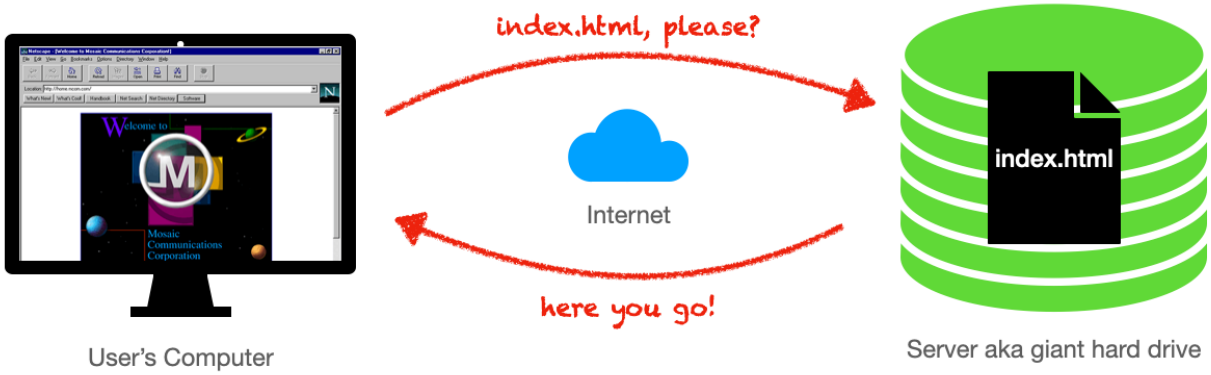
Server-Side Rendering vs. the "Modern Architecture" of SPAs

Note: Before continuing, make sure you have watched the mini "crash course" on HTML.

First, a little (recent) history lesson.

The beginning days of the web – the glorious 1990s. Most people (at least those who weren't already hard-core nerds) were just discovering the Internet. Many companies – even Fortune 500s – didn't yet have a website or an Internet presence of any kind. If you or your organization did have a website, chances are high that it was just a bunch of HTML files sitting on a shared hard disk somewhere. The story of an individual viewing a website was pretty simple, and went something like this:

- Wait for a time that nobody else was using the phone at the house, and dial into my Internet Service Provider (ISP) so I can gain access to the Internet
- Open up Netscape Navigator (the first and only web browser at the time), enter web address (there was no search 😳)
- Let the Internet do the work of reaching out to the file server where the web address pointed to, get back some HTML, and return it to the browser, which turns it into something I'm able to see



The Internet, circa 1990's

And most people were happy with this. The Internet was simply a tool for publishing – content providers were able to maintain a static set of files, in HTML format, and have those pieces of content available for end-users to view anywhere and at any time. This was perfect for company websites, libraries, news publications, and anyone else using the Internet at that time to publish information, even in large quantities – information that was written but then *hardly ever changed*.

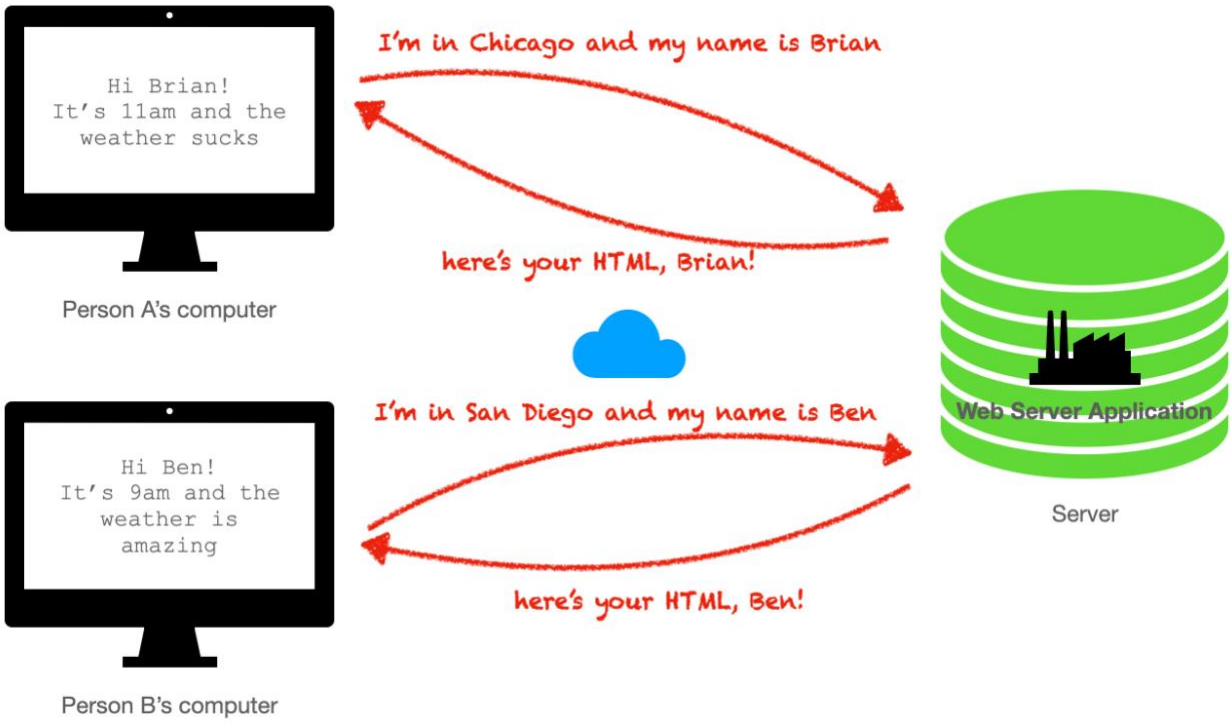
And HTML was a perfect medium for this content. Not too difficult to learn, and provided just enough functionality to allow content publishers to create articles, complete with formatting, images, and, most importantly, links to other articles on their sites or elsewhere on the web. The ability to link articles together is what transformed the Internet into an interconnected spider web (nay, *World Wide Web)* of information.

Dynamic Content

At some point, someone saw this perfect system, and decided to see if we could take it just one step further. The question was asked – *what if two different people viewing the same web page were able to NOT see exactly the same thing?* Just imagine if...

- Person A and Person B could be greeted each day by seeing their own respective names on the top of the page?
- Person A could see a different date and time on the top of the page than Person B, because they live in two different time zones?
- Person A and Person B could see some different, personalized content, like the current weather where they are?

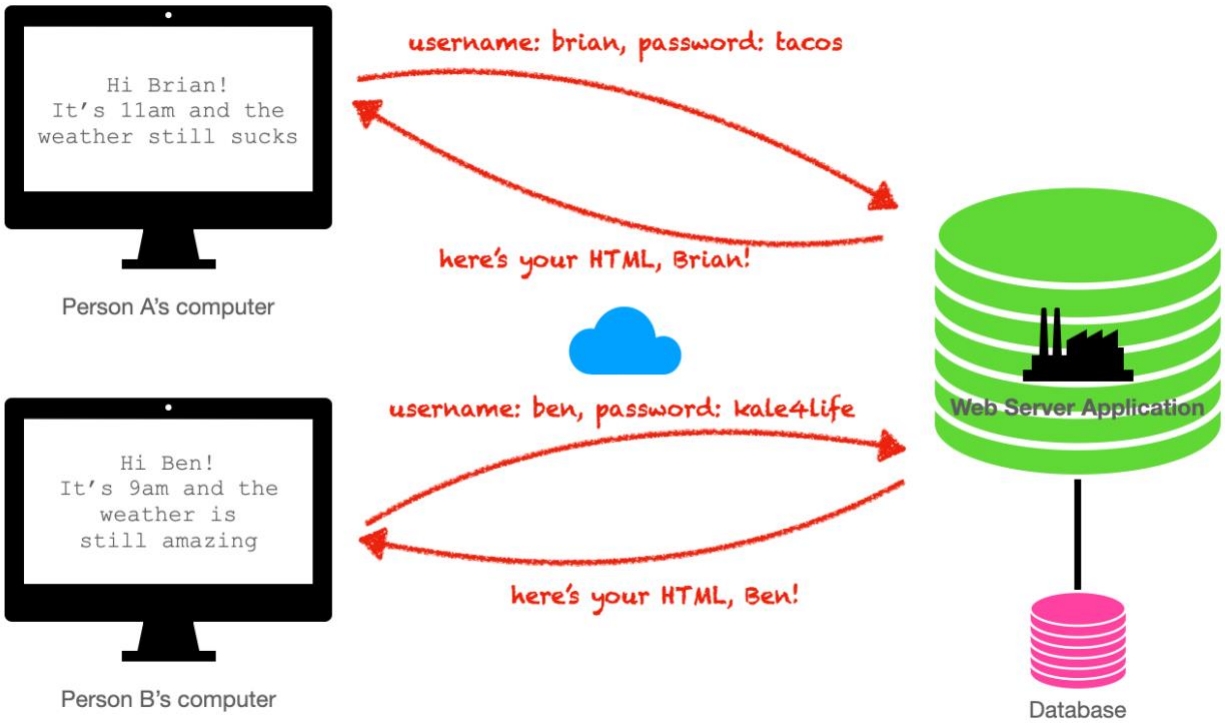
That would be amazing! How can that be done with just HTML, though? The answer: we can't – HTML is static. We would need a way for the user/web browser to send information to the server, and for the server to use that information in order to *generate different HTML for each person, based on the information sent*. The answer was to create the first *web server applications* using technologies like CGI, Perl, and PHP – software applications that did the job of interpreting the user's **request** (along with any data that came along with it) and generating a dynamic HTML **response** for every user.



Extremely accurate graphic of web application architecture from the 1990's

Data-Driven Dynamic Web Applications

Dynamically-generated web pages ended up working so well that software developers couldn't get enough – we had to take it further. Eventually, we decided that if we added a database to the web server application, we could truly take this game to the next level. We could avoid having to ask the user for information every time, and instead, we could simply have them log-in with a username and password. We could get the rest of the information we needed about the logged-in user from information we had stored in the database.



Added a database... sweet

Fast-Forward 30-ish Years

Today, web browsers may have evolved greatly (RIP Netscape), and there may be a myriad of shiny new technologies, but the end-goal remains the same: to dynamically show each user their own HTML. That's right; after all these years, HTML is still the native markup language of the web – every single website and application you use in your web browser is delivered to you via HTML.

To dive in a little deeper, there are currently three pieces of the **web browser** puzzle – three "languages" that any browser understands out-of-the-box:

- HTML (HyperText Markup Language) – a markup language used to create documents, and to describe the structure of content in those documents
- CSS (Cascading Style Sheets) – a mechanism used to add style (e.g. colors, fonts, spacing) to HTML documents
- JavaScript – a programming language – the only language that is understood by web browsers and one that is capable of manipulating HTML

In addition, we also have the **web server** application. It's still typically lives on a big computer and still does the job of delivering the HTML (and now also CSS and JavaScript) back to the web browser.

Now, the Bad News

Despite the relative simplicity of the end-goal – having a web browser ask a web server for content in a format it can understand and display to the end-user – web developers have managed to add quite a bit of complexity to this task over the years. There are now hundreds (maybe thousands) of tools and technologies that have been created to do this same job. Some reasons for this are:

- Developers have different preferences for programming languages they want to use
- Developers have different philosophies on how software should be built
- Developers have different opinions on the value of ease of learning vs. starting a project vs. long-term maintenance
- Developers have different business and technical requirements
- Developers love shiny things

The Current State of Web Development

If we tried to go out to the Internet to learn about web development today, we'd run into many, many different tutorials, blog posts, videos, and training courses. And if we spent our time sifting through a few of them, it's likely that no two would be exactly alike in terms of language, technologies, and tools used. The current state of web development is that fragmented, indeed. Not to worry, though. The vast majority of web application frameworks, without regard for subtle differences in language and tooling, can be distilled into two main types of applications, each representing a philosophy and approach to developing web apps:

- Server-side rendered (SSR) applications
- Single page applications (SPA)

Each of these has their own advantages and disadvantages, and choosing one or the other is certainly not one-size-fits-all. In fact, many organizations have chosen to employ both approaches, sometimes even within the same web application!

Let's examine what each of these approaches are and why we'd choose one over the other.

Server-Side Rendered Applications (SSR)

The 1990s dynamic web applications described at the beginning of this lesson – this is an example of server-side rendering. The story goes like this:

- The end-user visits a certain web address (URL) in their web browser

- The web server interprets the user's **request** and, based on the logic contained within the application, delivers an HTML-based **response**, perhaps talking to a database along the way
- The end-user clicks on a link, taking them to another URL, another **request-response cycle** happens, rinse and repeat – every time a new URL is requested, the web application creates a new HTML page and sends it back to the end-user's browser, which then displays the HTML page in its entirety

Single Page Applications (SPA)

SPAs represent a completely different philosophy. A very simplified version of the story might go something like this:

- The end-user visits a certain web address (URL) in their web browser
- The first time this happens, the **request-response cycle** is basically the same as with SSR applications – because this is how web browsers are designed to work – the server generates HTML and returns it to the web browser to be displayed
- When certain events happen (like the user clicking a link or a button, or simply time elapsing), there's *JavaScript code that tells the web browser to make requests to the web server for more data underneath the hood*
- The web server application responds with the data requested – it **could** be HTML, but in all likelihood it's in a more concise and data-driven format, like JSON
- There's more *JavaScript code that tells the web browser how to interpret the data and how to add to or manipulate the current HTML on the page*, so that it can be displayed to the user; often times, only a small portion of the page that would be affected by changes to data is updated, with the rest of the page remaining unchanged
- The web browser doesn't visit another URL – we simply stay on the same page, continuously updating in response to new events, hence the moniker Single Page Application

When to Use SPAs over SSR

If SPAs sound complicated in comparison to SSR applications, that's because they usually are. However, there are many scenarios where SPAs are superior to traditional, server-side rendering, both from a technical and business/organizational perspective. These don't apply in all situations, but are often good reasons to choose the SPA architecture:

- SPAs tend to *feel* more performant and real-time, as they don't need to perform a full page refresh anytime data changes; instead, only small portions of a full page that need to change are updated. As a result, applications with more complex

user interfaces tend to *feel* more like speedy and responsive desktop applications rather than "websites".
- Web applications that have large amounts of HTML markup, but relatively few dynamic, data-driven components, benefit from not having to re-download the full HTML of the page with every request; rather only small "payloads" of data are communicated to and from the web server and this, again, makes larger applications *feel* more snappy, especially on slower Internet connections.
- You and your development team(s) have made existing investments in, or otherwise really love, JavaScript. As mentioned earlier, JavaScript is the only programming language web browsers understand, so code that grabs data behind the scenes and uses it to update existing HTML markup – can only be written with JavaScript.
- Applications that generate server-side rendered HTML tend to be part of large, self-contained applications that include the application, the database, and all the other tools necessary to generate the HTML. By nature, these often require high-powered servers, a powerful, centralized database, and large amounts of disk space. Because SPAs don't ask for the complete HTML of every page, and instead only need access to small payloads of data, the applications that provide this data can potentially be split into many smaller and more manageable applications (i.e. "microservices") – ones that can be run on much lower cost infrastructure, such as those provided by Amazon Web Services (AWS).
- Along those same lines, larger organizations employing a microservices-based strategy are able to split the back-end data work into the hands of many, smaller teams. This lends itself to easier-to-manage teams and organizational structure, with fewer conflicts than with a single, giant team managing a huge, monolithic web server application.

When SSR > SPA

Now that we've discussed some compelling reasons for SPAs, it's time to talk about SSR applications and when it's appropriate to choose a server-rendered tech stack. We'll do so, by directly selling against the pros of SPAs listed above:

- Most applications, especially those being built by beginners, are probably not complex enough from a user interface standpoint to see a noticeable performance advantage from using SPAs; in fact, delivering entire HTML pages over-the-wire is really not that slow, particularly as very high-speed Internet speeds continue to be the trend for end-users.
- Many line-of-business applications, which represent a huge number of applications being built, don't need to be super-performant from the user's standpoint; it's ok that the screen flickers for 1 millisecond before seeing a new HTML page rendered.
- Building SPAs is often much more complex than building a SSR application, especially for novice programmers. The intricacies of sending and receiving data

from the back-end and manipulating the various components of the page in response can be hard to manage – so much that almost no developer does it without some sort of front-end framework. If you've heard of tools like React, Angular, Vue, jQuery – these are all tools developers use to manage this process.
- Since the majority of the work with SSR applications involve generating HTML on the server, any language – not just JavaScript – can be used. If an organization/development team has made existing investments – or simply prefers – languages like Python, Ruby, C#, Go, or others, there are widely supported web development frameworks available for those languages.
- Your company is probably not big enough (very few are) to take advantage of splitting business logic and data into a series of microservices. In fact, you'll probably find that managing everything in a centralized application and database much easier to understand and manage. And cost for server hosting for a startup or other small application is nominal.
- Your team is small. In fact, it's probably just you. Creating many small applications instead of one big one is just unnecessary at this point.

As you may have guessed by now, server-side rendered applications are what we are going to be building in this course.

It might surprise you that many people nowadays do choose the SPA path, despite the added complexity and much steeper learning curve. Even most of today's beginner-focused coding classes will teach React or some other SPA-focused framework. But this popularity has more to do with the current state of the web development economy and job market than it has to do with SPAs being the better tool for the job. That job being, within the context of this course, to get a product built in the easiest and shortest path possible, gaining the technical literacy you need to make your own decisions along the way.

Server-Side Rendering Using Ruby on Rails

Time to get to the nuts and bolts of a server-side rendered application. As we've discussed, the high-level story goes something like this:

- An end-user visits your application (i.e. they go to `https://WhateverYourDomainNameIs.com/SomePath`)
- That **request** is received by your web application, and it figures out what the appropriate **response** is (i.e. what HTML to send)
- The response is sent back over the wire and is received by the web browser, which then shows it to the end-user

Let's take each one of these steps and peel back one layer of the technical onion, so that we can take a deeper look.

URLs, Host Names, and Paths, oh my!

As an end-user of the web, you see this every day, and probably don't even think anything of it. But that address that sits in your web browser's address bar? That's the URL (Uniform Resource Locator) of the content you're looking at. The "locator" part of URL means that, in theory, every unique URL should produce a unique result, and the URL is simply a means for providing the location of that result. For example (again, in theory), these two different URLs should produce a page for books and a page for shoes, respectively:

```
https://YourDomainName.com/books
```

```
https://YourDomainName.com/shoes
```

Furthermore, *books* is the **resource** you'd expect to find at the first URL (in other words, you'd expect to find a page about books), and *shoes* is the **resource** you'd expect to find at the second URL. **Uniform Resource Locator**.

And, the *uniform* part refers to the fact that they all look the same:

- The **Protocol** – *https* or *http*
- Followed by the "colon-slash-slash"
- Followed by the **Host Name** – *YourDomainName.com*
- Followed by the **Path** – */books* or */shoes*

For the purposes of this lesson, we're going to concentrate on the **path**, much more than the other pieces. The protocol is always going to be *https* and the domain or host name is going to be given to us by Ona while we're developing our application. We'll learn about domain names and how to use them in a later lesson. *For this reason, we'll refer only to the path (e.g. `/books`) instead of the full URL from now on.*

It turns out that Rails is tailor-made to help us write these *resource-based* applications. In fact, **Rails assumes that the main thing you want to do is build an application that is a collection of things to CRUD**. *This is super-important, so let's make a mental note of that for later* – but for now, let's see how we can use Rails to create our first server-rendered HTML page!

Starting Fresh

Let's spin up a new Rails application on Ona so we can get coding. Head over to https://github.com/entr451-winter2026/web-apps and "Use this template" to create a new repository in our personal GitHub account – call it `my-first-app`. Then, open the code in Ona.

We're going to see that there are a lot of folders and files. Don't be intimidated – we're not going to use them all, and in fact, if we know exactly where things go, it's really quite straightforward.

The first thing to do is decide what our first resource will be. What will be the subject of our first page? Naturally, it will be a page about tacos.

Our first step is to tell our Rails application that we want to build a page about *tacos* – that is, that our app contains a **resource** named *tacos*. We do this using the *routes file* – which is located at `config/routes.rb`. Let's open it up and have a look; ignoring the comments, the file looks like this:

```
Rails.application.routes.draw do




end
```

To let Rails know about our *tacos* resource, we add a single line to the file:

```
Rails.application.routes.draw do

  resources "tacos"

end
```

And we're done with that. Our application is now configured to have a `/tacos` path.

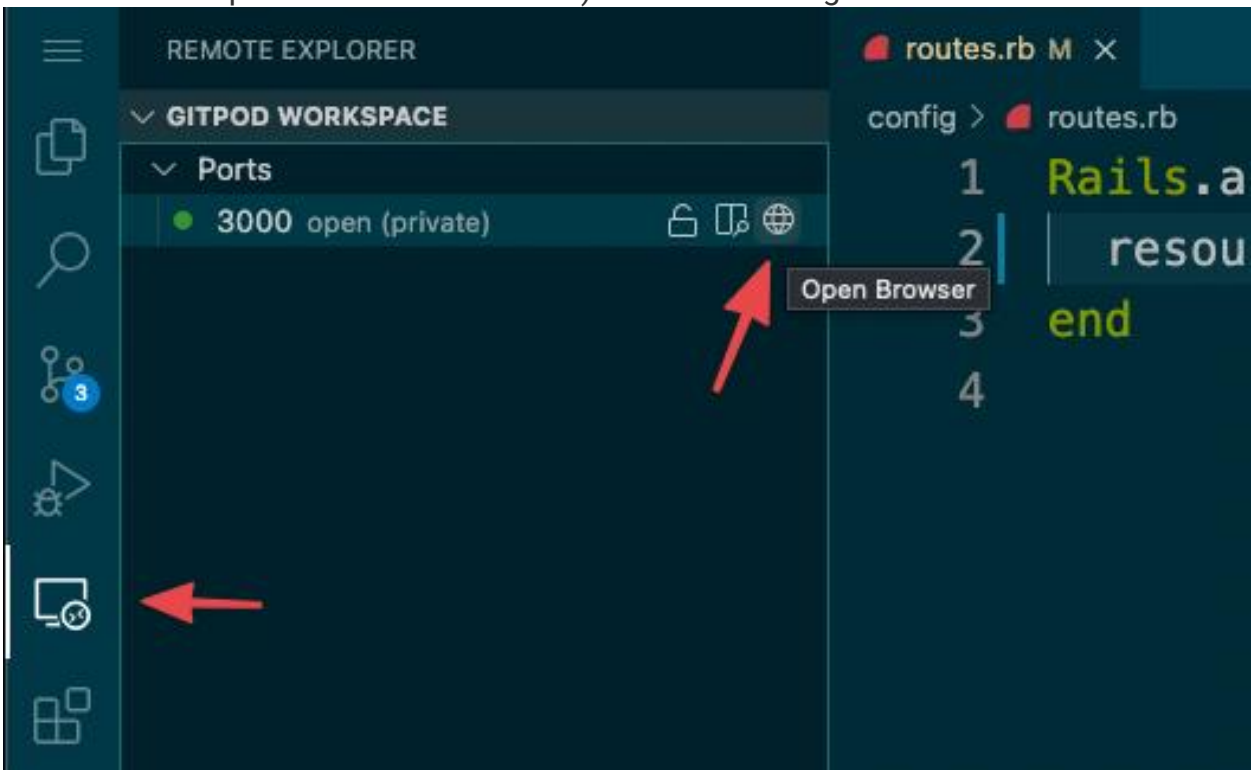
Let's fire up our Rails application and see what we've got! Let's hop into the Terminal (*View -> Terminal* in Ona, if it's not already open) and issue the following command:

```
rails server
```

What does that do? It executes the software application needed to serve up HTML pages from our Rails application! We should see a bit of terminal output, letting us know that the server application is now running, and to hit `CTRL-C` if we want to terminate the application and return to the normal Linux command prompt.

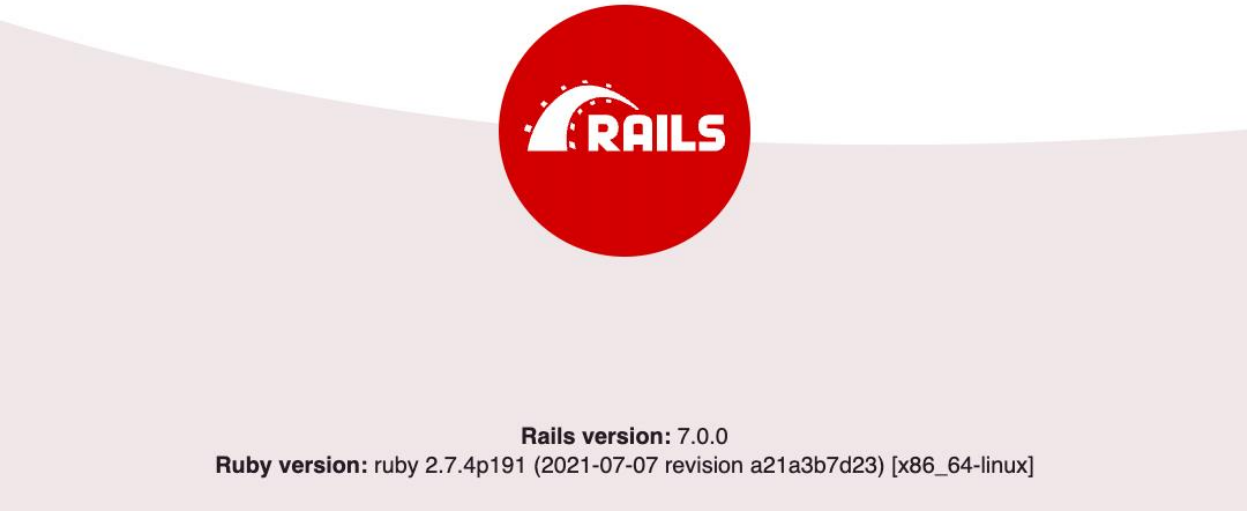
We're also going to a see a small window pop up in the corner, telling us that a service is available on port 3000. We can hit "Open Browser" from here to open a new tab containing the results of our application. If that pop-up goes away too quickly, or if it

doesn't appear, we can do the same thing by clicking on the "Remote Explorer" (the fifth icon from the top in the left-hand sidebar) and then clicking the "Globe" icon.



Accessing the "Open Browser" from Remote Explorer

From there, if everything went smoothly, we should see a fairly generic-looking page letting us know that our Rails application is running:



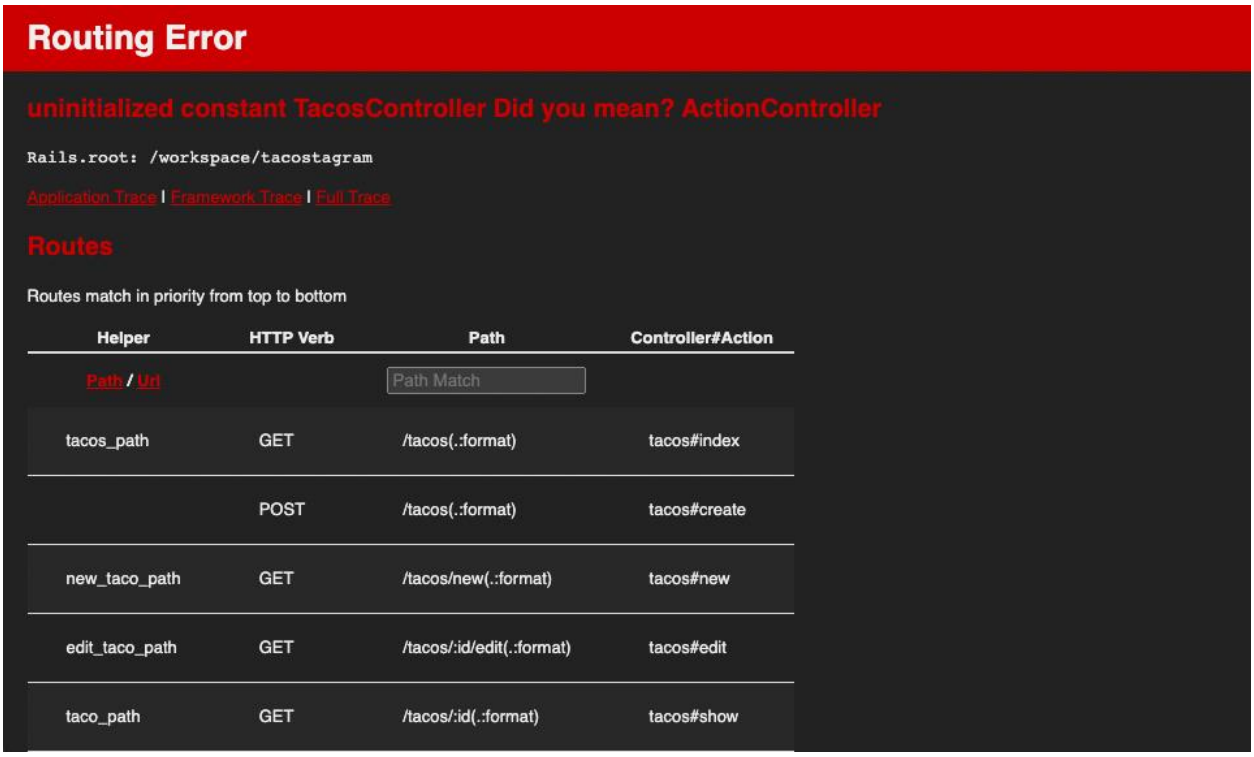
All is good in Rails-land.

We should also notice that the web address for that page is rather long – that is the temporary domain name that Ona has issued us, for the purposes of developing our application.

Ok – enough with the setup – let's get to that page about tacos!

Don't Fear the Error

Let's go to `/tacos` to see the results of adding our tacos resource to our routes file! Uh-oh, looks like that produces an error:



As we can see, error messages in Rails are quite loud and scary. But *don't be afraid of errors.* Error messages are there to help us. As a matter of fact, *error messages often tell us exactly what we need to do to fix problems in our code* – we just have to stop and read them.

In this particular case, the primary error is `uninitialized constant TacosController`. Since this our first time through the Rails universe, we didn't know that the next step needed, after creating our resource in our routes file, is to create a *controller* for that resource. But, the error message tells us that's exactly what we need to do. From our Terminal prompt, let's hit `CTRL-C` to stop the server application, then issue the following command:

```
rails generate controller tacos
```

That's going to see some output in the Terminal:

```
create  app/controllers/tacos_controller.rb

invoke  erb

create    app/views/tacos
```

That's just letting us know that a couple of files/folders were created. We'll get into the controller itself in a future lesson, but for now, our controller is created and we can move on. Start up our server again with `rails server` – and refresh the browser tab containing our application.

Another error! But we're not afraid. This one is a bit cryptic, and will make more sense later, but essentially this error is telling us that we need to have some output for this *tacos* resource, but we haven't created any yet. Fortunately, doing so is easy! In the File Explorer in the left-hand sidebar, find the `app/views/tacos` folder and create a file called `index.html.erb`. Of course, we could also use the command prompt to do so, with `touch app/views/tacos/index.html.erb`. Now, refresh the page again.

Such emptiness! But this is great – because there are no more errors! In fact, the only thing left to do is to fill the `index.html.erb` file up with HTML. Open up the file and plug some in:

```
<h1>I love tacos</h1>

<p>There are so many kinds of tacos:</p>

<ul>

  <li>Carnitas</li>

  <li>Al Pastor</li>
```

```
  <li>Steak</li>

  <li>Fish</li>

</ul>
```

Refresh our page again, et voilà!

Something to Notice – The Application Layout

Let's view the page source of the page we just created at the `/tacos` path. We can do that by right-clicking anywhere on the page and selecting *View Page Source* (in Chrome). We'll find that the generated HTML page looks something like this:

```
<!DOCTYPE html>

<html>

  <head>

    <title>StarterApp</title>

    <meta name="viewport" content="width=device-width,initial-scale=1">

    <meta name="csrf-param" content="authenticity_token" />

    <meta name="csrf-token" content="..." />

    <link rel="stylesheet" href="/stylesheets/application.css" />

  </head>



  <body>

    <h1>I love tacos</h1>
```

```
    <p>There are so many kinds of tacos:</p>

    <ul>

      <li>Carnitas</li>

      <li>Al Pastor</li>

      <li>Steak</li>

      <li>Fish</li>

    </ul>

  </body>

</html>
```

Whoa! Where did all that HTML come from? We only typed the stuff inside the `<body>` element – where did the other stuff come from?

By default in Rails, we only have to put the stuff that goes inside the `body` element into our view files (everything in `app/views`). The outer shell of the HTML actually comes from the `app/views/layouts/application.html.erb` file – the *application layout*. Using the same common layout, or outer wrapper HTML like the `DOCTYPE`, `html` and `head` elements, is such a common task in web development that Rails ships with it out-of-the-box.

Of course, that was still a pretty long path to getting to essentially the same result as a simple HTML page. But we've now done so in a way that all the pieces of the infrastructure puzzle are set up – we are now ready to harness the power of combining the technologies we've learned about so far, which is what we'll do in the next lesson.

Two Great Things

If HTML is our chocolate, and Ruby is our peanut butter, it's now time to combine them together to make something even better. (Yes, that's stolen directly from a Reese's commercial).

index.html.erb

That's a weird file extension, isn't it? Normally, files only have one extension like `.txt` for text files, `.docx` for Microsoft Word files, `.pdf` for PDFs, and so on. This file that we've created has two extensions – `.html` and `.erb`. That's right – we're combining two things together – HTML and Ruby (ERB stands for Embedded Ruby).

Let's dive right in to see how this works. Edit the `index.html.erb` and replace it with the following:

```
<% fillings = ["Carnitas", "Al Pastor", "Steak", "Fish"] %>

<h1>I love tacos</h1>

<p>There are so many kinds of tacos:</p>

<ul>

  <% for filling in fillings %>

    <li><%= filling %></li>

  <% end %>

</ul>

<p>Currently, there are <%= fillings.size %> kinds of tacos.</p>
```

There's a bit of weird syntax going on here, and it can take some getting used to, but this is a good example of using Rails' ability to combine HTML and Ruby together. Let's break it down:

- If we view the resulting page source (i.e. *View Page Source* in Chrome), we can see that the result is HTML – as we've previously learned, **the web browser does not understand anything other than HTML.**
- Any time we want to write Ruby inside our HTML file, we use the special `<%` and `%>` symbols in our code. That is, anything inside `<%` and `%>` – otherwise known as *ERB tags* – will be interpreted as Ruby instead of HTML. In this case, instead of having our fillings (carnitas, al pastor, steak, and fish) hard-coded into our HTML `li` elements, we've moved them into a Ruby array. Now, when we want to add more fillings, we simply modify the array.

- We can see that there are ERB tags with and without the equals sign – `<%` and `%>` and `<%=` and `%>`. **Any Ruby code inside the version with the equals sign will be written to the resulting HTML.**

Let's examine that last point further, because it's that important to understand. We can see that there are times when we simply want to execute Ruby code, for example, setting the value of a variable:

```
<% fillings = ["Carnitas", "Al Pastor", "Steak", "Fish"] %>
```

In this case, we want to set the value of the `fillings` variable to the Array of Strings we've specified. We don't want anything to be displayed to the end-user as a result (i.e. we don't need anything written out to the resulting HTML).

Likewise, our for-loop is just Ruby code we want executed – we **don't** actually want anything to be written to the resulting HTML for the `<% for... in %>` and `<% end %>` lines of code.

What we **do** want to be written to our resulting HTML is the value of each filling (e.g. "carnitas"), in between a set of start and end `<li>` tags, i.e.:

```
<li><%= filling %></li>
```

The `<li>` and `</li>` are HTML, and `filling` is Ruby. And we want the value of `filling` to be written out to our HTML, so we can produce a list of taco fillings – so we must use the version of the ERB tags with the equals sign.

Similarly, the last line that prints the total number of fillings, intermingled with HTML:

```
<p>Currently, there are <%= fillings.size %> kinds of tacos.</p>
```

Once again, we have the printing, equals sign version of the ERB tags, because we want the value to appear in the resulting HTML.

Roll the Dice

Time to use this newfound knowledge to build a simple dice game! Let's review the steps for adding new functionality to our app:

- Decide what *resource* the functionality represents
- Add the resource to the *routes file*
- Create the controller
- Write the view using HTML and embedded Ruby (ERB)

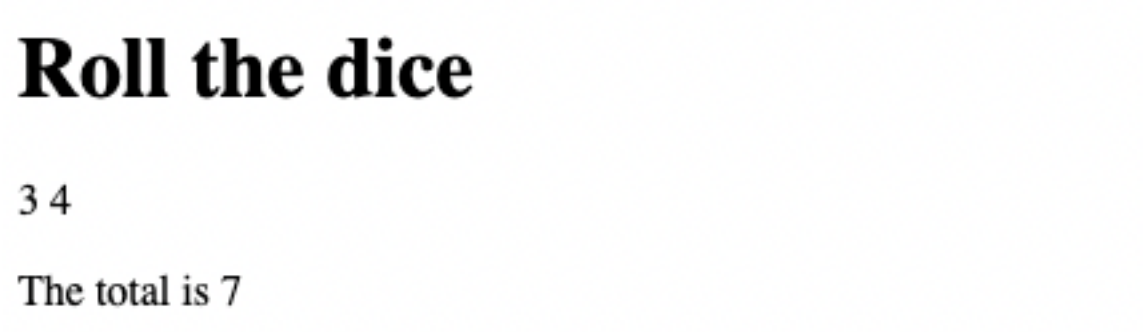
We'll repeat the steps that we used to create the *tacos* resource, a little faster this time.

- What word would be best for describing this resource? How about *dice?*
- Add `resources "dice"` to our *routes file at* `config/routes.rb`
- Generate a controller by doing `rails generate controller dice` at the Terminal prompt, stopping and starting the server as necessary
- Create an empty file at `app/views/dice/index.html.erb` and get ready to build our code

We'll start with some basic HTML to represent the value of our two dice.
In `app/views/dice/index.html.erb`:

```
<h1>Roll the dice</h1>

<p>

  3

  4

</p>

<p>

  The total is 7

</p>
```

Making sure our `rails server` is running, head over to `/dice` in the browser and see the result. If everything went well, we should see something similar to:



If we hit refresh on the browser a few times, we'll see that, unsurprisingly, the result of the roll of our two dice is 3 and 4, every time. We'd win a lot of money in Vegas!

As exciting as that would be, that's not a very complete game we've built. Instead of 3, 4, and 7, those values should probably be dynamically and randomly generated. Let's write the Ruby code to generate three variables to represent the value of the first die, second die, and total. How would we do this in Ruby? Maybe something like:

```
die1 = rand(1..6)

die2 = rand(1..6)

total = die1 + die2
```

(The `rand` function is built into Ruby – here, we're asking for a random number between 1 and 6). Now, we have Ruby code that creates the three variables that we can use in our HTML, and it's simply a matter of wrapping that code in ERB tags and using the results in the right places:

```
<%
  die1 = rand(1..6)
  die2 = rand(1..6)
  total = die1 + die2
%>
<h1>Roll the dice</h1>
<p>
  <%= die1 %>
  <%= die2 %>
</p>
<p>
  The total is <%= total %>
</p>
```

app/views/dice/index.html.erb

Note that we could have written the first three lines using ERB tags on each line, like this:

```
<% die1 = rand(1..6) %>

<% die2 = rand(1..6) %>

<% total = die1 + die2 %>
```

But, the way we've done it saves us some typing. Now, if we head back to our finished product in the web browser, and hit refresh a few times... success! We get the values of two random dice displayed, along with a total, with each refresh of the browser.

To make a finer point of this, and to make sure we connect this to what we've learned before, every refresh of the browser initiates a new **request** to `/dice`, our application code in the *routes file* and `index.html.erb` does its job, which is to return the appropriate **response** (by crunching the code we've given it in order to generate HTML) to the web browser.

Just Hit Refresh?

It's a perfectly fine development experience, but asking our end-user to hit the *Refresh* button in the web browser is probably not the best idea for acquiring and retaining end-users, so let's make it better. A simple link that reads *Roll Again* should do the trick.

We know that we can create a link in HTML by using the `<a>` or *anchor element* – but where should it go, i.e. what should the `href` be? In this case, a link back to the resource we're already on should suffice:

```
<a href="/dice">Roll Again</a>
```

Wrap that in a new paragraph, and our code (so far) looks like:

```
<%

  die1 = rand(1..6)

  die2 = rand(1..6)

  total = die1 + die2

%>

<h1>Roll the dice</h1>

<p>

  <%= die1 %>
```

```
  <%= die2 %>

</p>

<p>

  The total is <%= total %>

</p>

<p>

  <a href="/dice">Roll Again</a>

</p>
```

Great – now the end-user doesn't have to hit refresh and won't be scared away by our app!

Making it Complete

No dice game would be complete without... images of dice. So let's make that happen! The actual game images have already been provided – if we look in the `public/images` directory, we can see them there.

Note: a file in the `public` directory can be accessed on the web via its file path, without the `public` part. For example, the file located at `public/images/dice/1.svg` is available on the web at `/images/dice/1.svg`.

Currently, these two lines of code...

```
<%= die1 %>

<%= die2 %>
```

... do the job of simply spitting out the two random numbers contained within the `die1` and `die2` variables, respectively. How do we replace this with images of dice? With the HTML `img` tag, of course! Let's start by replacing those two lines with hard-coded images of the 3 and 4 dice:

```
<img src="/images/dice/3.svg">


<img src="/images/dice/4.svg">
```

Hitting refresh in the browser (or using the *Roll Again* button) shows us that we're now displaying images of dice, but they're back to being the hard-coded values of 3 and 4, as expected. Of course, we don't want that, so let's replace just the part of the code that needs to replaced with the dynamically-generated values, to get our finished product:

```
<img src="/images/dice/<%= die1 %>.svg">


<img src="/images/dice/<%= die2 %>.svg">
```

One more refresh of the page, and we see our finished dice game, in all its glory. Here's the complete code, for reference and comparison, in case you got lost for any reason along the way:

```
<%
  die1 = rand(1..6)
  die2 = rand(1..6)
  total = die1 + die2
%>
<h1>Roll the dice</h1>
<p>
  <img src="/images/dice/<%= die1 %>.svg">
  <img src="/images/dice/<%= die2 %>.svg">
</p>
<p>
  The total is <%= total %>
</p>
<p>
  <a href="/dice">Roll Again</a>
</p>
```

app/views/dice/index.html.erb

Lab

Sticking with casino games, let's create a game that deals a five-card hand of poker.

Challenges:

- (First try) Follow the same pattern as we used for dice – write it this way first, then think about...
- (Refactor) How do we avoid a duplication of cards?
- (Refactor again) What if the business requirements changed from 5 cards to 7 cards? How about 100 cards?

Data in Views

Let's recap the major things we've done up to this point in the course

- We've learned to use the basic tools, like Git, the Linux command-line interface, and a code editor environment
- We've learned to query and manipulate data using SQL
- We've learned the basics of the Ruby programming language
- We've learned how to use Ruby to query and manipulate data, instead of using SQL
- We've created a basic web app that dynamically changes content with each page load

Now it's time to put all of these tools to work at the same time, and create a web application that talks to the data in our database!

Begin by creating a `Company` and `Contacts` model and corresponding database tables:

```
rails generate model Company
```

```
rails generate model Contact
```

That will create migration files; in them:

```
class CreateCompanies < ActiveRecord::Migration[7.0]

  def change

    create_table :companies do |t|

      t.string "name"

      t.string "city"

      t.string "state"

      t.timestamps

    end
```

```
  end

end

class CreateContacts < ActiveRecord::Migration[7.0]

  def change

    create_table :contacts do |t|

      t.string "first_name"

      t.string "last_name"

      t.string "email"

      t.string "phone_number"

      t.integer "company_id"

      t.timestamps

    end

  end

end
```

Then:

```
rails db:migrate
```

Next, if you look at `scripts/create_data.rb` you'll find a simple script to create three companies and four contacts. Run the script by executing:

```
rails runner scripts/create_data.rb
```

Next, configure a resource for companies...

```
Rails.application.routes.draw do

  resources "tacos"

  resources "companies"

end
```

...and create a controller for companies. Note that the controller name is pluralized.

```
rails generate controller companies
```

Finally, create a view for companies:

```
<% companies = Company.all %>
<h1>Companies</h1>
<ul>
  <% for company in companies %>
    <li><%= company["name"] %> - <%= company["city"] %>, <%=
company["state"] %></li>
  <% end %>
</ul>
```

app/views/companies/index.html.erb

Try it!

Create a resource, controller, and "index" view for contacts, showing all contacts with contact information and the name of the company for each contact.