BIG PICTURE
Today you're building RailsLite.
It would probably help to have a crystal clear idea of what Rails is actually doing for you.
Rails organizes the code that generates an HTTP response based on an HTTP request.

<<< DRAW REQUEST (host / path / method) / RESPONSE CYCLE (headers / body - html, etc.) >>>

That's it! It's really not so bad.

Up to this point, you've been building apps.
Now you're going to build a framework that makes building apps easier.

WHAT DO I MEAN BY SERVER?
Something that has historically been confusing about today's topic is that the word 'server' is very overloaded.
'Server' can mean any of the following:
  1) The physical computer in a basement somewhere running your rails app
  2) The piece of software on that computer that receives HTTP requests and sends HTTP responses
  3) Your rails app itself, which is really just a ruby program running on that computer in a basement
  4) The hardware/software combination – in our request/response cycle, everything <<<OVER HERE>>>

  To avoid confusion, I'm going to try to refer
  to the first thing as "a computer in a basement",
  to the second thing as a "web server",
  and to the third thing as "your app".
  If I say "server" and you are confused, please STOP ME!
  Mid-sentence!
  I won't be insulted.

MIDDLEWARE
One of your readings was about middleware. When we say middleware, we are talking about
software that gets run in between the web server and the web framework.

<<< DRAW OUT client --> puma --> rack --> rails app --> rack --> puma --> client
>>>

One app can have many middlewares, each with one very specific purpose.
If you run 'rake middleware' in a rails app, you will see how much middleware
Rails uses by default. These middlewares run before the request even hits your
router.

Today we will work with one specific middleware, Rack. Rack papers over some
of
the implementation details of different web servers. Remember, we are writing
a tool for other developers to use. We don't want to force them to use a
particular
web server, we want to enable them to use whatever they want. One approach
would
be to bloat our framework with logic about which web server is being used.
Another
approach would be to use an intermediate technology – one whose MAIN
purpose is
to paper over the differences in web server implementation, parsing HTTP
requests
and preparing HTTP responses that will work for whichever server is in place.
As long as Rack is compatible with a particular web server, and as long as we
use Rack,
then our framework will also be compatible with that web server. This will make
our
framework flexible, modular, and only concerned with the stuff it should be
concerned with.

Rack uses "server" to mean "program capable of responding to HTTP requests".
Apologies.

We're going to do a little demo to show you how Rack works. By the end of the
demo,
we'll have built a tiny baby app.

```
require 'rack'
require 'byebug'
require 'json'

SIMPLEST VERSION – APP MUST RESPOND TO "#start" AND
RETURN AN ARRAY
CONSISTING OF:
  1) status code
  2) response headers, eg. content type
  3) response body, which must respond to #each

app = Proc.new do
  ['200', { 'Content-Type' => 'text/html'}, ['hello world']]
end

Rack::Server.start(
  app: app
)

# ==========================================

USE RACK RESPONSE INSTEAD OF AN ARRAY
app = Proc.new do
  res = Rack::Response
  res.status = '200'
  res['Content-Type'] = 'text/html'
  res.write('hello world')
  res.finish
end

Rack::Server.start(
  app: app
)
```

```
# ============================================
```

DOING STUFF WITH THE REQUEST PARAMS

```
app = Proc.new do |env|
  req = Rack::Request.new(env)
  res = Rack::Response.new
  res.status = '200'
  res['Content-Type'] = 'text/html'

  name = req["name"] ? req["name"] : "stranger"
  res.write("hello #{name}")
  res.finish
end

Rack::Server.start(
  app: app
)

# ============================================
```

DOING STUFF BASED ON THE REQUEST PATH / USING RACK REQUEST

```
app = Proc.new do |env|
  req = Rack::Request.new(env)
  res = Rack::Response.new
  res.status = '200'
  res['Content-Type'] = 'text/html'
  if req.path == "/cats"
    name = req["name"]
    res.write("#{name} says MEOW")
  elsif req.path == "/not_cats"
    res.write("why don't you care about cats are you
heartless???")
  end
  res.finish
end

Rack::Server.start(
  app: app
)

# ============================================

READING FROM AND WRITING TO COOKIES
app = Proc.new do |env|
  req = Rack::Request.new(env)
  res = Rack::Response.new
  res.status = '200'
  res['Content-Type'] = 'text/html'
  if req.path == "/cats"
    cats = req.cookies["cats"]
    res.write("ALL THE CATS: #{cats}")
  elsif req.path == "/add_cat"
    new_cat = req["name"]
    if new_cat
      if req.cookies["cats"]
        old_cats = JSON.parse(req.cookies["cats"])
      else
        old_cats = []
      end
      old_cats << new_cat
      res.set_cookie("cats", old_cats.to_json)
    end
  elsif req.path == "/not_cats"
    res.write("why don't you care about cats are you heartless???")
  end

  res.finish
```

```
end

Rack::Server.start(
  app: app
)

# ============================================
```

ADD REDIRECT AFTER ADDING A CAT

```
app = Proc.new do |env|
  req = Rack::Request.new(env)
  res = Rack::Response.new
  status = '200'
  res['Content-Type'] = 'text/html'
  if req.path == "/cats" # purely to prevent hitting debugger twice
    cats = req.cookies["cats"]
    res.write("ALL THE CATS: #{cats}")
  elsif req.path == "/add_cat"
    new_cat = req["name"]
    if new_cat
      if req.cookies["cats"]
        old_cats = JSON.parse(req.cookies["cats"])
      else
        old_cats = []
      end
      old_cats << new_cat
      res.set_cookie("cats", old_cats.to_json)
    end
    status = '302'
    res['location'] = 'http://localhost:8080/cats'
  elsif req.path == "/not_cats"
    res.write("why don't you care about cats are you heartless???")
  end
  res.status = status
  res.finish
end

Rack::Server.start(
  app: app
)
```