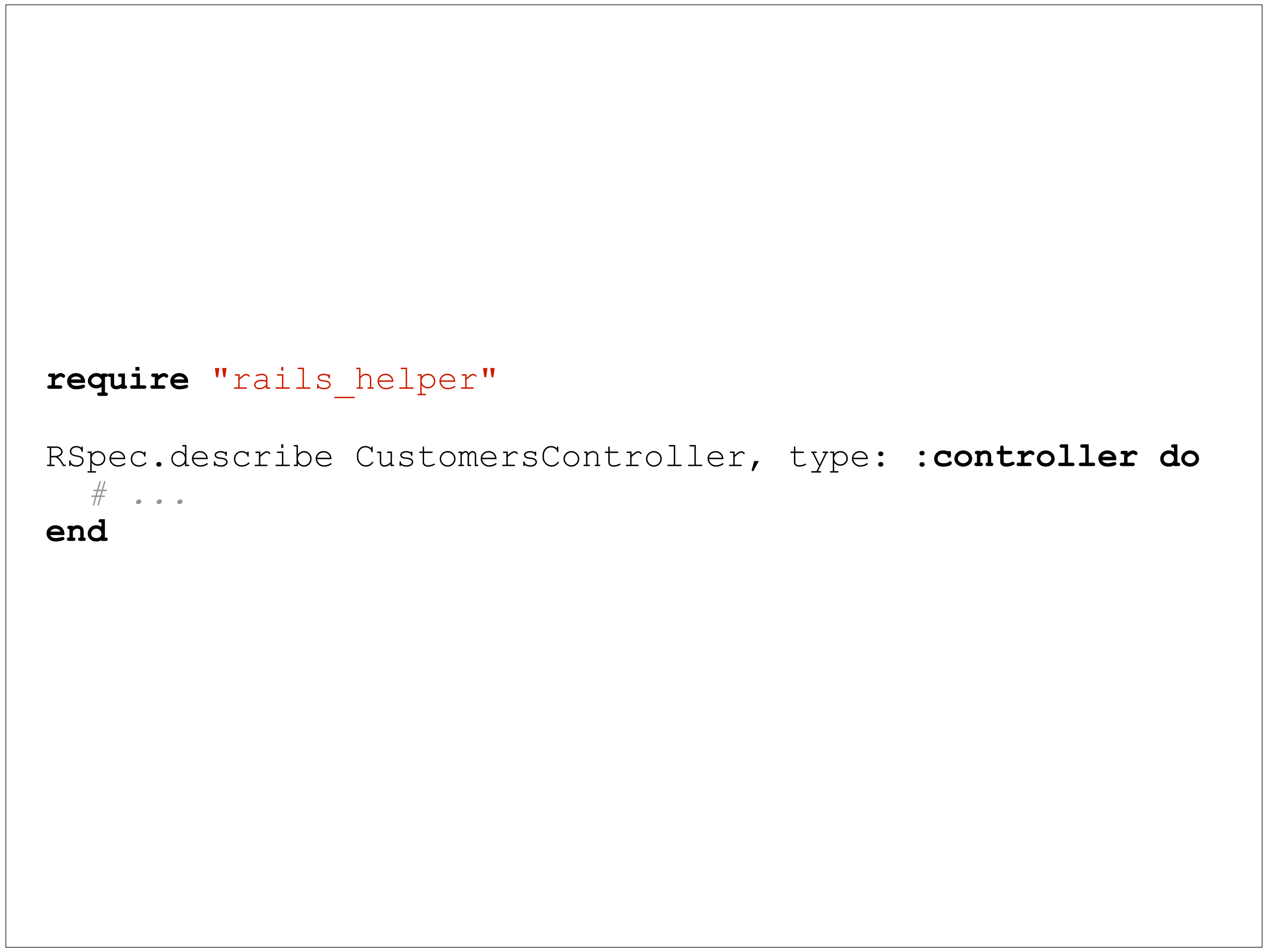
So, let's think back to our original controller from the beginning.

We want to refactor our three filters into middleware, using Simple Middleware.



```
class CustomersController < ApplicationController
   around_action :with_locale
   before_action :check_authorization_header
   before_action :check_access_token

  def create
    customer = Customer.new(customer_params.merge(user: @user))

    if customer.save
      render json: customer
    else
      render json: { errors: customer.errors.full_messages }, status: 422
    end
  end

  private

  def customer_params
    params.require(:data).permit(:email, :iban)
  end
end
```

We’ll then also rewrite the `create` method as a middleware too, to go at the end of the pipeline. We write this as a middleware rather than keeping it in the controller action for consistency.

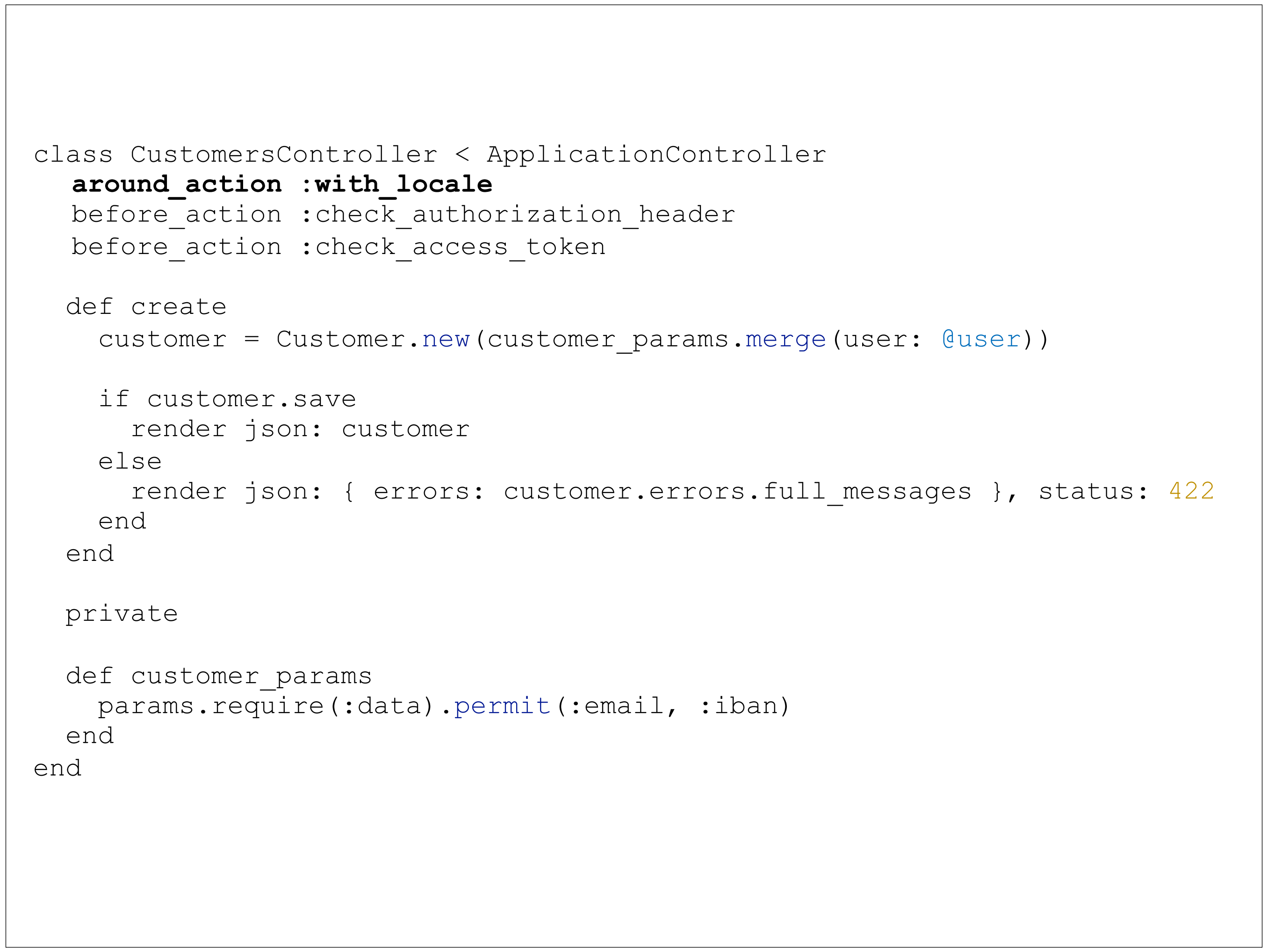


```ruby
require "rails_helper"

RSpec.describe CustomersController, type: :controller do
  # ...
end
```

Any specs we’ve already written for our controller can really help us out.

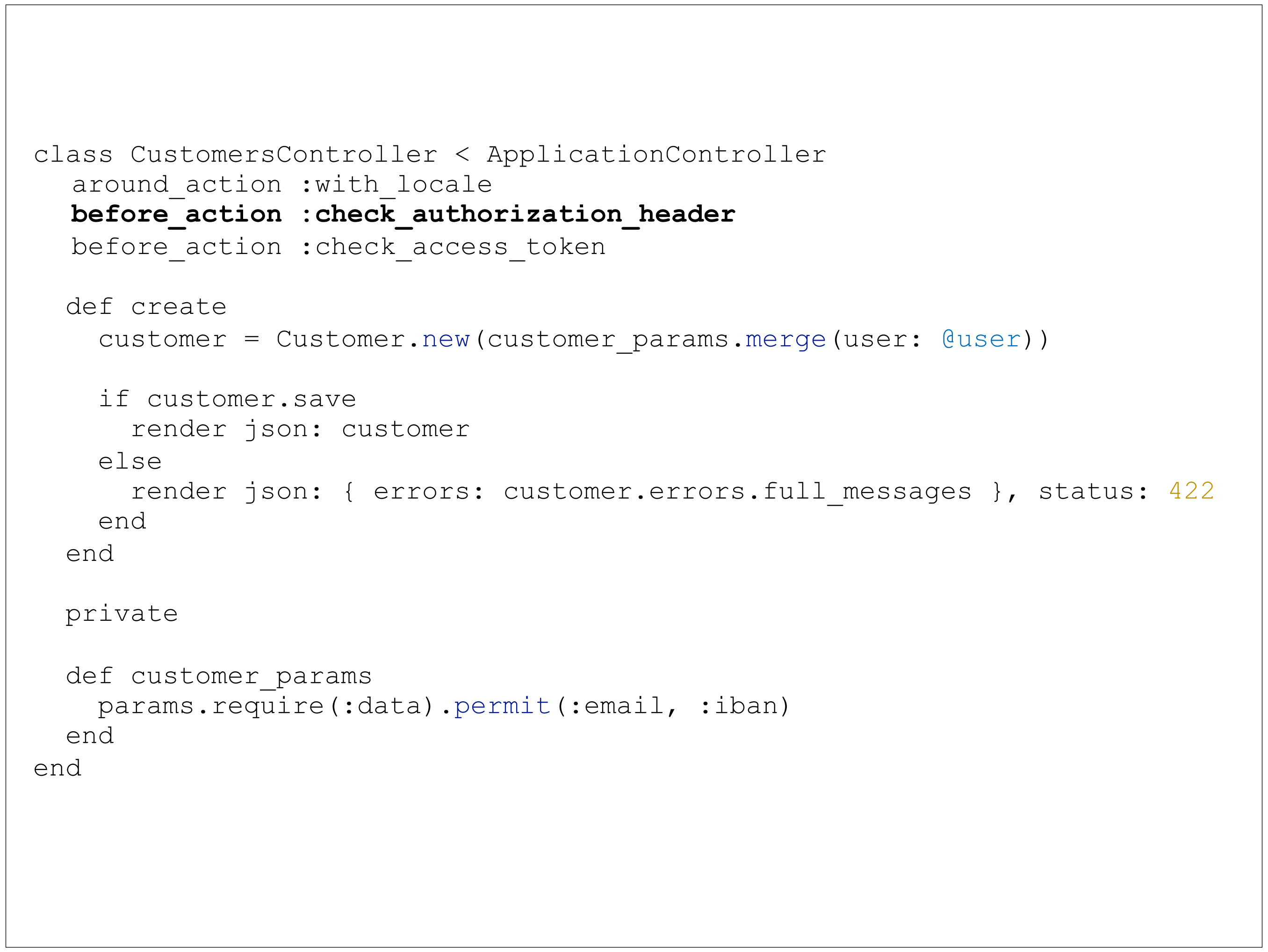
They can act as integration tests as we move our code to the middleware pattern, testing the edge cases and giving us confidence as we make changes.

So for example, I’ve already written some tests for the original controller, which you’ll find in the GitHub project.



```ruby
class CustomersController < ApplicationController
  around_action :with_locale
  before_action :check_authorization_header
  before_action :check_access_token

  def create
    customer = Customer.new(customer_params.merge(user: @user))

    if customer.save
      render json: customer
    else
      render json: { errors: customer.errors.full_messages }, status: 422
    end
  end

  private

  def customer_params
    params.require(:data).permit(:email, :iban)
  end
end
```

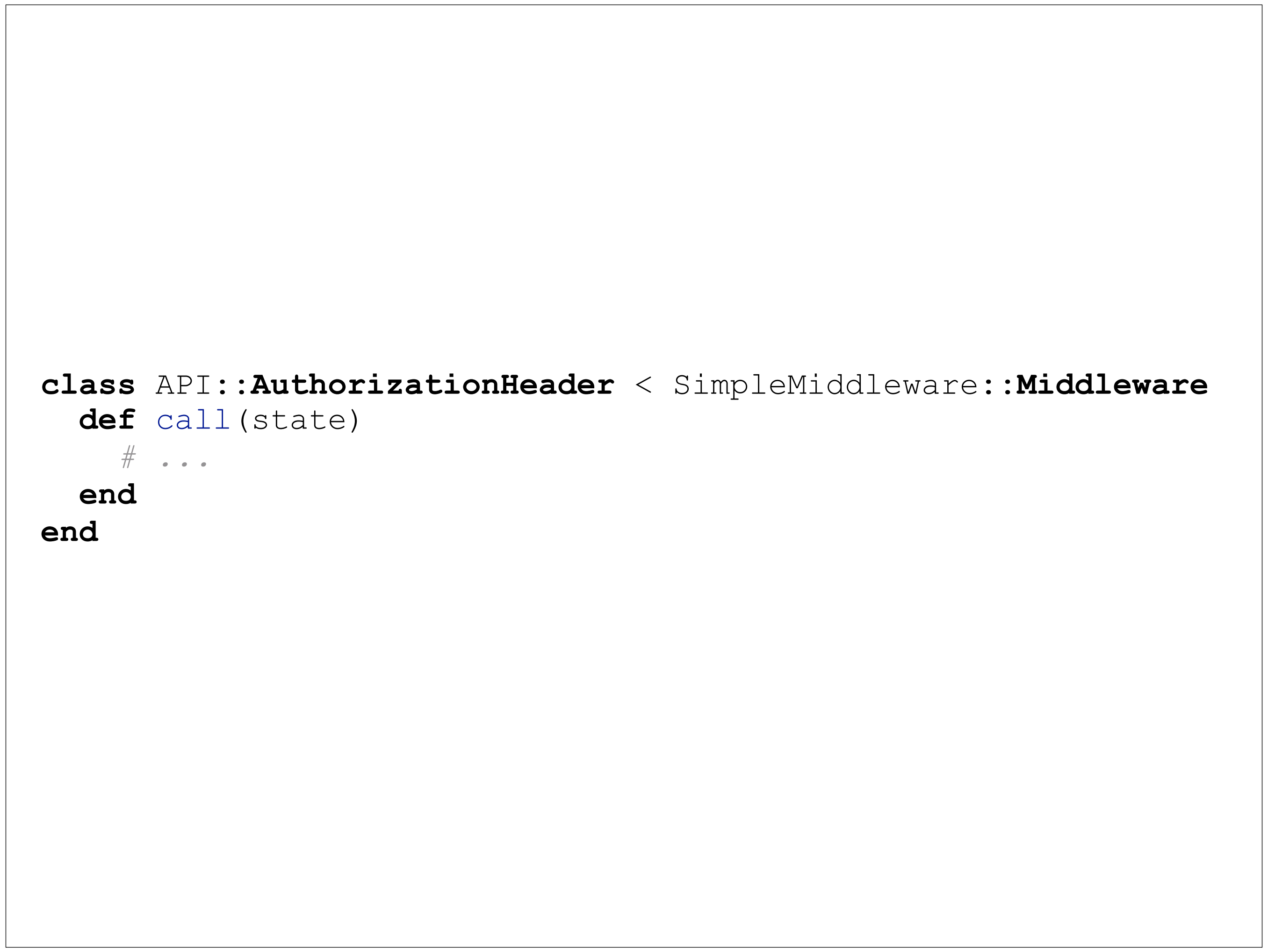
We’re going to skip the “with locale” filter, as it isn’t super interesting.



```ruby
class CustomersController < ApplicationController
  around_action :with_locale
  before_action :check_authorization_header
  before_action :check_access_token

  def create
    customer = Customer.new(customer_params.merge(user: @user))

    if customer.save
      render json: customer
    else
      render json: { errors: customer.errors.full_messages }, status: 422
    end
  end

  private

  def customer_params
    params.require(:data).permit(:email, :iban)
  end
end
```

Let’s start by refactoring the “check authorization header” filter.

We’ll turn that to a middleware and add unit tests.

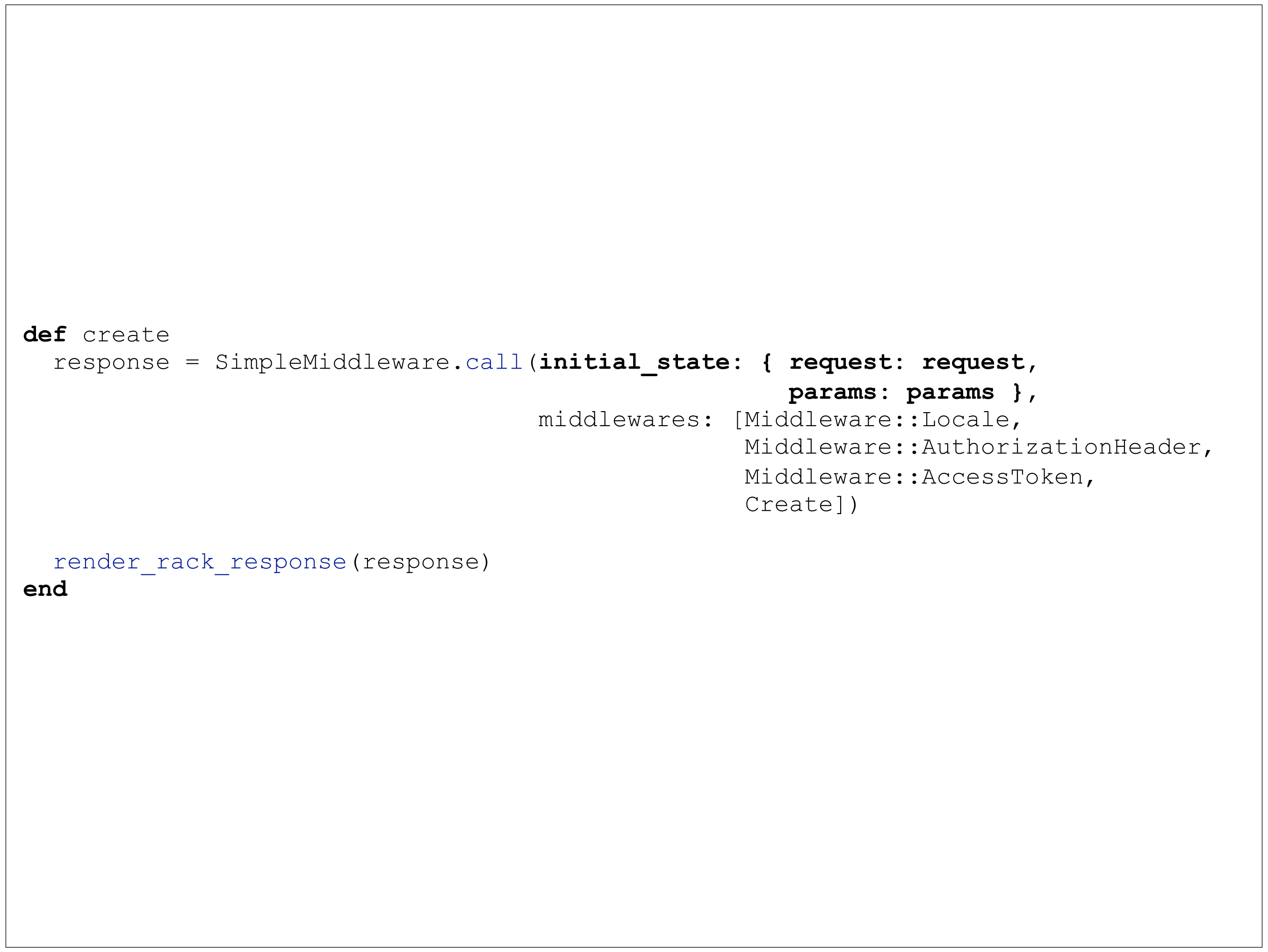


```ruby
class API::AuthorizationHeader < SimpleMiddleware::Middleware
  def call(state)
    # ...
  end
end
```

The fundamentals of a middleware are simple.

We have a class that inherits from `SimpleMiddleware::Middleware`, and defines a `#call` method which expects to be passed the current state.

The `#call` method can access the next middleware in the chain, and can either call it or return a response itself.



```
def create
  response = SimpleMiddleware.call(initial_state: { request: request,
                                                    params: params },
                                   middlewares: [Middleware::Locale,
                                                 Middleware::AuthorizationHeader,
                                                 Middleware::AccessToken,
                                                 Create])

  render_rack_response(response)
end
```

Thinking back to the controller, as part of the initial state, we pass in the Rails request object, as well as the request parameters

```
def call(state)
  header_value = state[:request].headers["HTTP_AUTHORIZATION"]

  # ...
end
```

That exposes the HTTP headers, so we can grab the value of the Authorization header from before.

```
def call(state)
  header_value = state[:request].headers["HTTP_AUTHORIZATION"]

  return missing_access_token_error unless header_value.present?
  token_type, token = header_value.split(" ", 2)
  return missing_access_token_error unless token_type == "Bearer"

  # ...
end

private

def missing_access_token_error
  render status: 401,
         headers: { "Content-Type" => "application/json" },
         body: JSON.generate(errors: [I18n.t("errors.missing_access_token")])
end
```

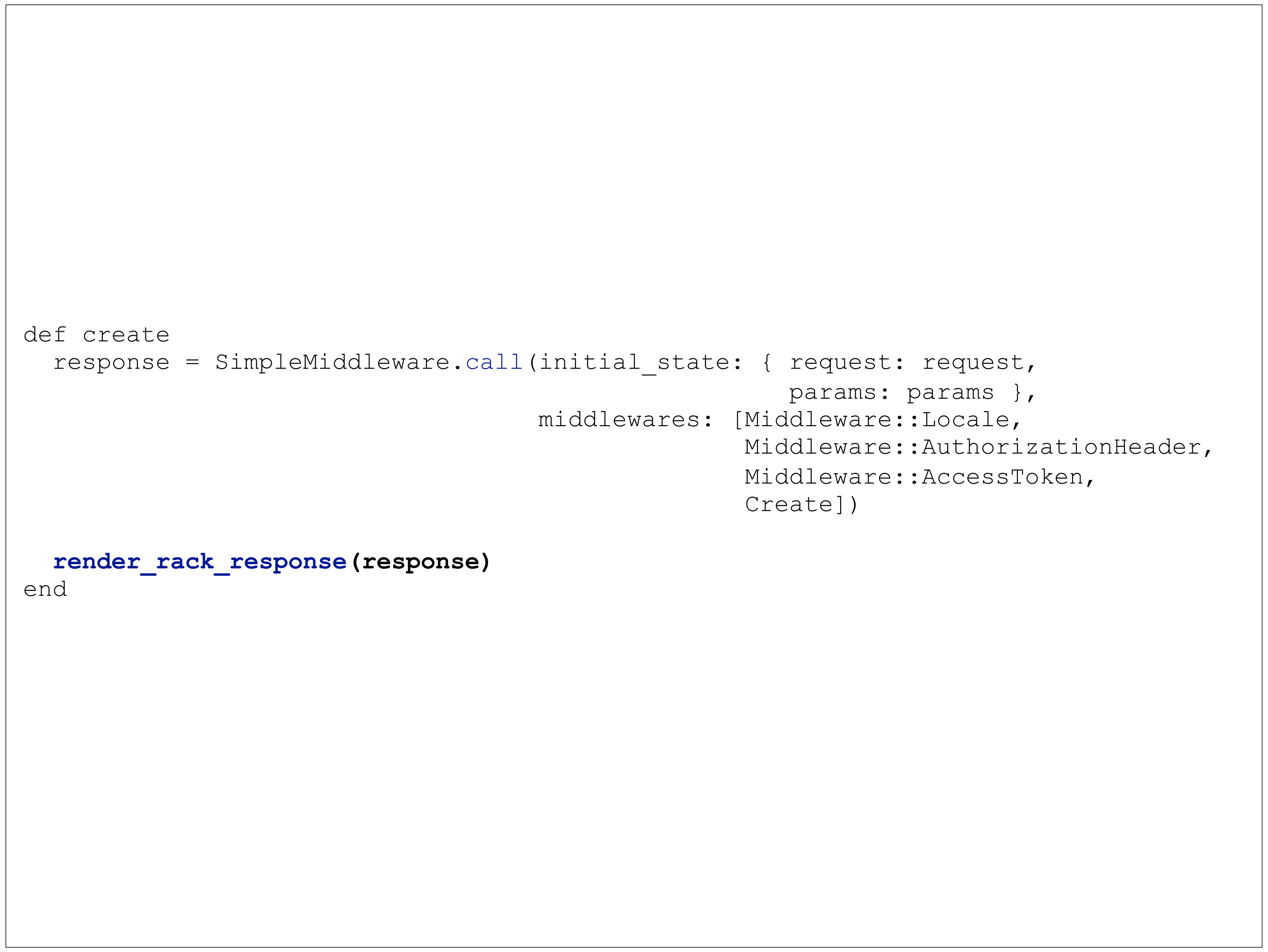
Now we check the header, with exactly the same code as before, and return an error if the header doesn’t look right.

```ruby
def call(state)
  header_value = state[:request].headers["HTTP_AUTHORIZATION"]

  return missing_access_token_error unless header_value.present?
  token_type, token = header_value.split(" ", 2)
  return missing_access_token_error unless token_type == "Bearer"

  # ...
end

private

def missing_access_token_error
  render status: 401,
         headers: { "Content-Type" => "application/json" },
         body: JSON.generate(errors: [I18n.t("errors.missing_access_token")])
end
```

SimpleMiddleware defines a `render` method. It’s just a simple helper…

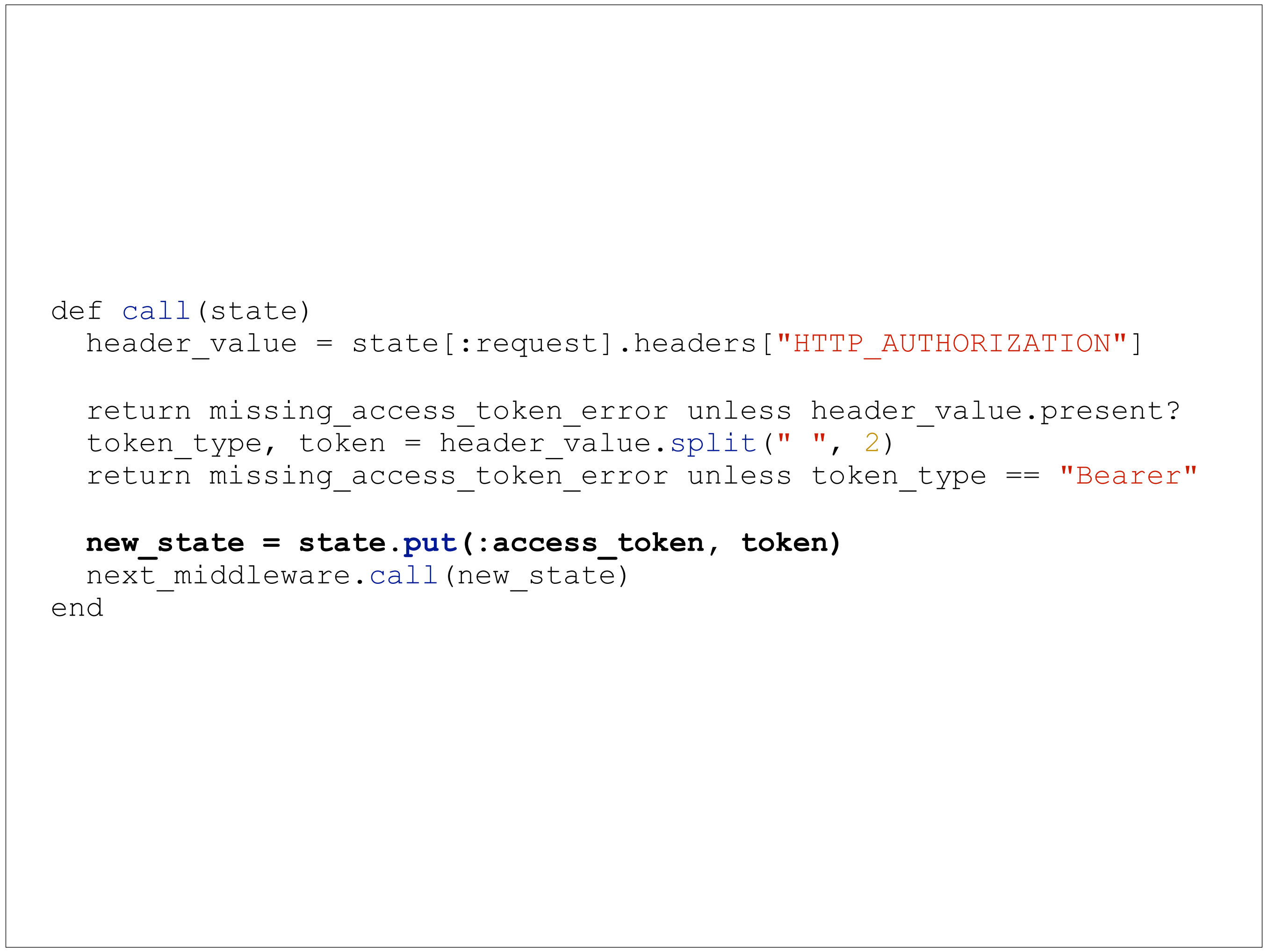
```
[
  401,
  { "Content-Type" => "application/json" },
  "{\"errors\":[\"Missing access token\"]}"
]
```

…that returns a Rack response, which is an array with a status code, response headers and response body.



```ruby
def create
  response = SimpleMiddleware.call(initial_state: { request: request,
                                                    params: params },
                                   middlewares: [Middleware::Locale,
                                                 Middleware::AuthorizationHeader,
                                                 Middleware::AccessToken,
                                                 Create])

  render_rack_response(response)
end
```

In our controller, we take that and render it back to the user.

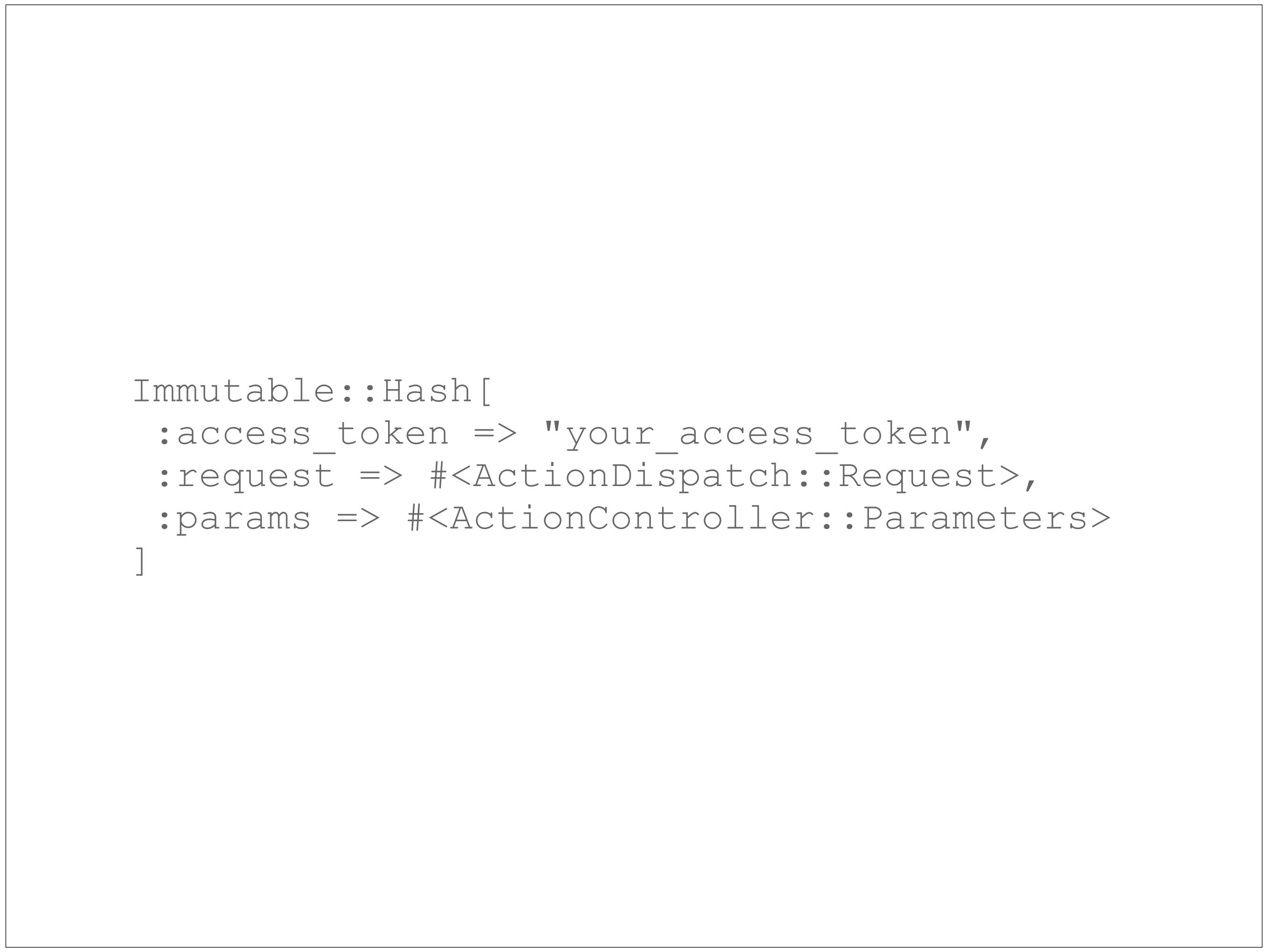


```
def call(state)
  header_value = state[:request].headers["HTTP_AUTHORIZATION"]

  return missing_access_token_error unless header_value.present?
  token_type, token = header_value.split(" ", 2)
  return missing_access_token_error unless token_type == "Bearer"

  new_state = state.put(:access_token, token)
  next_middleware.call(new_state)
end
```

What about the case where there's an access token in the header, and we don't need to return an error?

We want to pass the access token we've found on to the next middleware in the chain.

We call our immutable hash's `#put` method, which adds a key-value pair to the hash, and then returns the new version.

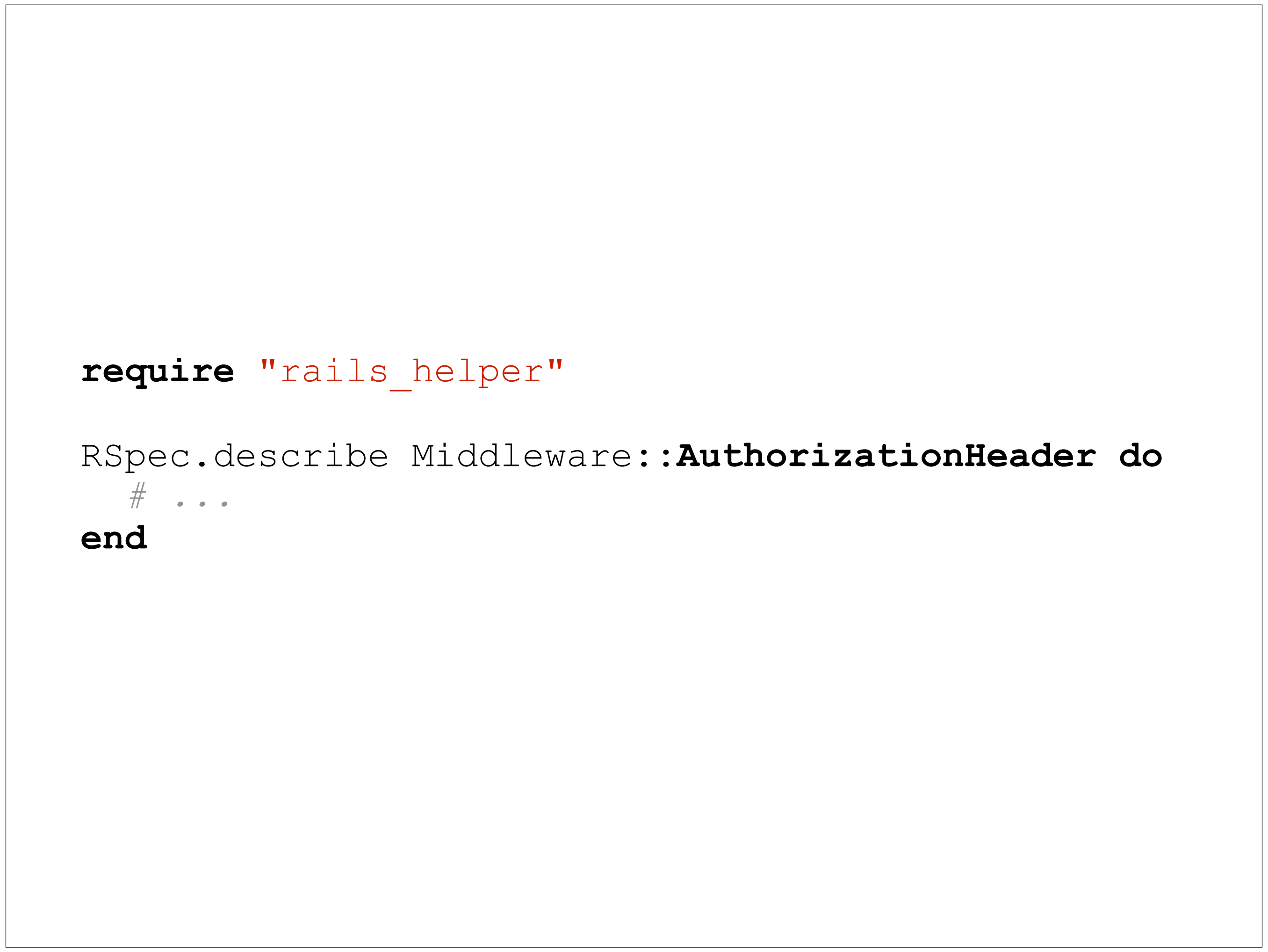


```
Immutable::Hash[
 :access_token => "your_access_token",
 :request => #<ActionDispatch::Request>,
 :params => #<ActionController::Parameters>
]
```

So our new state has three things in it: the initial state from before, so the request and the parameters, plus the access token that we’ve just added.

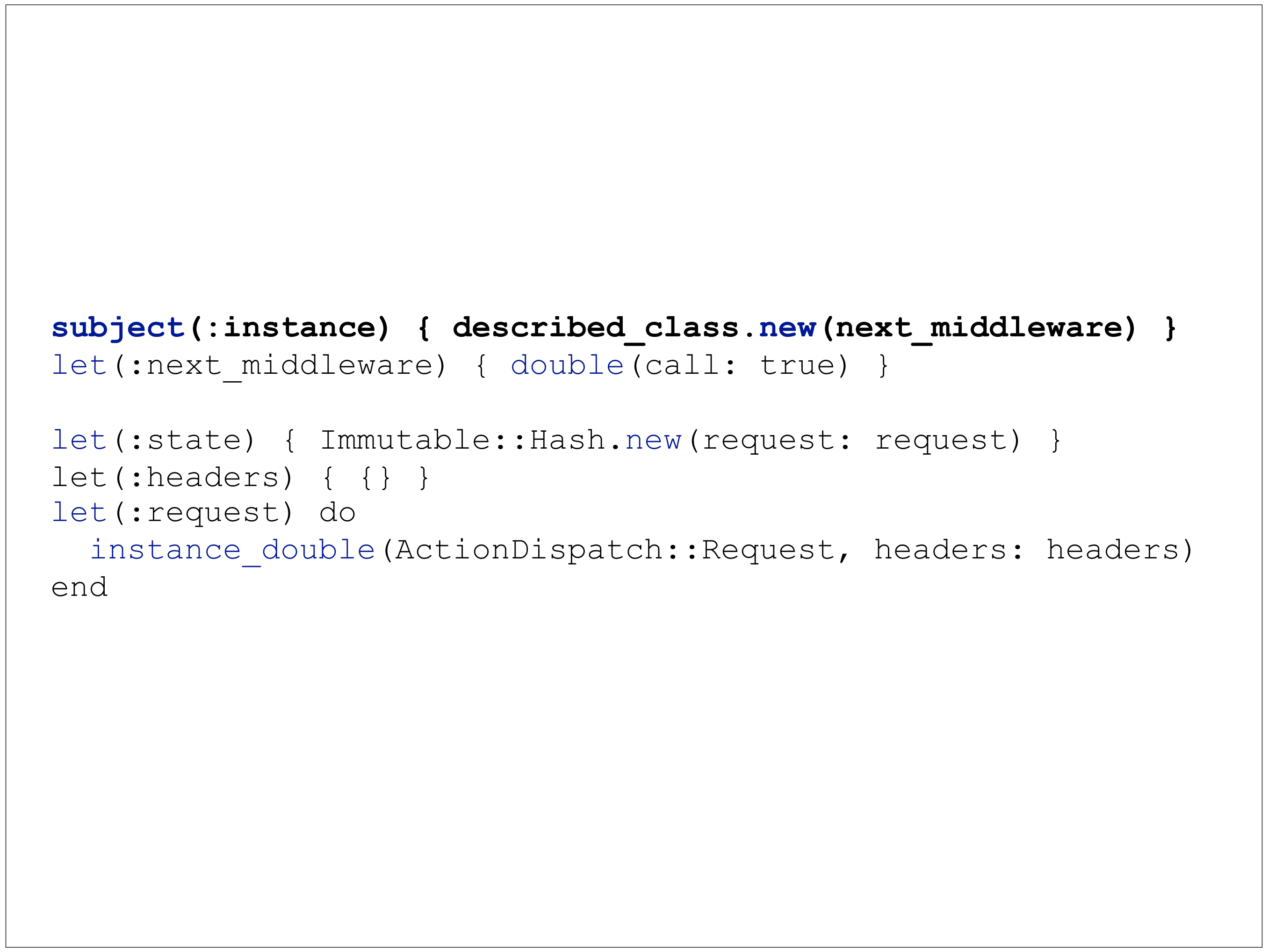
```
def call(state)
  header_value = state[:request].headers["HTTP_AUTHORIZATION"]

  return missing_access_token_error unless header_value.present?
  token_type, token = header_value.split(" ", 2)
  return missing_access_token_error unless token_type == "Bearer"

  new_state = state.put(:access_token, token)
  next_middleware.call(new_state)
end
```

We call the next middleware, passing on the new state. The next middleware will be responsible for checking if the access token exists.



```
require "rails_helper"

RSpec.describe Middleware::AuthorizationHeader do
  # ...
end
```

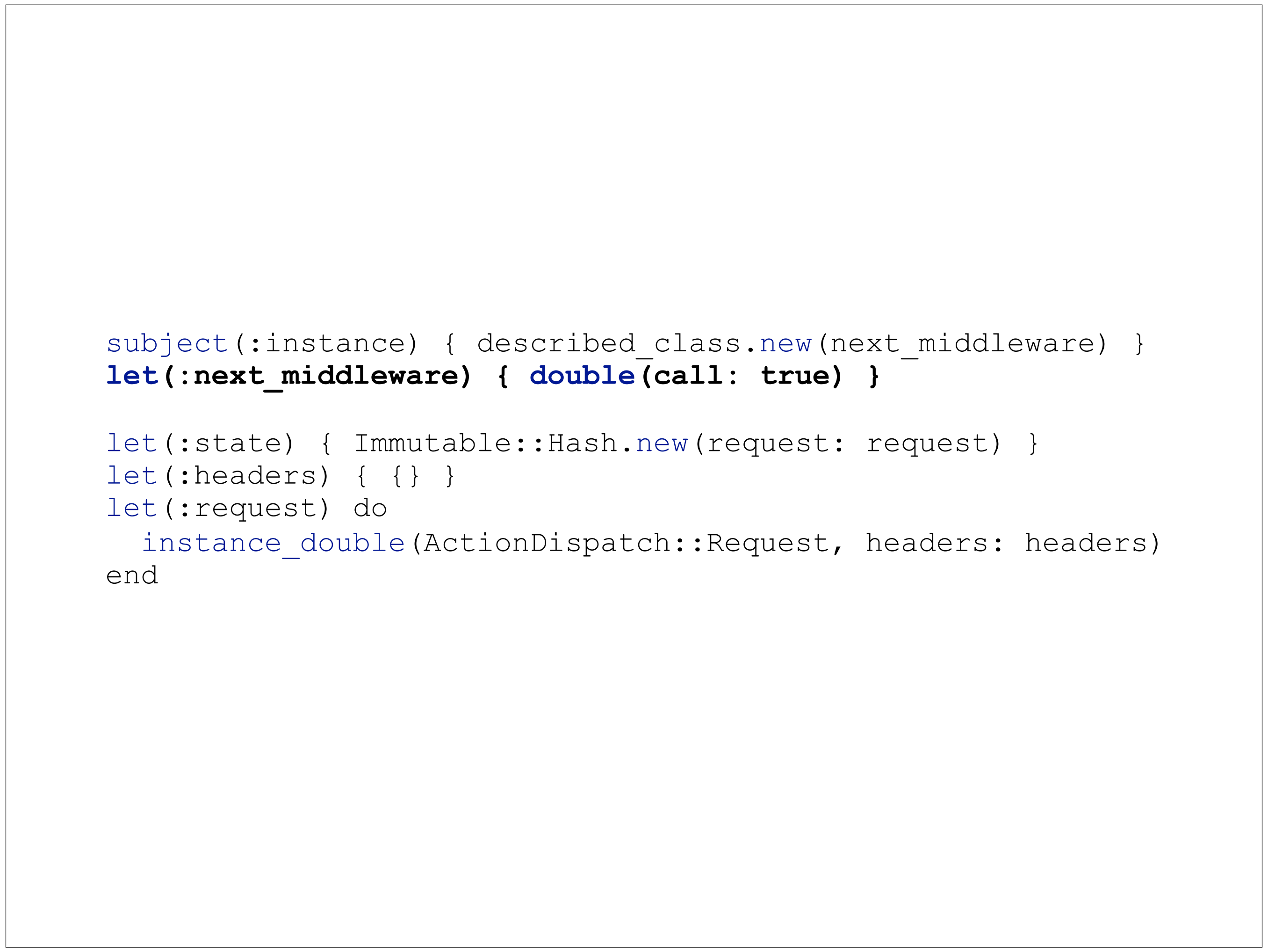
The beauty of the middleware pattern is what we can write simple unit tests, passing in the initial state and a fake “next middleware”, and then check what is returned, or whether the next middleware gets called.



```
subject(:instance) { described_class.new(next_middleware) }
let(:next_middleware) { double(call: true) }

let(:state) { Immutable::Hash.new(request: request) }
let(:headers) { {} }
let(:request) do
  instance_double(ActionDispatch::Request, headers: headers)
end
```

First, let’s look at the setup we need for our tests.

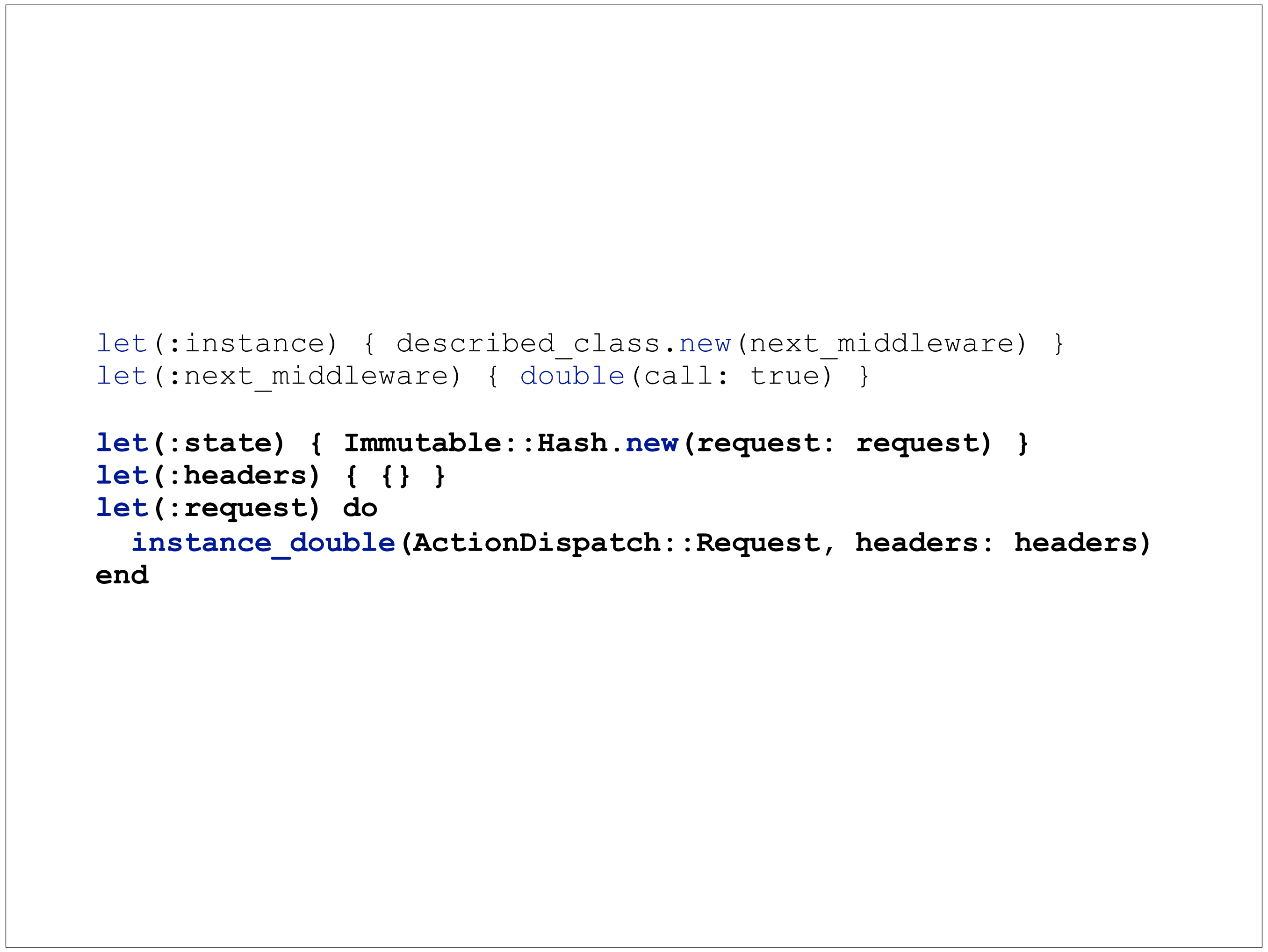
The subject of our tests is an instance of the middleware. When we instantiate a middleware, we pass in the next middleware in the pipeline.



```
subject(:instance) { described_class.new(next_middleware) }
let(:next_middleware) { double(call: true) }

let(:state) { Immutable::Hash.new(request: request) }
let(:headers) { {} }
let(:request) do
  instance_double(ActionDispatch::Request, headers: headers)
end
```

The next middleware is just an RSpec double.

This is just an object which responds to `call`, and does nothing. In our tests, we can check that the middleware passes on to the next middleware in the chain by setting an RSpec message expectation. We'll see that in a couple of seconds.



```ruby
let(:instance) { described_class.new(next_middleware) }
let(:next_middleware) { double(call: true) }

let(:state) { Immutable::Hash.new(request: request) }
let(:headers) { {} }
let(:request) do
  instance_double(ActionDispatch::Request, headers: headers)
end
```

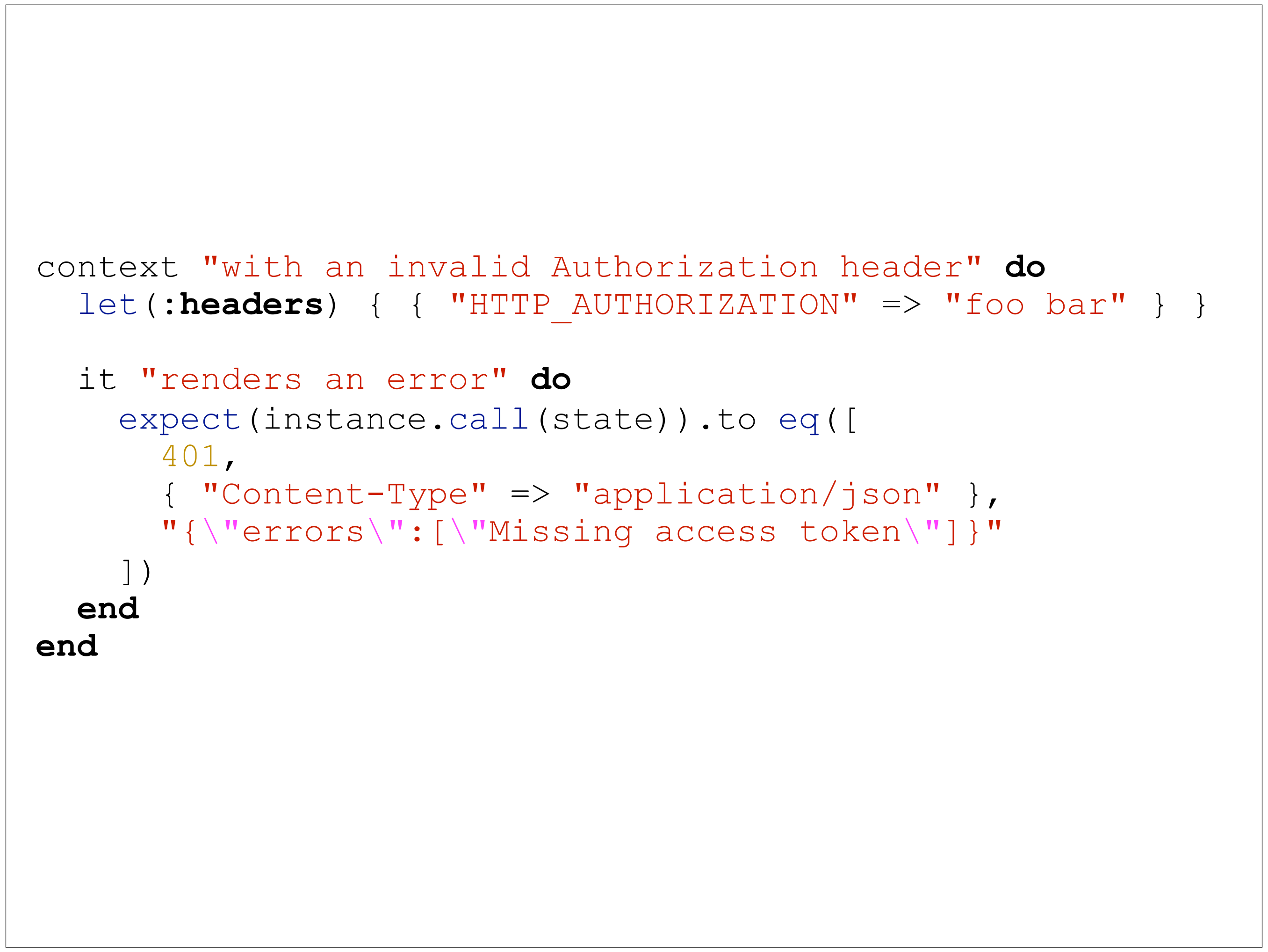
Our initial state is just a fake Rails request. The middleware will refer to this to grab the Authorization header.

You might remember that in our controller, we passed in the request *and* the parameters object. We won't do that here, as this middleware doesn't need the params. We might as well just pass in the bare minimum state required.

```ruby
context "with no Authorization header" do
  let(:headers) { {} }

  it "renders an error" do
    expect(instance.call(state)).to eq([
      401,
      { "Content-Type" => "application/json" },
      "{\"errors\":[\"Missing access token\"]}"
    ])
  end
end
```

Let’s write our first spec: the case when there’s no Authorization header.

We pass in the initial state - a request with no headers - and check what is returned: a 401 error with the correct body and response headers. This return value is a Rack response - an array with a status code, response headers and response body.



```ruby
context "with an invalid Authorization header" do
  let(:headers) { { "HTTP_AUTHORIZATION" => "foo bar" } }

  it "renders an error" do
    expect(instance.call(state)).to eq([
      401,
      { "Content-Type" => "application/json" },
      "{\"errors\":[\"Missing access token\"]}"
    ])
  end
end
```

The case where the header isn't structured correctly is very similar. We pass in some state - this time, a wrongly-formatted Authorization header - and check that the right error is returned.

```ruby
context "with a correctly-structured Authorization header" do
  let(:headers) { { "HTTP_AUTHORIZATION" => "Bearer your_access_token" } }

  it "calls the next middleware, passing on the access token" do
    expect(next_middleware).to receive(:call).
      with(Immutable::Hash.new(request: request,
                               access_token: "your_access_token"))

    instance.call(state)
  end
end
```

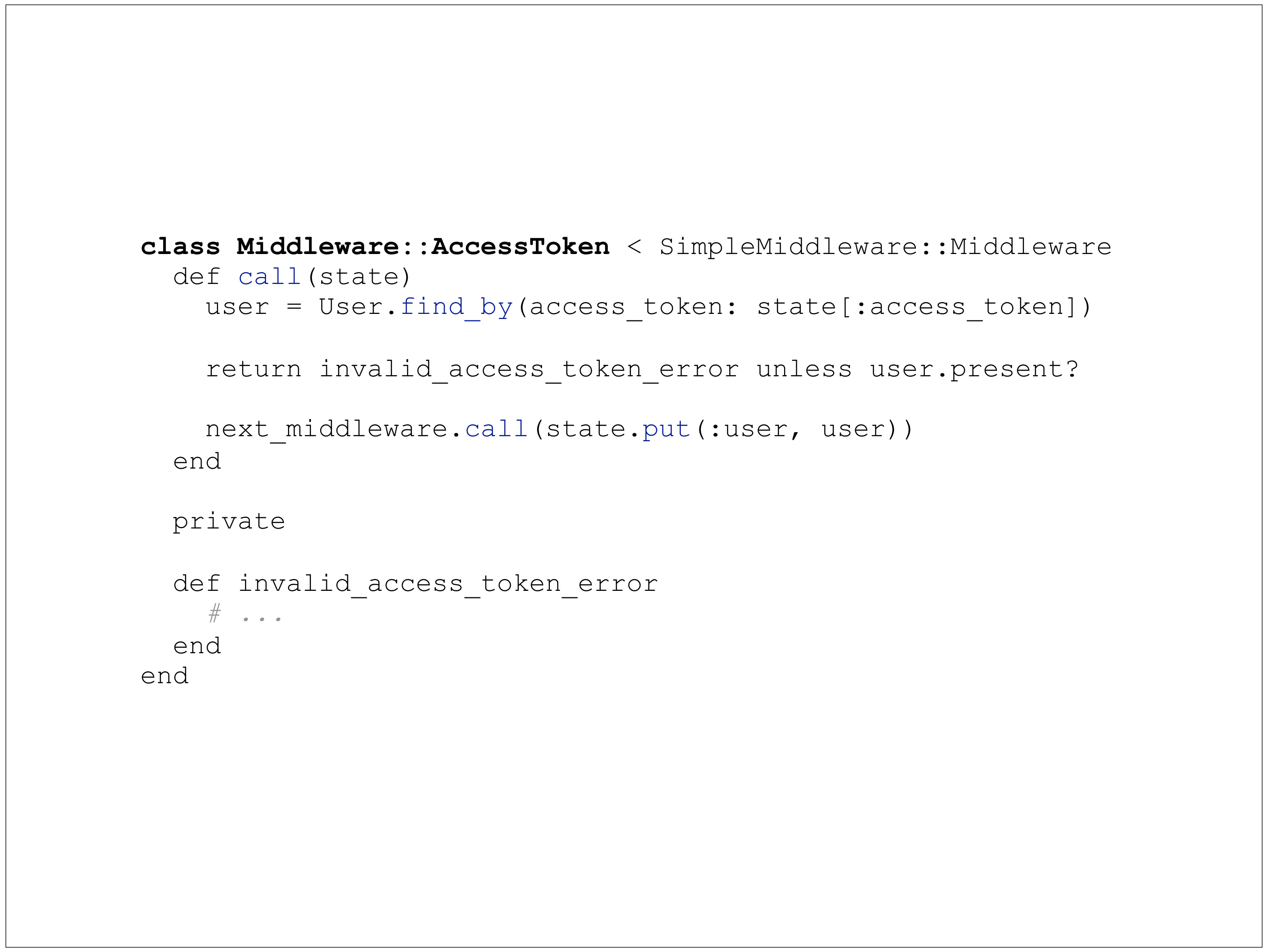
The final case we need to test is where the Authorization header is present and valid.

In that case, we want to make sure that the next middleware in the chain is called, and is passed the original state, plus the access token that has been pulled out from the header.

We use a RSpec message expectation to say that the next middleware should be called with the correct parameters.

```
context "with a correctly-structured Authorization header" do
  let(:headers) { { "HTTP_AUTHORIZATION" => "Bearer your_access_token" } }

  it "calls the next middleware, passing on the access token" do
    expect(next_middleware).to receive(:call).
      with(Immutable::Hash.new(request: request,
                               access_token: "your_access_token"))

    instance.call(state)
  end
end
```

We don’t expect anything to be returned - that’s the job of the next middleware in the chain.



```
class Middleware::AccessToken < SimpleMiddleware::Middleware
  def call(state)
    user = User.find_by(access_token: state[:access_token])

    return invalid_access_token_error unless user.present?

    next_middleware.call(state.put(:user, user))
  end

  private

  def invalid_access_token_error
    # ...
  end
end
```

Now, let’s go through the access token middleware.

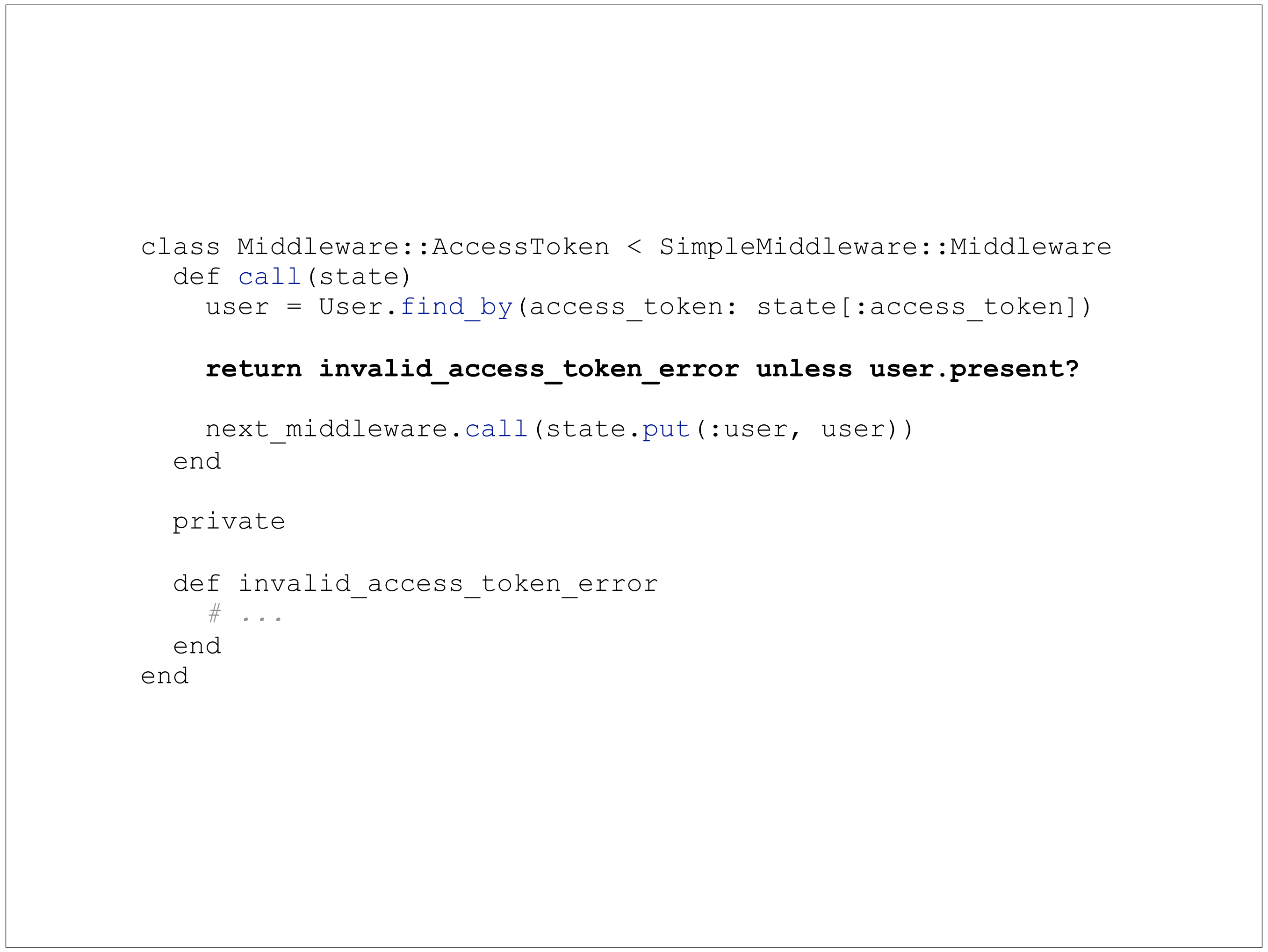
We’ll go through this one a bit more quickly, since we’ve already seen an example.

```
class Middleware::AccessToken < SimpleMiddleware::Middleware
  def call(state)
    user = User.find_by(access_token: state[:access_token])

    return invalid_access_token_error unless user.present?

    next_middleware.call(state.put(:user, user))
  end

  private

  def invalid_access_token_error
    # ...
  end
end
```

We pull out the access token from the state, which has been added by the “authorization header” middleware, and we look for a user with a matching access token.

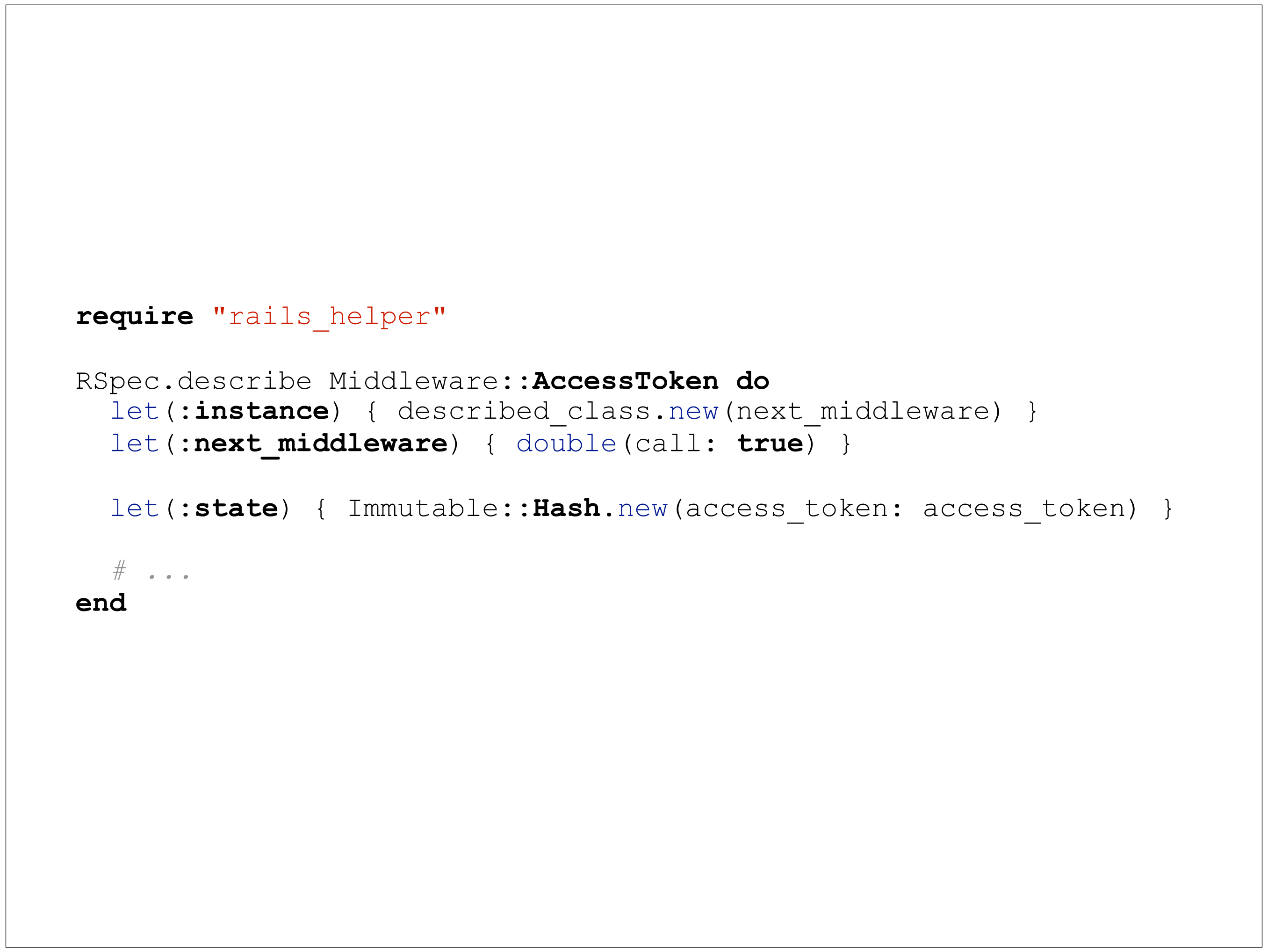
```
class Middleware::AccessToken < SimpleMiddleware::Middleware
  def call(state)
    user = User.find_by(access_token: state[:access_token])

    return invalid_access_token_error unless user.present?

    next_middleware.call(state.put(:user, user))
  end

  private

  def invalid_access_token_error
    # ...
  end
end
```

If there’s a match, we call the next middleware, passing on the authenticated user.



```ruby
class Middleware::AccessToken < SimpleMiddleware::Middleware
  def call(state)
    user = User.find_by(access_token: state[:access_token])

    return invalid_access_token_error unless user.present?

    next_middleware.call(state.put(:user, user))
  end

  private

  def invalid_access_token_error
    # ...
  end
end
```

If there isn’t a match, we return an error.

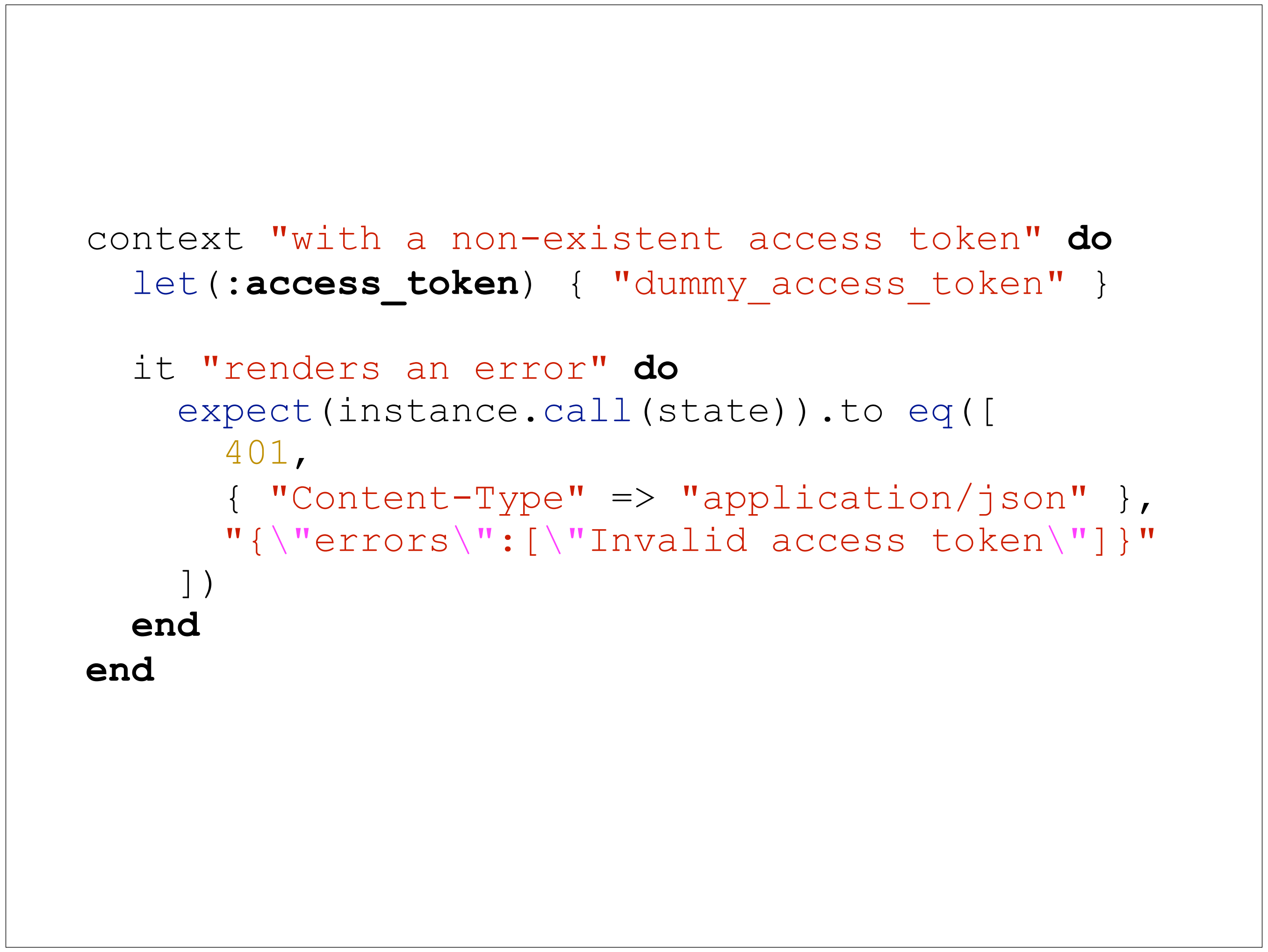


```ruby
require "rails_helper"

RSpec.describe Middleware::AccessToken do
  let(:instance) { described_class.new(next_middleware) }
  let(:next_middleware) { double(call: true) }

  let(:state) { Immutable::Hash.new(access_token: access_token) }

  # ...
end
```

The specs for our second middleware have pretty much the same setup as the first one.

We set up our middleware instance, passing in a dummy “next middleware”, and build the state. This middleware only needs an access token, so we only pass that in.

So, what cases do we need to test? We want to test when the access token does exist, and when there isn’t a match.



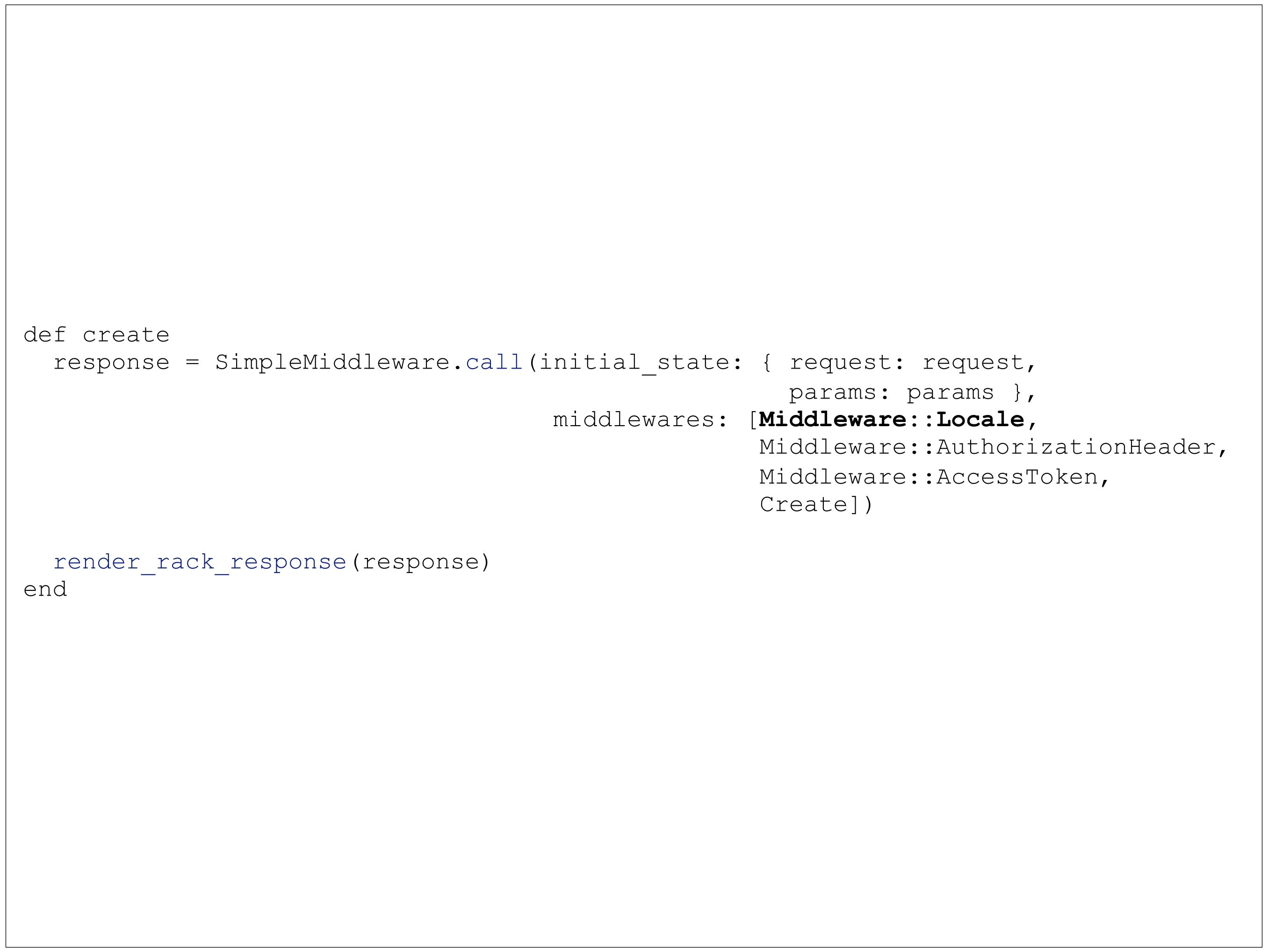
```ruby
context "with a non-existent access token" do
  let(:access_token) { "dummy_access_token" }

  it "renders an error" do
    expect(instance.call(state)).to eq([
      401,
      { "Content-Type" => "application/json" },
      "{\"errors\":[\"Invalid access token\"]}"
    ])
  end
end
```

We'll start with the sad path, giving a non-existent access token, and checking that the right error is returned.

```ruby
context "with a valid access token" do
  let(:user) { FactoryBot.create(:user) }
  let(:access_token) { user.access_token }

  it "calls the next middleware, passing on the user” do
    expect(next_middleware).to receive(:call).
      with(Immutable::Hash.new(access_token: access_token, user: user))

    instance.call(state)
  end
end
```

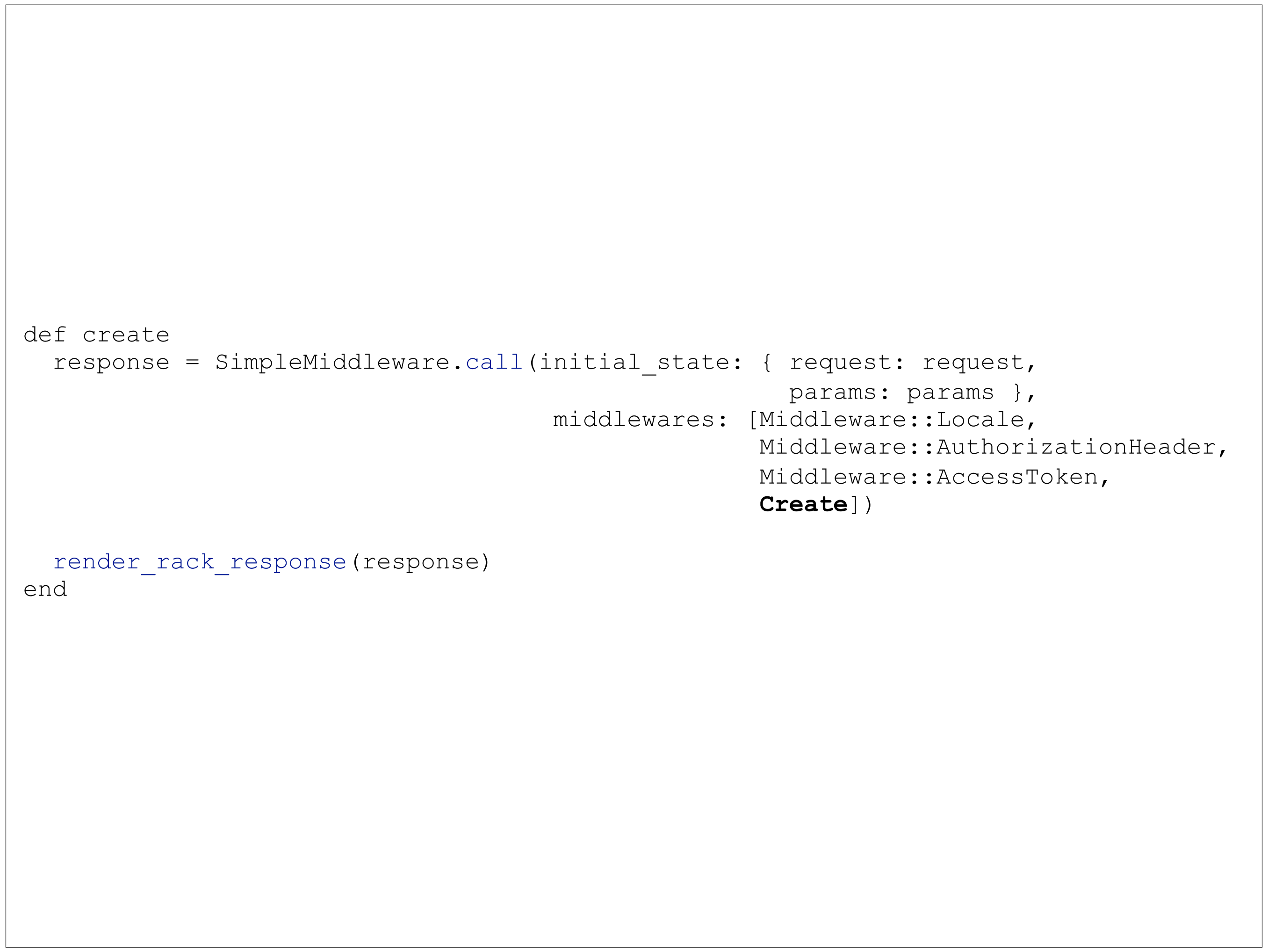
And then we’ll test the case where the access token is valid.

We expect to hand off to the next middleware in the chain, passing in the authenticated user.



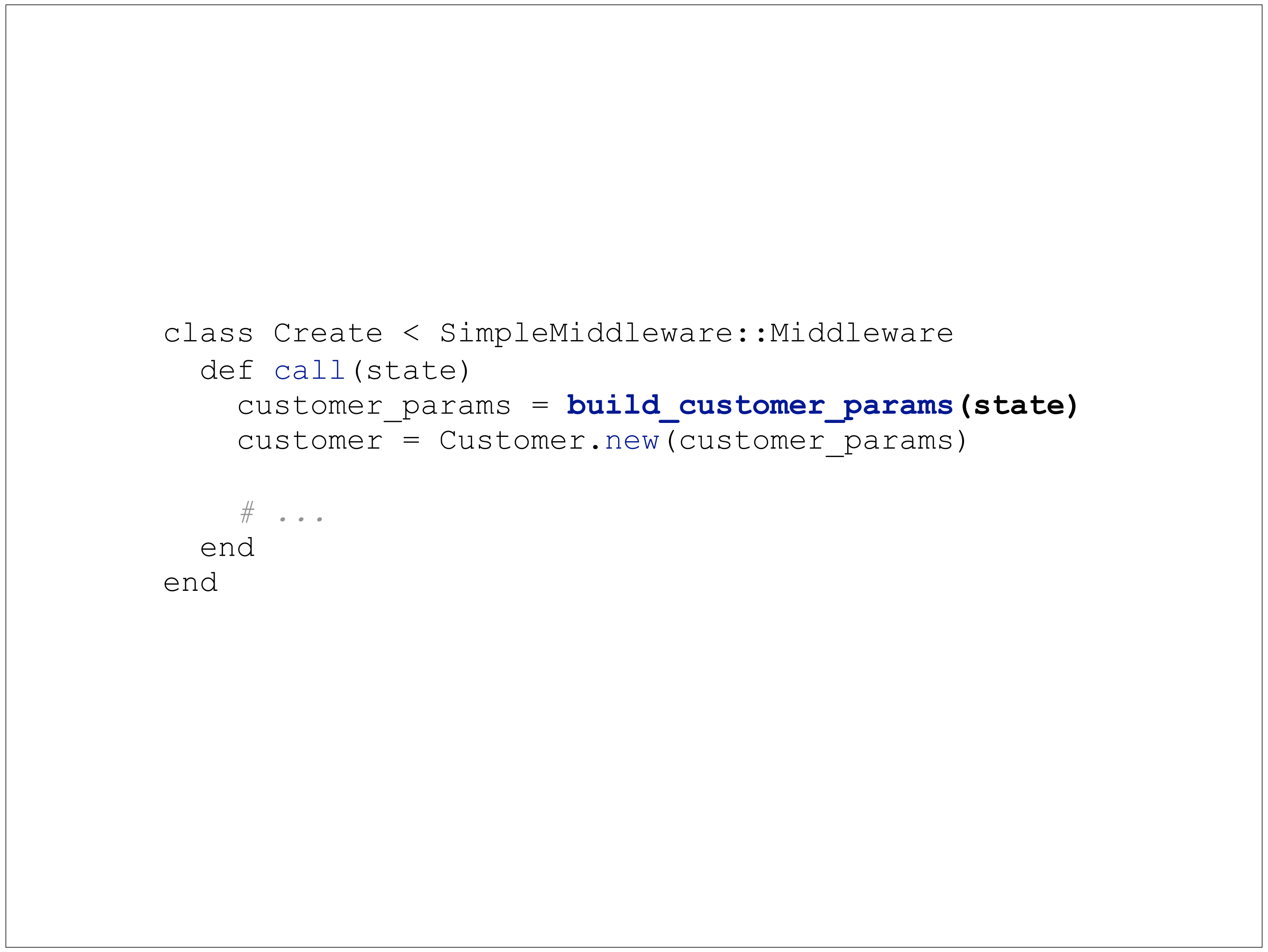
```
def create
  response = SimpleMiddleware.call(initial_state: { request: request,
                                                    params: params },
                                   middlewares: [Middleware::Locale,
                                                 Middleware::AuthorizationHeader,
                                                 Middleware::AccessToken,
                                                 Create])

  render_rack_response(response)
end
```

We’re going to skip our middleware for setting the locale for now.



```
def create
  response = SimpleMiddleware.call(initial_state: { request: request,
                                                    params: params },
                                    middlewares: [Middleware::Locale,
                                                  Middleware::AuthorizationHeader,
                                                  Middleware::AccessToken,
                                                  Create])

  render_rack_response(response)
end
```

So all we have left to re-write is our endpoint-specific code, as our `Create` middleware.



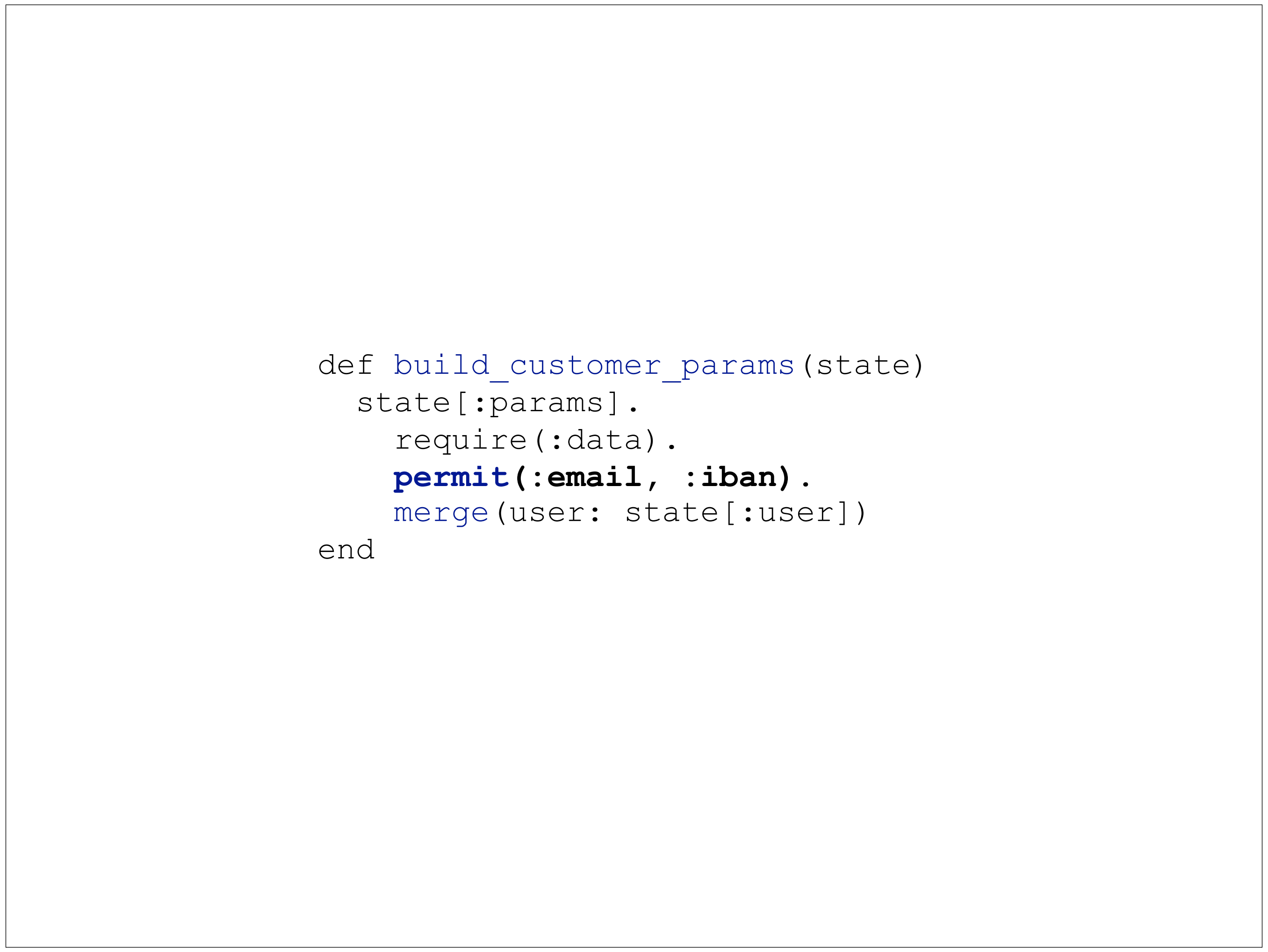
```ruby
class Create < SimpleMiddleware::Middleware
  def call(state)
    customer_params = build_customer_params(state)
    customer = Customer.new(customer_params)

    # ...
  end
end
```

Here, we want to build a customer, try to save it, and then either render back the created customer, or return an error.

We use the state to build the customer attributes. What does that look like?

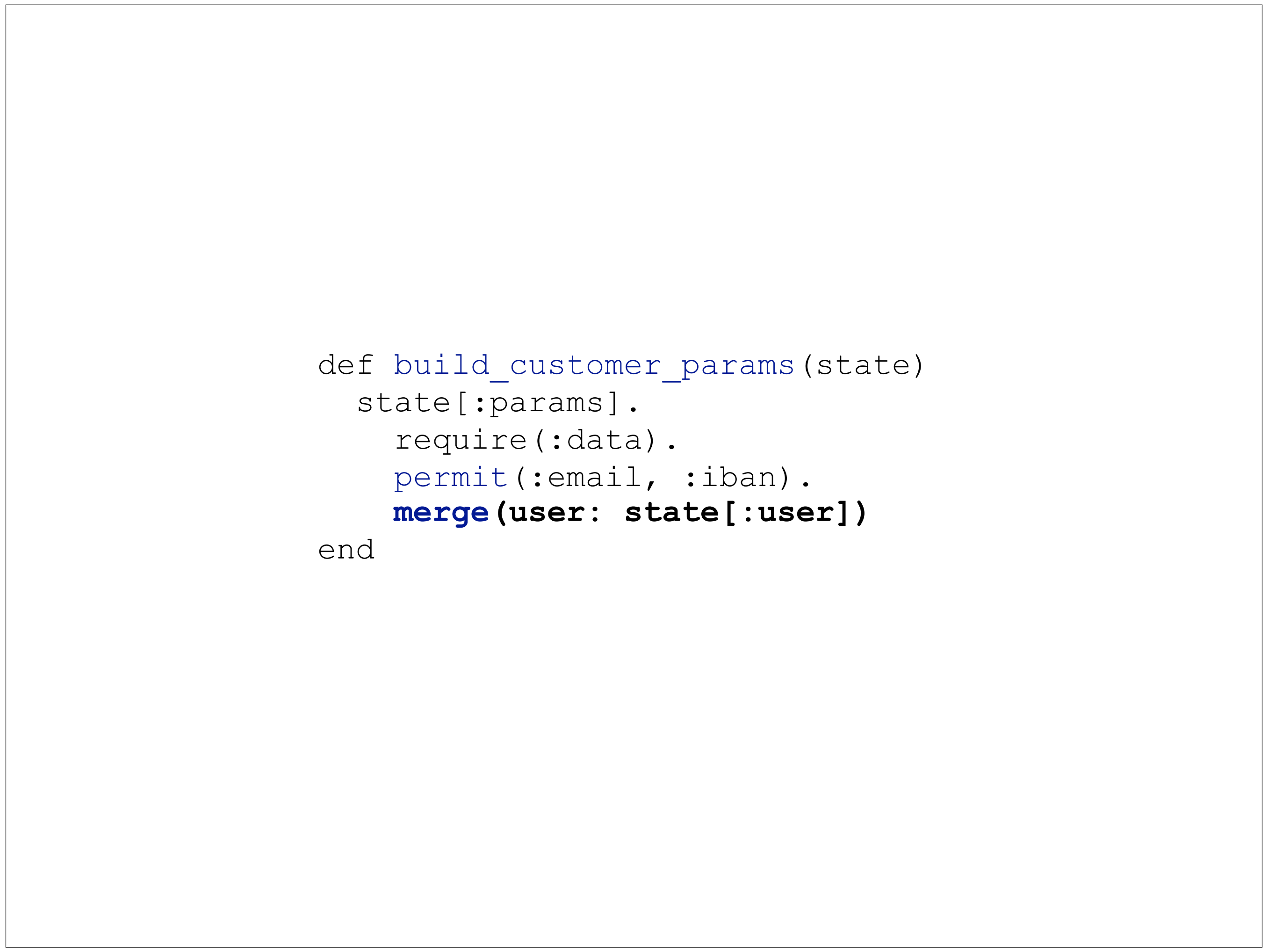
```ruby
def build_customer_params(state)
  state[:params].
    require(:data).
    permit(:email, :iban).
    merge(user: state[:user])
end
```

In the state, we have the parameters, which is an ActionController::Parameters instance, and the authenticated user.



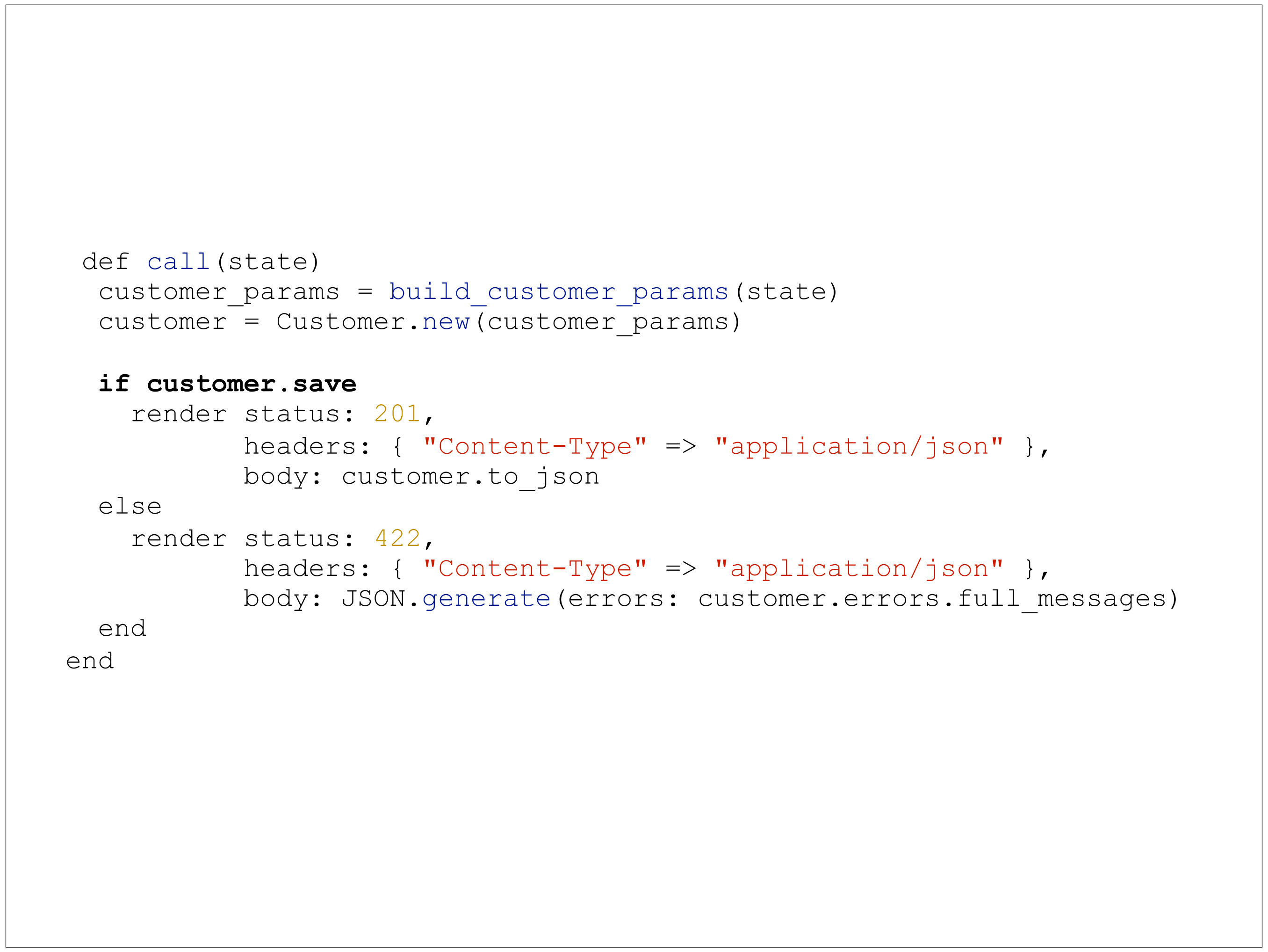
```ruby
def build_customer_params(state)
  state[:params].
    require(:data).
    permit(:email, :iban).
    merge(user: state[:user])
end
```

We pull out the “email” and “IBAN” from the request parameters



```
def build_customer_params(state)
  state[:params].
    require(:data).
    permit(:email, :iban).
    merge(user: state[:user])
end
```

…and then we merge in the authenticated user, so the customer will belong to them.

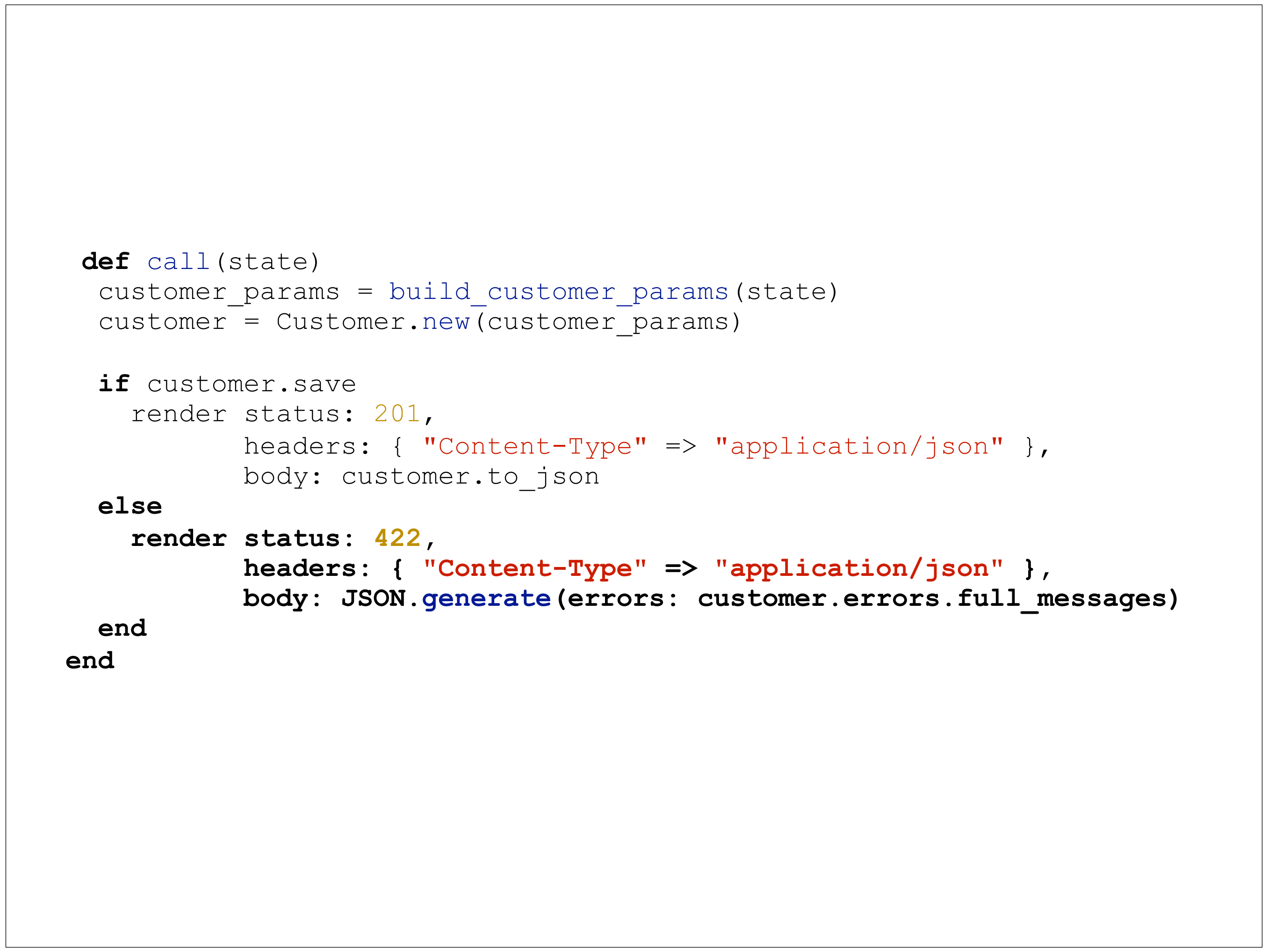


```
 def call(state)
  customer_params = build_customer_params(state)
  customer = Customer.new(customer_params)

  if customer.save
    render status: 201,
           headers: { "Content-Type" => "application/json" },
           body: customer.to_json
  else
    render status: 422,
           headers: { "Content-Type" => "application/json" },
           body: JSON.generate(errors: customer.errors.full_messages)
  end
end
```

Having built the customer, we try to save it

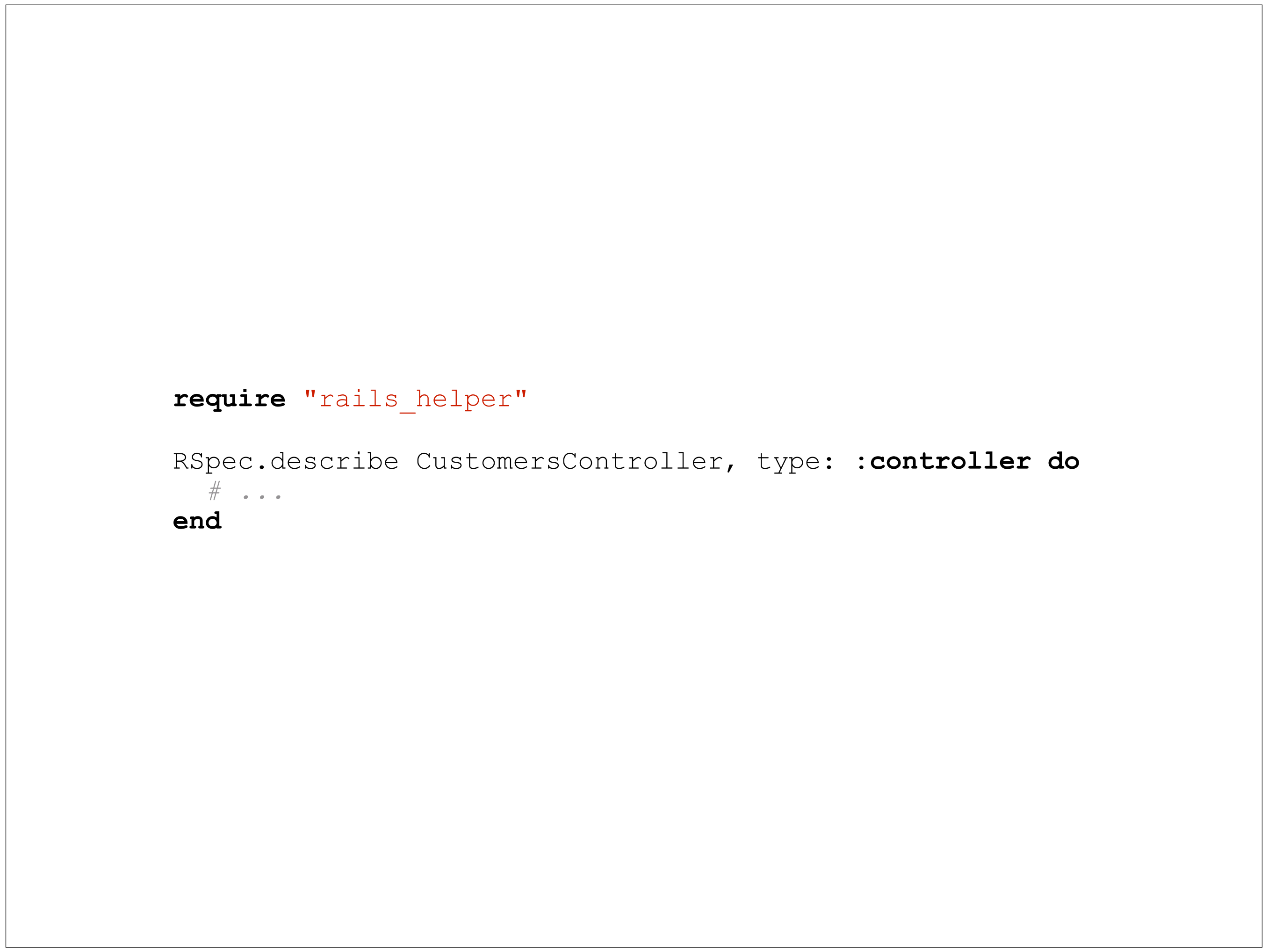
```ruby
 def call(state)
  customer_params = build_customer_params(state)
  customer = Customer.new(customer_params)

  if customer.save
    render status: 201,
           headers: { "Content-Type" => "application/json" },
           body: customer.to_json
  else
    render status: 422,
           headers: { "Content-Type" => "application/json" },
           body: JSON.generate(errors: customer.errors.full_messages)
  end
end
```

If that’s successful, we render the created customer



```ruby
 def call(state)
  customer_params = build_customer_params(state)
  customer = Customer.new(customer_params)

  if customer.save
    render status: 201,
           headers: { "Content-Type" => "application/json" },
           body: customer.to_json
  else
    render status: 422,
           headers: { "Content-Type" => "application/json" },
           body: JSON.generate(errors: customer.errors.full_messages)
  end
end
```

If something goes wrong, we render back the validation errors.



```ruby
require "rails_helper"

RSpec.describe CustomersController, type: :controller do
  # ...
end
```

If everything has gone to plan and we haven't made any mistakes, our controller specs will still pass. Hooray! We've refactored our Rails API using middleware, and have made it miles more maintainable.

https://github.com/timrogers/simple_middleware_example

As a reminder, the sample application we've just built is on GitHub - you'll find it at https://github.com/timrogers/simple_middleware_example

We've learnt to make APIs miles more maintainable with middleware

I'm convinced from my experience of using this pattern that it makes your code miles more maintainable

But I wouldn't *use* Simple Middleware

But I wouldn’t actually use Simple Middleware

We can do *way* better in terms of developer experience

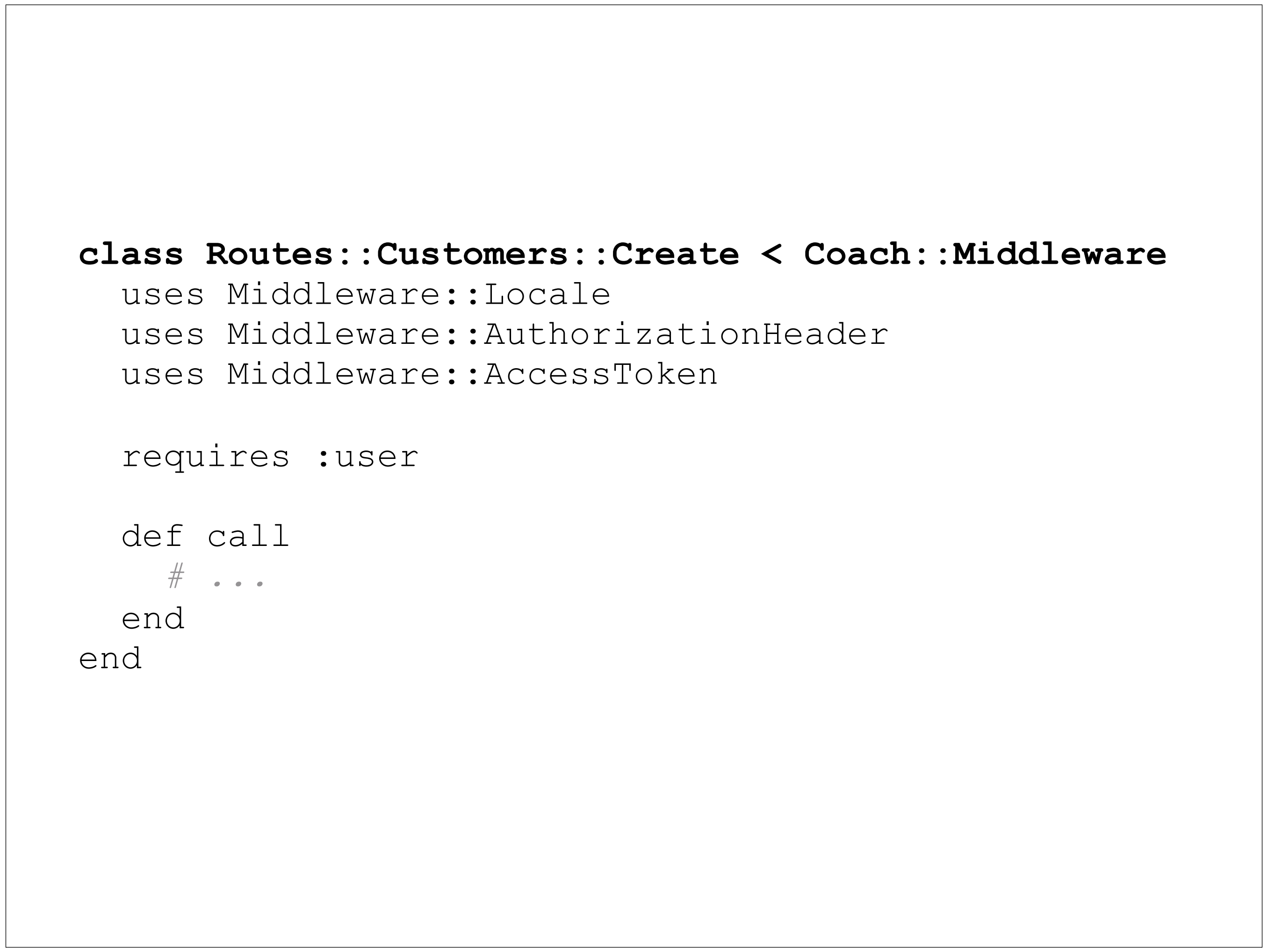
With a more fully-featured middleware library, we can make our APIs even more maintainable. Simple Middleware is missing a lot of sugar which can help to make the developer experience fantastic.

But using the simplest implementation possible helps you to understand the concepts

But using a really simple implementation of the middleware pattern helps us to understand the core principles, without getting distracted by more complicated features.

At GoCardless, we built Coach.

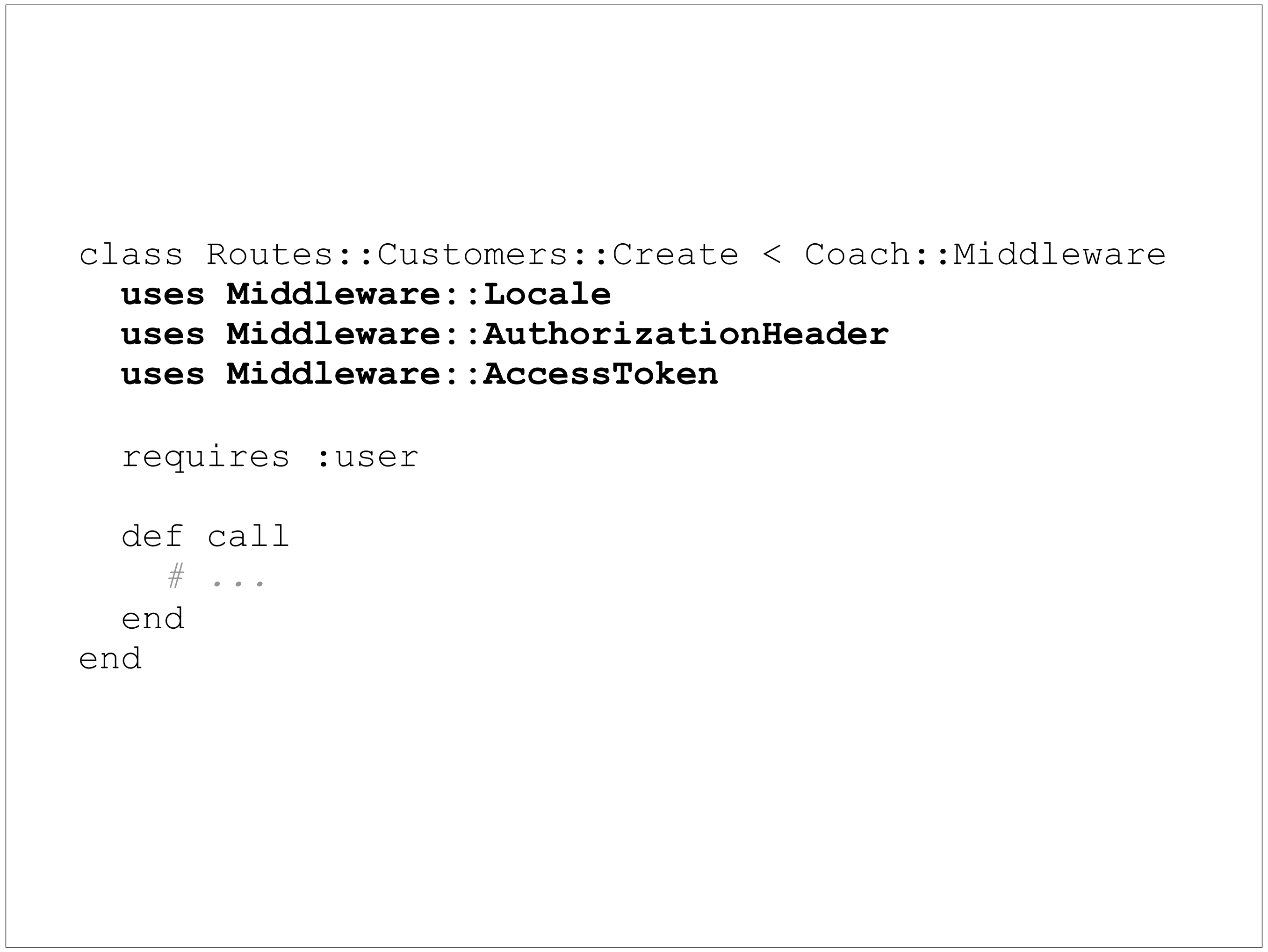
At GoCardless, we built Coach, a more fully-feature implementation of the pattern, with a bunch of extra features. I'll take you through just a few of them.



```ruby
class Routes::Customers::Create < Coach::Middleware
  uses Middleware::Locale
  uses Middleware::AuthorizationHeader
  uses Middleware::AccessToken

  requires :user

  def call
    # ...
  end
end
```

Firstly, with Coach, middleware can *depend on* other middleware. This makes it easier to build your pipeline.

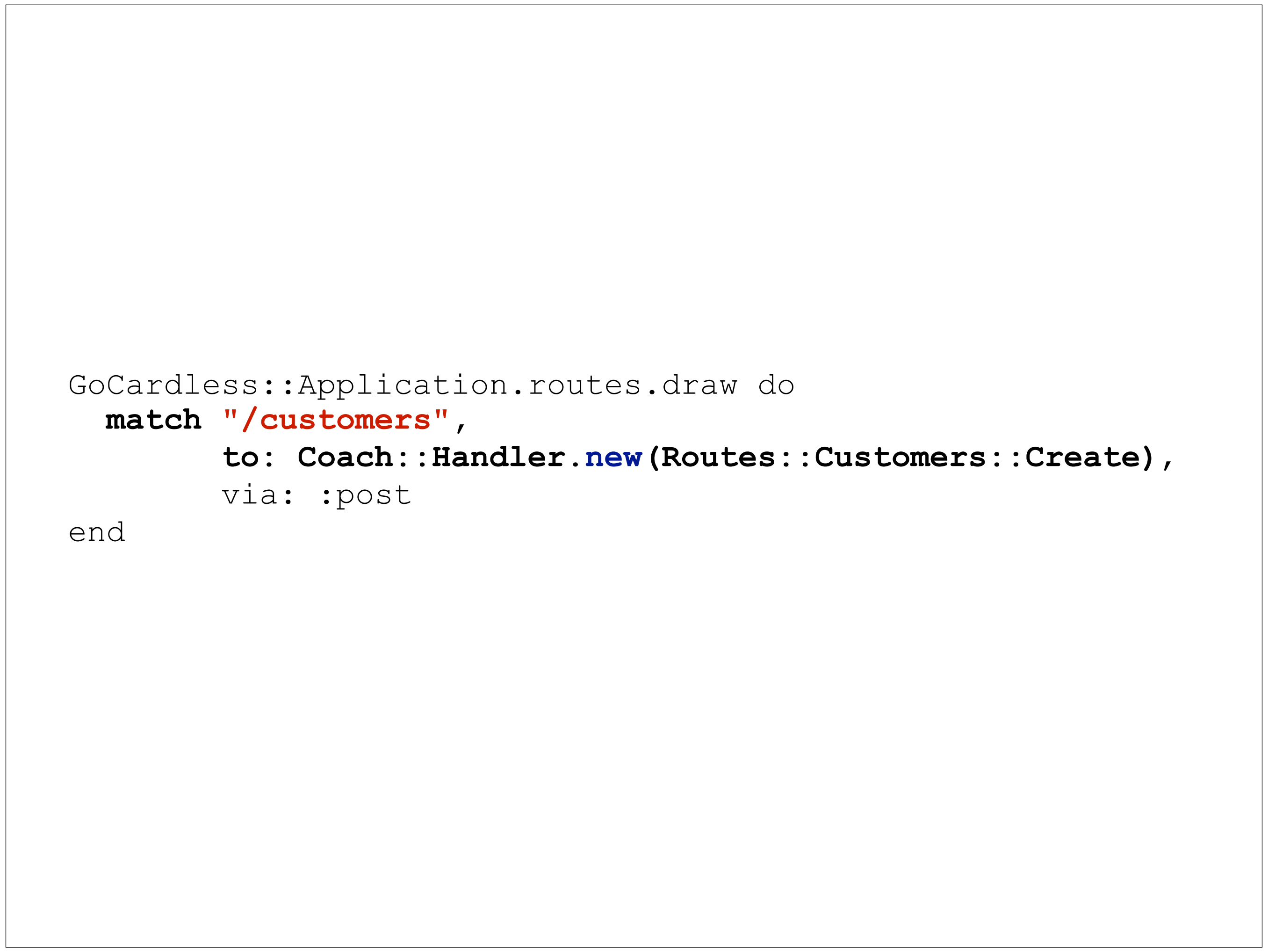
For example, here we define the endpoint-specific code as a middleware, called Routes::CustomersCreate



```
class Routes::Customers::Create < Coach::Middleware
  uses Middleware::Locale
  uses Middleware::AuthorizationHeader
  uses Middleware::AccessToken

  requires :user

  def call
    # ...
  end
end
```

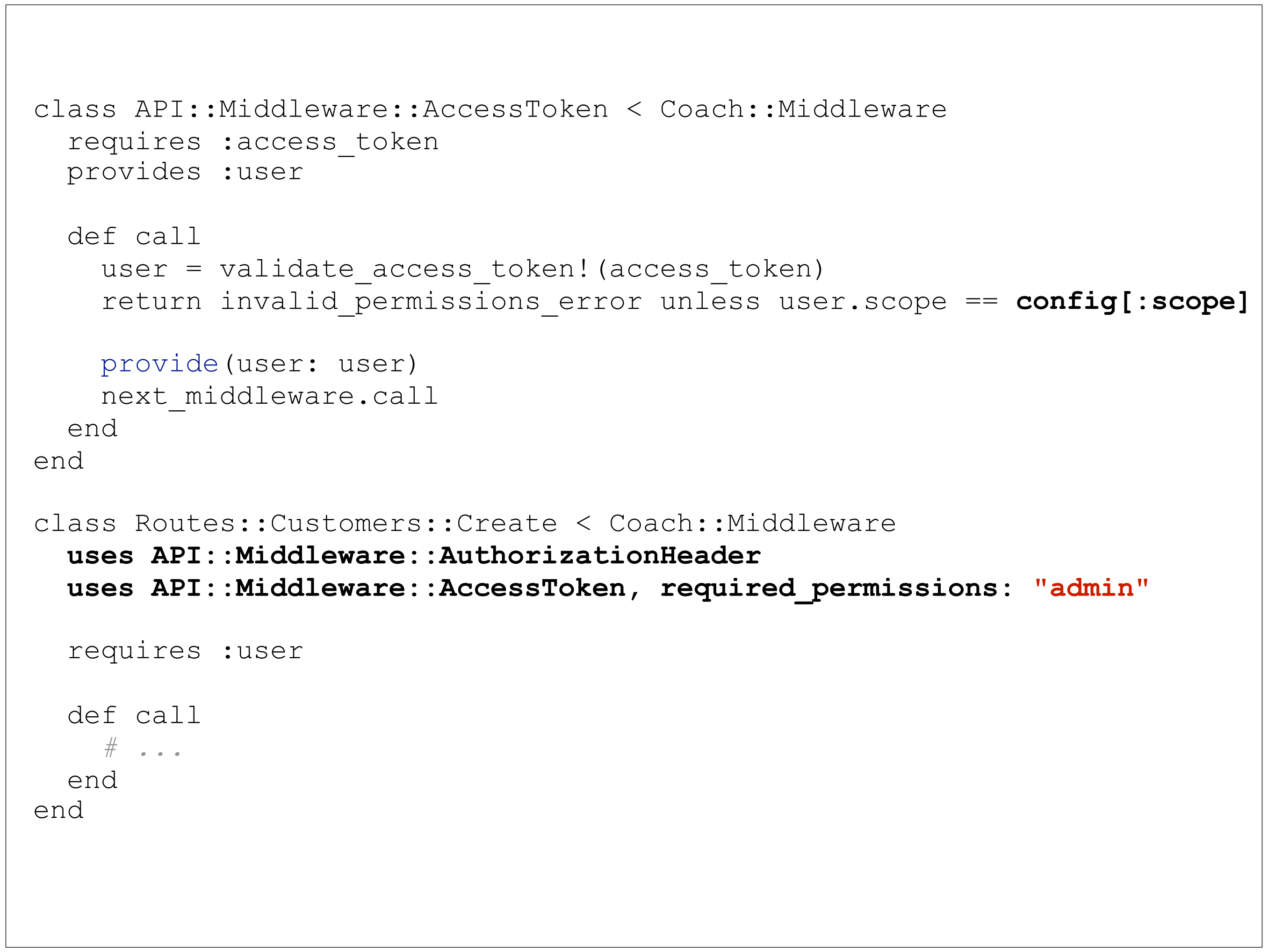
and then we make it *depend on* our three other middleware.

At GoCardless, we refer to this endpoint-specific code as a “route”, but really, it’s just a middleware.



```
GoCardless::Application.routes.draw do
  match "/customers",
        to: Coach::Handler.new(Routes::Customers::Create),
        via: :post
end
```

Making it work that way means we can hook our middleware pipeline straight into the Rails router, completely skipping ActionController. We still love Rails - especially ActiveModel - and Ruby in general, but we find the middleware pattern works much better.



```
class API::Middleware::AccessToken < Coach::Middleware
  requires :access_token
  provides :user

  def call
    user = validate_access_token!(access_token)
    return invalid_permissions_error unless user.scope == config[:scope]

    provide(user: user)
    next_middleware.call
  end
end

class Routes::Customers::Create < Coach::Middleware
  uses API::Middleware::AuthorizationHeader
  uses API::Middleware::AccessToken, required_permissions: "admin"

  requires :user

  def call
    # ...
  end
end
```

Middlewares can even be configured, making them even more adaptable and reusable.

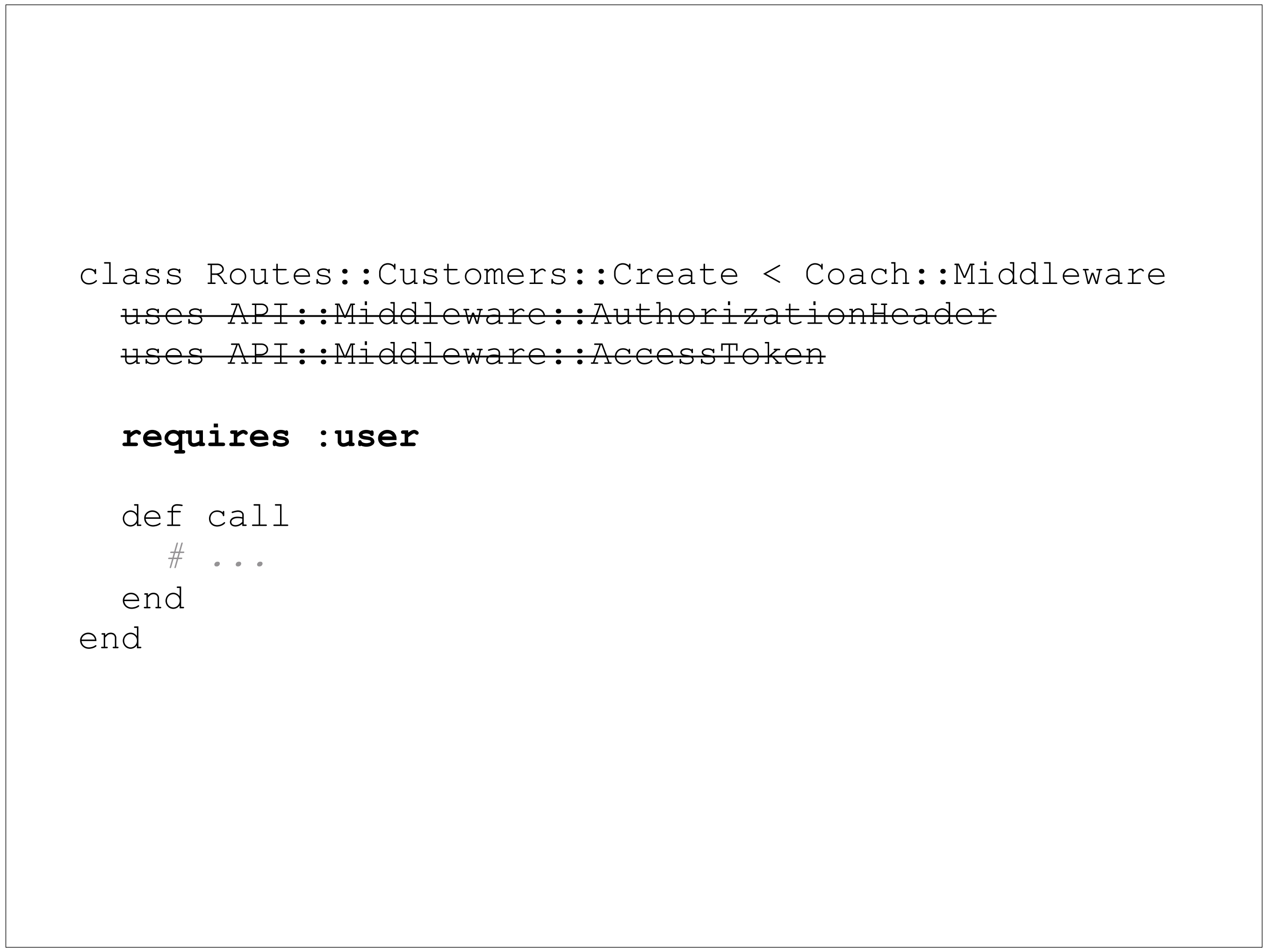
When we depend on the middleware by calling the `uses` method, we can pass in configuration, which we can then refer to in the `#call` method. For example, here, we make the access token middleware configurable. You can now say what permission level the user has to have.

```ruby
class API::Middleware::AccessToken < Coach::Middleware
  requires :access_token
  provides :user

  def call
    user = validate_access_token!(access_token)

    provide(user: user)
    next_middleware.call
  end
end

class Routes::Customers::Create < Coach::Middleware
  uses API::Middleware::AuthorizationHeader
  uses API::Middleware::AccessToken

  requires :user

  def call
    # ...
  end
end
```

A middleware can define what data it passes on, and what data it needs.

This makes your code easier to follow.

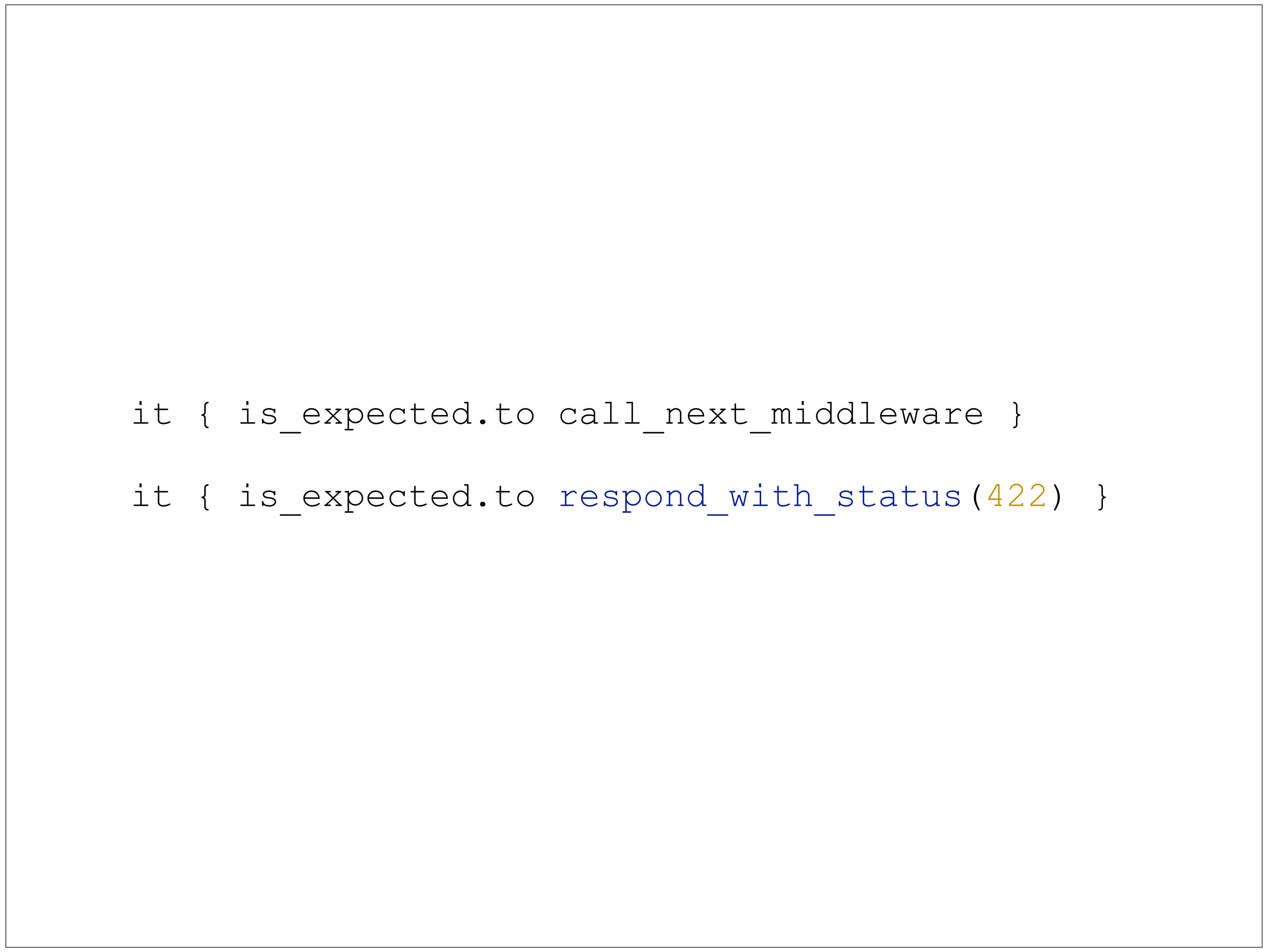
Here, for example, we say that the access token middleware is responsible for finding the user, and that the create route *needs* a user.



```
class Routes::Customers::Create < Coach::Middleware
  ~~uses API::Middleware::AuthorizationHeader~~
  ~~uses API::Middleware::AccessToken~~

  requires :user

  def call
    # ...
  end
end
```

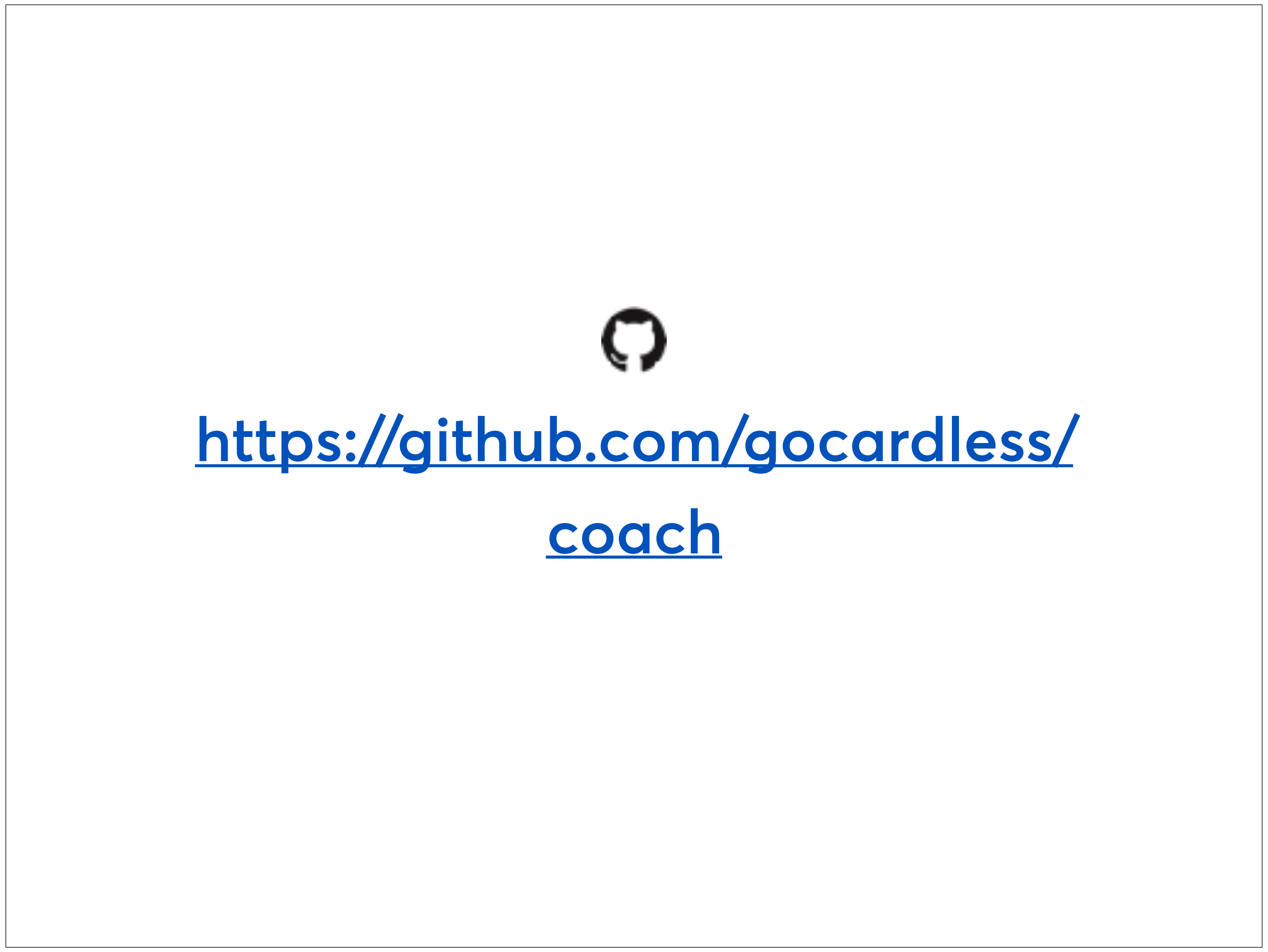
This also gives us a really cool feature, which is a bit like static analysis.

When our application boots, it can check all of our middleware, and check that their dependencies are satisfied. For example, if I declared that my route needed to have a user, but I didn't have the middleware that provided it, then the application would error at boot, alerting me to the problem.



```
it { is_expected.to call_next_middleware }

it { is_expected.to respond_with_status(422) }
```

Finally, we have built-in test helpers - this makes it really easy to unit test your middleware in an elegant way, that is declarative rather than long and repetitive



https://github.com/gocardless/coach

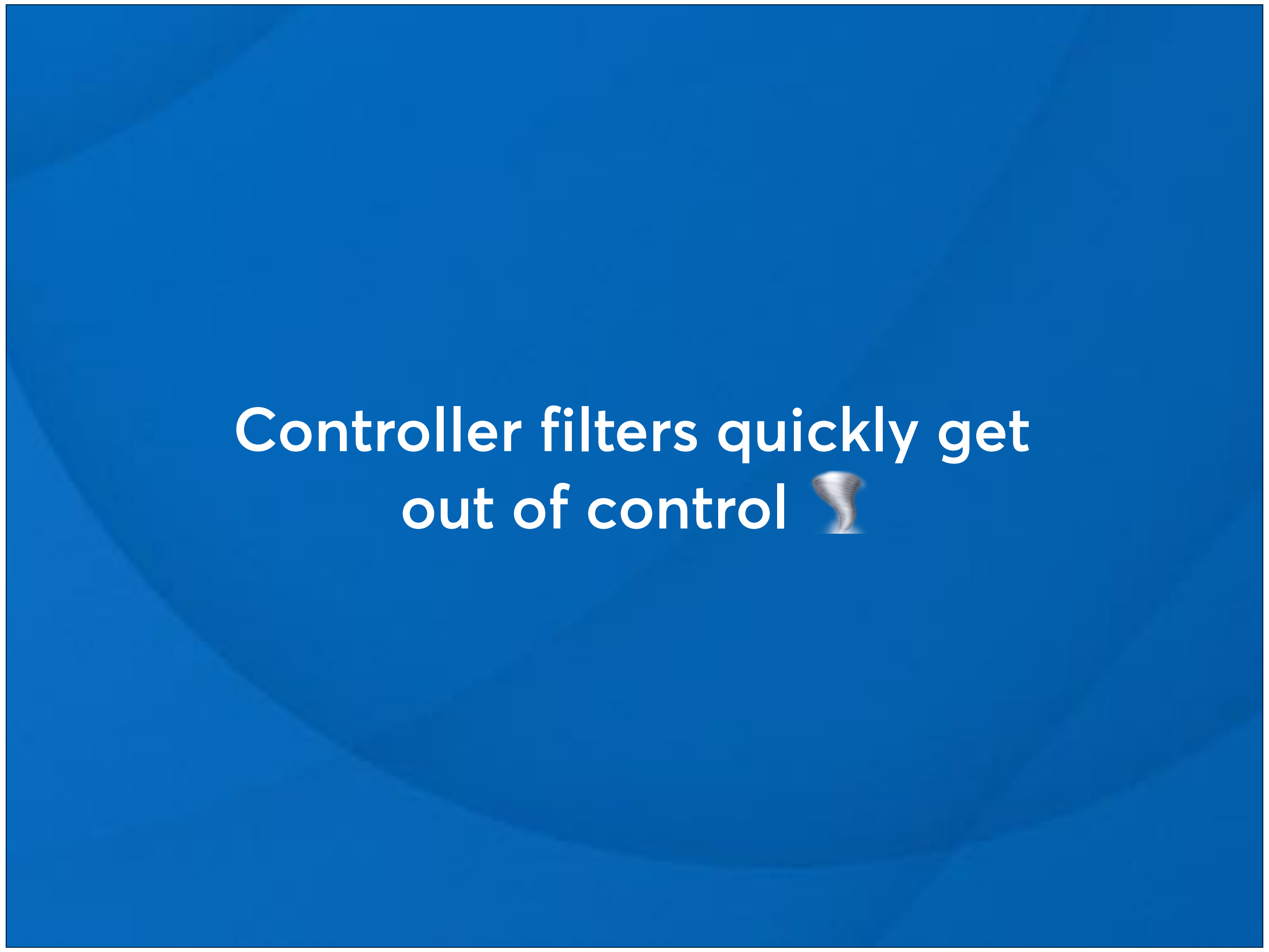
You can find Coach on GitHub at https://github.com/gocardless/coach. With it, you'll find an example application and a blog post you can follow to get to grips with it.

We want our code to be as maintainable as possible ❤️

We want our code to be as maintainable as possible

That is, we want it to be understandable, testable and reusable ✅

That is, we want it to be understandable, testable and reusable



Controller filters quickly get out of control 🌪️

Controller filters quickly get out of control. We’ve seen with a real example how they lead to unmaintainable code.

The Single Responsibility Principle points us in the right direction

The Single Responsibility Principle points us in the right direction towards simple classes that do one thing

The middleware pattern can help us build more maintainable APIs

The middleware pattern is an example of the Single Responsibility Principle in action, and it can help us build better APIs.

Using Coach, you can hit the ground running with the middleware pattern

Simple Middleware helps us to understand how things work, but with Coach, you can really hit the ground running with the middleware pattern

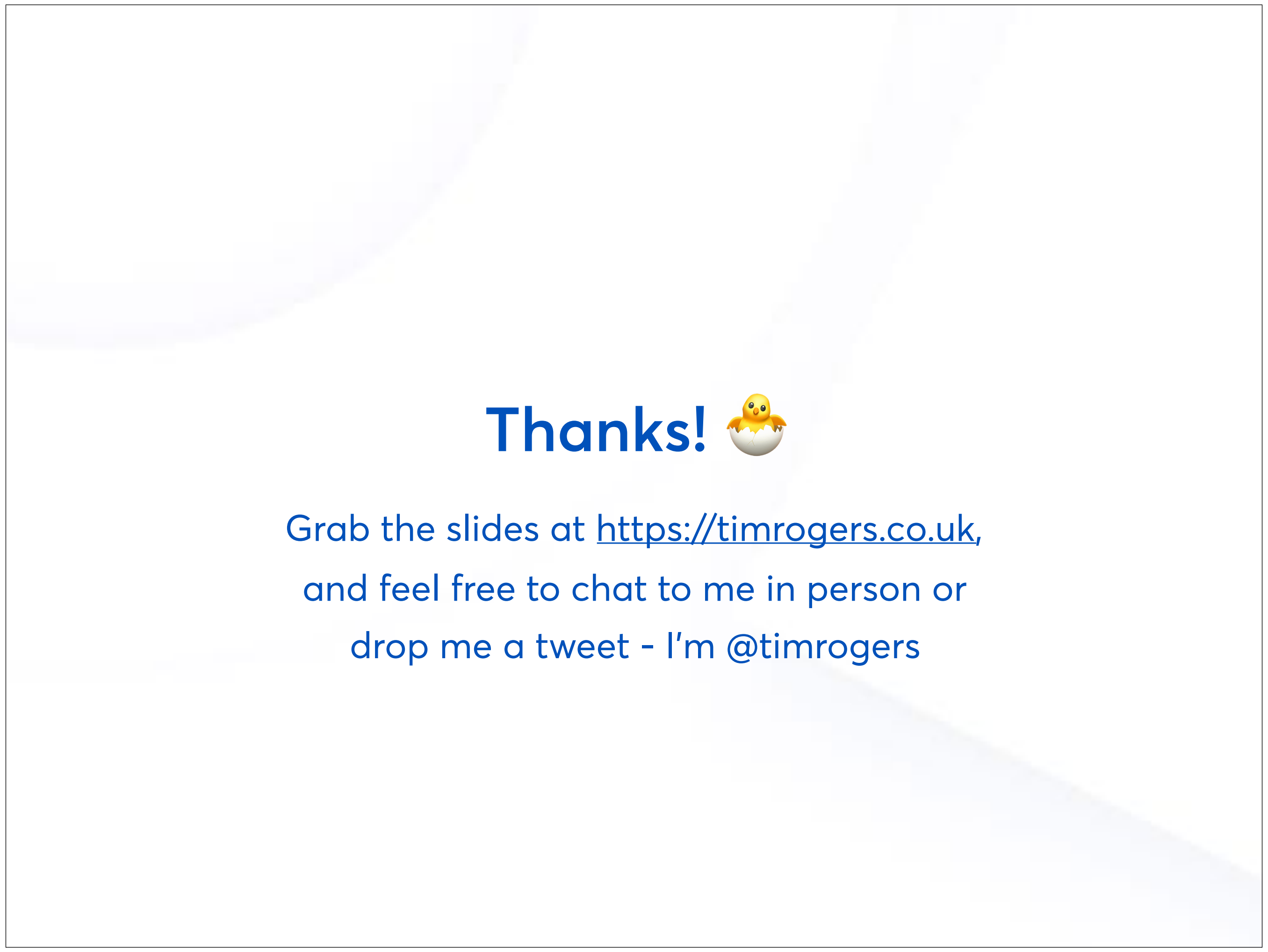
We're hiring 👨‍👩‍👧‍👦

https://gocardless.com/jobs

GOCARDLESS

If you like what you've heard today and would be interested in joining the GoCardless team in London, we're hiring! Just head to gocardless.com/jobs.

I've been in the company for years and have had the most fantastic time, so if you'd like to know more, do come and chat.



Thanks! 🐣

Grab the slides at https://timrogers.co.uk,
and feel free to chat to me in person or
drop me a tweet - I'm @timrogers

Thanks so much for your time today, and I hope this talk has been interesting and useful.

You'll be able to find all the slides for this talk on my website. Just go to timrogers.co.uk and click "Speaking".

I'd love to hear what you think and will be hanging around to chat for the rest of the weekend or you can tweet me - I'm @timrogers.