The MVC Design Pattern

Let's talk about *design patterns*. The term sounds so cool and computer-sciencey and can be used to impress your friends and colleagues at parties. But all it really means to *implement a design pattern* is to write some code in a way that lots of other software developers have done before, in order to solve a common problem. Do a Google search for "software design pattern", and we'll find dozens of patterns that pro developers use every day.

Shockingly! – we are not the first software developers on Earth to be writing a dynamic web application backed by a database. In fact, it's one of the more common types of applications that web developers want to build. And there is a design pattern that's been used time and time again, by millions of developers, in order to do so – the Model-View-Controller (MVC) design pattern.

The MVC pattern, in its most basic form, says that we should separate our application into three layers, with the code for each layer in their own files and folders. These layers are:

- **Models** – code that talks directly to our database
- **Views** – code that creates what the user sees in the browser, like HTML and its cousin ERB
- **Controllers** – code that connects and controls traffic between the models and views

Of course, we've already been introduced to models and views in previous lessons. And we've also seen that Ruby on Rails implements the MVC design pattern right out of the box, with the separate `app/models`, `app/views`, and `app/controllers` directories that come with Rails by default.

So what is the *controller* and how does it fit into the equation? After all, we've created working applications already, without the controller – what is it and why do we need it? Let's just dive right in and implement one, then we'll discuss why the controller is important – it's actually quite straightforward. Let's begin with our *dice* resource from last time:

```
<%
  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

Let's make just a few subtle changes, including moving those first three lines of Ruby out of the view and into the `DiceController`, which was automatically created when we did `rails generate controller dice` earlier.

```
class DiceController < ApplicationController

  def index
    @die1 = rand(1..6)
    @die2 = rand(1..6)
    @total = @die1 + @die2
  end

end
```
app/controllers/dice_controller.rb

```
<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

There are quite a few new concepts happening here – let's just take it one at a time!

- We've opened up the source code for our `DiceController`, which is a Ruby **class**. Classes in Ruby are used to encapsulate functionality, as we've previously seen with *model classes.* Controller classes do a different job than model class though – while model classes are used to communicate with the database, controller classes are used as the glue between views and the rest of our code.
- Each **method** of the controller class, i.e. the code that begins with `def` and concludes with `end`, is known as an **action**. We'll notice that the name of each action corresponds to the name of each view. In this example, we have a view called `index.html.erb` and an action method called `index`.
- We've moved the code that previously lived at the top of our `index.html.erb` view into this new controller/action – but we need to use variables like `die1` and `die2` that are shared between the two files. In order to

share variables between the controller and the view, the variable must be prefixed with the `@` symbol. So we've renamed `die1` and `die2` to `@die1` and `@die2`.

But why do this? Why not leave well enough alone? It seems like we're adding more complexity to do the same thing we did before! Seems like *MV* was just fine; no need for *MVC*. These are valid points – but there are benefits to fully implementing the MVC pattern, even in this tiny example:

- Our view is now free and clean of extraneous Ruby code. All of the application logic needed to supply the view with data is now in the controller, and there are only tiny sprinkles of ERB – only where absolutely needed to dynamically embed data in our HTML.
- Coupling our Ruby-based logic with the view just gets plain ugly as our applications become more complex. It's only three lines today, but imagine 100 lines of code at the top of our ERB file.
- Web applications may need to produce other types of views. Imagine a time when our application might produce other types of output, like a JSON-based API, a PDF, or an Excel spreadsheet. Or perhaps we want different HTML views for mobile and desktop, each sharing the same logic and data. Moving the code that generates this data to the controller layer allows for this flexibility.

Browser State, Query Strings and the "params" Hash

Fun fact: *HTTP is a stateless protocol.* Bust that one out at the next party you attend, and see if anyone is impressed. (If anyone is, you might be at the right party!) In all seriousness, this is an important one – in fact, *statelessness* is a concept that is the basis for all things web development. Let's break it down.

We've already seen how web applications are delivered to the end-user via a **request-response** cycle; that is, the web browser sends a request to the web server, and the web server responds with HTML (and perhaps CSS/JS). What we haven't talked about yet is the fact that *each of the request-response cycles performed throughout the end-user's time using our application are completely independent from each other*. Consider the journey of an end-user using a e-commerce bookstore application we've built:

1. The user visits `/books` to browse a list of books for sale
2. The user clicks on a a particular title, which takes them to `/books/123` where the details of the book are displayed
3. The user clicks on the "My Orders" tab, which directs them to `/orders` where a list of their orders is shown
4. The user clicks on one of their open orders, which goes to `/orders/42` where the details and status of order #42 can be viewed

Each step of this journey involves a separate request to the web server and subsequent response. However, request #2 doesn't know that we came from request #1, request #3 has no idea about request #2, and so on. Each of these request-response cycles happen independently of each other, and the web application doesn't really even know (or care) about the order in which they happened. By the time the end-user is looking at a past order, the application has no idea that their trip originated at browsing through the book catalog.

Said another way – the web browser doesn't maintain any *state* between requests. *HTTP is a stateless protocol.*

Well, that sucks.

But we *need* our web application to be able to maintain state, right? That's the user experience we've come to expect from web applications. For example, a typical web application:

- Knows who is using the application (i.e. logged-in)
- Knows what the user did earlier (for example, added something to a shopping cart)
- Is perfectly capable of displaying dynamic content based on the user's journey (for example, showing a "thank you" message if the user places an order)

So how does this happen if HTTP is stateless and doesn't know what the user did before or the path they have taken? The answer: *we have to trick the browser into thinking it does indeed have state.* There are a couple of main ways that web developers do this today – in this lesson, we'll start with the most basic way and work our way forward in a future lesson.

The Query String

Using a *query string* is a very popular way of passing data to a request. To see how it works, let's try visiting the following URL:

https://google.com/maps?q=paris

We'll see that this takes us to Google Maps, with Paris, France at the center of the map. Now, let's go to a similar, but slightly different URL:

https://google.com/maps?q=las+vegas

We'll see that it's still Google Maps, but puts us in the heart of Las Vegas instead. If we look carefully at these two URLs, we'll notice that the protocol (https), host/domain name (google.com), and resource/path (/maps) are identical. The part that's different is the part after the `?q=` – either `paris` or `las+vegas`. The part of the URL after the question mark `?` is known as the *query string.* Let's have one more example:

https://google.com/maps?q=paris&hl=fr

Here we see a query string with two pieces of information – also known as the two *query string parameters* – separated by the `&` symbol. In Google parlance, `q` is the "query" and `hl` is the "host language" – so with this particular query string, we're passing two pieces of data to Google Maps – the first is that we'd like to search for "paris" and secondly, we'd like the results to be in French. Of course, it's still completely up to the Google Maps server application to decide what to do with these *query string parameters* – but in this case, it works.

At this point, a keen observer might notice that the *query string parameters* are a set of **key-value pairs.** *Sound familiar? What else do we know of that is a set of key-value pairs?* Let's explore further, this time in our own Rails application.

We're going to supply our *dice* resource with data, much like we supplied Google Maps with data in the previous example. Let's open the page to roll the dice again, but this time, let's add a bit of data to the URL and visit `/dice?name=Ben`.

Note that the output is no different than it was before, despite sending data via the query string. But this is only because our server application doesn't do anything with the data it's given! For the first time, we're going to have a look at our Rails **server log.** Where is the server log? The complete log lives at `log/development.log`, but you also see it within the output of our `rails server` command. The server log provides us with extremely valuable information about what's going under the hood of our application, and we should start to become comfortable looking at it often. If we look at the server log output for this particular request, we're going to see something similar to this:

```
Started GET "/dice?name=Ben" for <IP ADDRESS> at <DATETIME>

Processing by DiceController#index as HTML

  Parameters: {"name"=>"Ben"}

  Rendering layout layouts/application.html.erb

  Rendering dice/index.html.erb within layouts/application
```

Everything we'd want to know about how our application is receiving the request is shown right here in the server log. We can see the exact URL path, which controller is processing the request, and – most importantly in the context of this lesson – any data being passed along with the request.

The `params` Hash

The data being received by our request is so important, in fact, that it lives in a special place.

```
Parameters: { "name" => "Ben" }
```

Remember that whole thing about **key-value pairs?** That's right, the key-value pairs supplied in the URL – in this case, `name=Ben` – are being translated into a Ruby *Hash*. And this *Hash* has a special name – `params`. So if we wanted to grab the name "Ben" that's being given to us in the URL, we'd simply pull it out of the hash:

```
params["name"]
```

Let's use it in our dice application! In our *dice controller*, let's grab that value and assign it to a variable that we can use in our view:

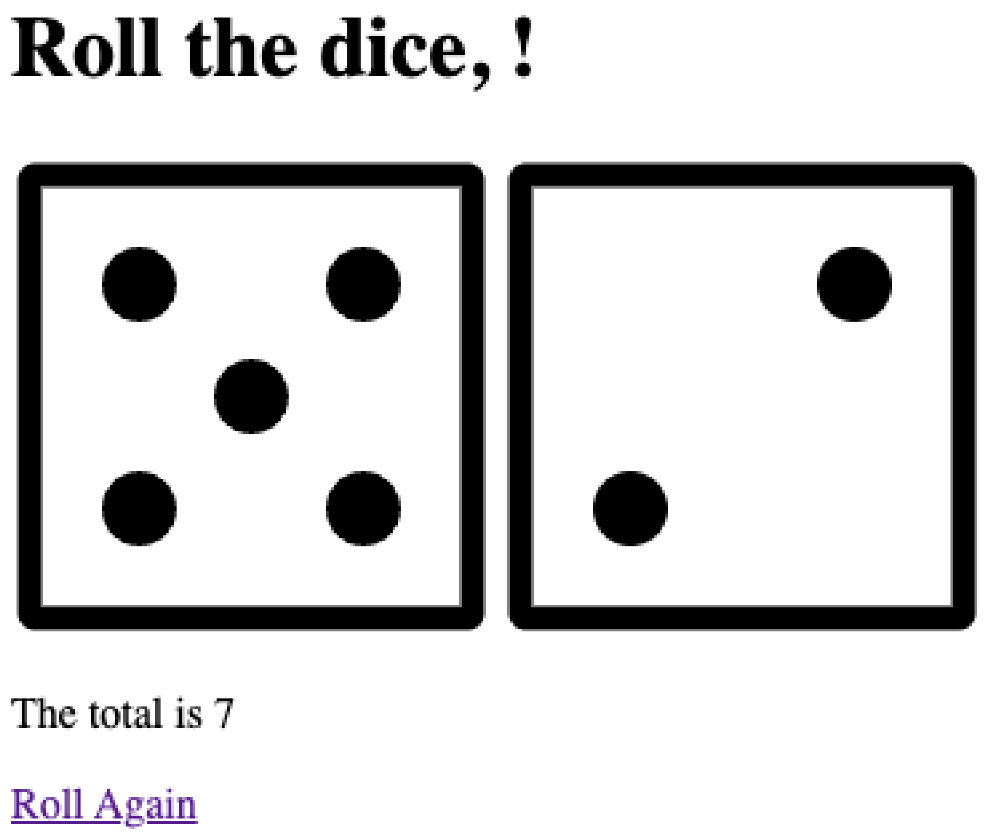
```
def index

  @die1 = rand(1..6)

  @die2 = rand(1..6)

  @total = @die1 + @die2

  @name = params["name"]

end
```

Then, in our view, change the first line so that it reads:

```
<h1>Roll the dice, <%= @name %>!</h1>
```

Refresh the browser (making sure the path is still `/dice?name=Ben`) and see the result. And, try changing the name parameter in the URL to see how the page changes. Our page is now accepting the data from the URL's query string and using it to dynamically alter our resulting HTML. Cool!

To roll again, hit the "Roll Again" link on the page... uh-oh.


Roll the dice, !

The total is 7

Roll Again

What happened?! The URL tells us everything we need to know – we can see that the path is now `/dice` and contains no query string parameters. Why? Because *HTTP is stateless*. Unless we do the job of taking the data passed to the current request and sending it along to the subsequent request, it doesn't happen automatically. Indeed, we've got to modify our link to "Roll Again" in the view in order to make that happen:

```
<a href="/dice?name=<%= @name %>">Roll Again</a>
```

Visit `/dice?name=Ben` again, and we'll see the successful transfer of "state" from one page to the next.

Completing the Game

Of course, the game only works properly at the moment if a name is initially supplied in the URL – again, not a great user experience. It would be better if we could ask for a name, in the event that the player's name is not supplied in the URL. The following, completed code shows us a way we could do that:

```
class DiceController < ApplicationController

  def index
    if params["name"]
      @die1 = rand(1..6)
      @die2 = rand(1..6)
      @total = @die1 + @die2
      @name = params["name"]
    end
  end

end
```

app/controllers/dice_controller.rb

```
<% if @name %>
  <h1>Roll the dice, <%= @name %>!</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?name=<%= @name %>">Roll Again</a>
  </p>
<% else %>
  <h1>What is your name?</h1>
  <form action="/dice" method="GET">
    <input type="text" name="name">
    <button>Let's play!</button>
  </form>
<% end %>
```

app/views/dice/index.html.erb

Cookies

The query string was very much "the way" to persist state across request-response cycles for a very long time on the web. However, the technique comes with a rather large problem – it's too easily hacked. After all, if all the data being transmitted with a new request simply lives in the URL, then all a user needs to do to potentially completely change the application's behavior is to mess with the URL. Eventually, browsers started to support other ways of storing data locally – such as cookies and lightweight databases.

However, this is a discussion for another day. We'll revisit this topic when we discuss application security and user authentication in a future lesson.

POST Parameters

Another, more secure way to pass information from one page to the next is via POST parameters, which is what we'll talk about in the next lesson.

REST

Before we talk about whatever this "REST" thing is, let's revisit how a web request travels from the end-user's web browser, through the maze of our application, and out the other side:

- The end-user hits a path, like `/tacos`, in our application – this request is made possible by the **routes** file, which lets our application know that we want to define a **resource** for *tacos*
- Because we have a resource called *tacos,* we need to have a *tacos* ***controller***. And, in that controller, we've had an *index* ***action***.
- The code in the action creates and populates variables needed by the *index* **view**, which is essentially HTML with a few dynamic bits where we're displaying the contents of those variables. Plain ol' vanilla HTML is then generated and sent back as the **response** to the user's web browser.

It's time to peel another layer of the onion away, and go more in-depth on the topic of **actions**. So far, we've seen the *index* action, and that it's simply a method in our controller class defined by `def index... end` – but what is an **action** and what is it for, exactly? To understand this, we've got to go for a ride through some new concepts, so hang on tight.

A Web App == Things to CRUD

In our initial discussion about Ruby on Rails (in Server-Side Rendering Using Ruby on Rails), we talked about how the primary function of the Rails framework is to allow us **to build applications that are a collection of things to CRUD (Create-Read-Update-Delete)**. We talked about how important that idea is, and to hang onto that thought for later... well, later is now!

If we really think about it, indeed, almost all applications are really a collection of things we want to CRUD in our application's database. A CRM application is where we can CRUD accounts, contacts, sales activities, etc. A photo-sharing social media application, a la Tacostagram, is where we CRUD posts, likes, comments, etc. An e-commerce platform allows for the CRUD of products, reviews, orders, shipments, etc. The web application simply provides a front-end to allow users to perform all those actions in the database.

That's a pretty straightforward concept; however, users can't write SQL directly in our application! Rather, a user fills out a form or clicks a button in our app – how does doing these things translate to say, a SQL `SELECT` or `INSERT` or `UPDATE` statement in the database?

The answer is REST.

REST

What is REST? It stands for Representational State Transfer 🥱 – but nevermind that. All we need to know is that it's yet **another** design pattern! And it's one that's designed to do precisely the job described above – to map user actions in web applications to data. More specifically, it gives us a proven method to map things a user can do in the web browser to the CRUD capabilities of SQL.

The web (HTTP) + databases (CRUD) = REST.

HTTP

We have to learn one more critical thing about web browsers to complete the story of REST. We already seen how web browsers make a **request** to a web server, and expect a HTML **response** in return. In fact, you're probably sick of hearing about it by now! What we haven't looked at, however, is that *web browsers can send (and web servers can accept) different types of requests*.

When we type `https://SomeWebSite.com/` into the web browser's address bar, we are sending a **`GET` request** – `GET` requests are the type of request we've seen thus far. The web address, or URL, is plainly exposed by the web browser, via the address bar. We either type the URL into the address bar ourselves, or we access them via a link on the page we're using – either way, these are `GET` requests.

`POST`, `PATCH`, and `DELETE` requests occur underneath the hood of the web browser. An end-user can't force a web browser to make these types of requests – only the web *developer* can, and it's typically built to do so when end-users submit forms. End-users regularly make these types requests without knowing. For example, the only time we're made aware of `POST` requests are when we try to refresh a page that's the result of a `POST` request, and we get that "are you sure you want to POST this form again" message that we've likely seen before.

Types of requests – `GET`, `POST`, `PATCH`, `DELETE` – are known as *HTTP Methods (or Verbs)* – the actions that web browsers can take. Unlike the "nouns" of the web (aka our *resources*), of which we can have an infinite amount, there are only 4 HTTP Methods (Verbs) in all:

HTTP Method	Usage	CRUD
`GET`	Visiting a web page	Read
`POST`	Submitting a form that creates new data	Create
`PATCH`	Submitting a form that updates existing data	Update
`DELETE`	Submitting a form that deletes existing data	Delete

And, as we can see, each HTTP Method is intended to map to a different part of CRUD we're doing in the database. This is REST! REST is simply a way to design our applications such that we have nouns (aka *resources)* – as many as we want – but each resource has only four verbs (aka *HTTP Methods*) that can be performed.

Implementing REST in Rails

Now that we understand the underlying theory of REST, let's put it to work in Rails.

In Rails, we've already seen how resources (the nouns of REST) are implemented – there is a line in the *routes* file that tells the application the resource exists, and there is a *controller* for the resource. And each of the resource (noun) and HTTP method (verb) combinations result in – you guessed it – an **action**.

Let's start with 2 of the `GET` actions – the ones where a user *reads* data about a particular resource. In practice, this is implemented with two actions – one where we're reading a collection of resources and one where we're reading a single resource. For example:

- A page that's a list of all books in our store, and a page about a single book.

- A page that's the list of all courses in our school, and a page that's about a single course.
- A page that's the list of all accounts in our CRM, and a page that's about a single account.

We've already been introduced to the page that's a list of all records – this is our old friend, the **index** action. When we go to `/books` or `/tacos` or `/dice` or `/cards` – that hits the `index` method of the resource's controller, and subsequently displays the `index.html.erb` view of that resource. For example, in our CRM application, we might want to have a list of all companies. Naturally, this would be the *companies* resource. The complete implementation might look something like this:

```
resources :companies
```

In the routes file – config/routes.rb

```
def index
  @companies = Company.all
end
```

In app/controllers/companies_controller.rb

```
<h1>My CRM</h1>

 <ul>
   <% for company in @companies %>
    <li>
      <%= company["name"] %>
    </li>
   <% end %>
</ul>
```

In app/views/companies/index.html.erb

Not much different than what we did last time (in Data in Views), aside from the fact that we're now properly following the MVC data pattern. Also, note the the action name – `index` – corresponds to the filename of the view – `index.html.erb`. That's good enough for our page of companies – our `index` action – now, let's create the page for a single company.

The `show` Action

The page for a single company is known as the `show` action. Since this is a page for a single resource, we need to know which resource we want. And, we know that every resource (e.g. every company) is identified in the database by an *auto-incrementing primary key ID.* So we have to supply that ID in the `GET` request – in other words, by making it part of the URL path.

So, if the `index` action is the one that displays all companies, and the HTTP method and path is:

```
GET /companies
```

Then, the HTTP method and path for a single company is:

```
GET /companies/123
```

... where `123` can be replaced with whatever the ID is of a record in the database. Let's write the controller code for our `show` action next:

```
def show

  @company = Company.find_by({ "id" => params["id"] })

end
```

There are a couple of new things happening here:

- We're using the `find_by` method of an ActiveRecord object. This method returns exactly one object corresponding to the record that has that `id`.
- `params["id"]` – this is what contains the "123" part of the URL, which is the ID of the record in question

Then, much like the `index` action, we add a view that contains the HTML markup for however we'd like to present the single resource – a single company, in this case – to the user. The name of the view should match the name of the action, followed by `.html.erb`, so for our `show` action, that's `show.html.erb`:

```
<h1><%= @company["name"] %></h1>
<h2><%= @company["city"] %>, <%= @company["state"] %></h2>
```

app/views/companies/show.html.erb

Now we have a page that shows a company and all of that company's pertinent information. Lastly, for the sake of completeness, let's link the `index` and `show` pages together, by editing the `index.html.erb` view:

```
<h1>My CRM</h1>

<ul>
  <% for company in @companies %>
    <li>
      <a href="/companies/<%= company["id"] %>"><%= company["name"] %></a>
    </li>
  <% end %>
</ul>
```

app/views/companies/index.html.erb

Now, when we click on a company's name in the list of companies, we see the detailed page for that company – a (sort of) complete system, from a read-only perspective!

The 7 Actions

It turns out that there are 7 actions in all. This can be counter-intuitive, since there are 4 letters in CRUD, and 4 HTTP methods. But there are extra actions needed to support it all. Here is a full list (assuming "tacos" are the resource in question):

Action	Use Case	HTTP Method	Path
index	Read (display) a list of all tacos	GET	/tacos
show	Read (display) information on a single taco	GET	/tacos/123
new	A form to create a new taco	GET	/tacos/new
create	Receives the information from the "new" form and creates the new taco	POST	/tacos
edit	A form to update an existing taco	GET	/tacos/123/edit
update	Received the information from the "edit" form and updates the existing taco	PATCH	/tacos/123
destroy	Processes a request to destroy an existing taco	DELETE	/tacos/123

Key points:

- There are multiple (4) GETs. Each of these 4 actions is requested by its URL path and can be linked to/bookmarked

- There are 2 pairs of actions that are intended to work together, namely *new/create* and *edit/update* – one is the form, and the other receives the form data
- There are duplicate URL paths – i.e. `/tacos` and `/tacos/123` – but notice that the HTTP methods are different. There is no duplication of the combination of URL path and HTTP method.
- Not all actions have views! In fact, typically only the GET requests do. The other actions simply accept parameters, do something with them (like create or update a record in the database), and send the user on their way.

In the next lessons, we'll be looking in-depth into the remaining (new/create, edit/update, and destroy) actions.

Creating New Records

Here's the "cheatsheet" for all of the available actions for a resource, from the last lesson:

Action	Use Case	HTTP Method	Path
index	Read (display) a list of all tacos	GET	/tacos
show	Read (display) information on a single taco	GET	/tacos/123
new	A form to create a new taco	GET	/tacos/new
create	Receives the information from the "new" form and creates the new taco	POST	/tacos
edit	A form to update an existing taco	GET	/tacos/123/edit

Action	Use Case	HTTP Method	Path
update	Received the information from the "edit" form and updates the existing taco	PATCH	/tacos/123
destroy	Processes a request to destroy an existing taco	DELETE	/tacos/123

In this lesson, we're going to concentrate on creating new records. As briefly explained in the last lesson, creating new instances of a resource involves two actions – the `new` and `create` actions. This is simply due to the nature of HTML – we need to display a form to fill out – the `new` action – and also need to have an action to accept the form data as parameters, do the work of creating the new record in the database, and send the user on their way – the `create` action.

The `new` Action and Rails' Form Helpers

Let's continue building out our CRM application and give the user the ability to create a new company. Here's the code for both the controller and view:

```
def new
  render :template => "companies/new"
end
```

In app/controllers/companies_controller.rb

To break it down, we have a `new` action defined in our controller, and the only thing happening is that we're rendering the view template file which will have the form. Note that the new doesn't save anything to the database; it's just the request to display the form to the user.

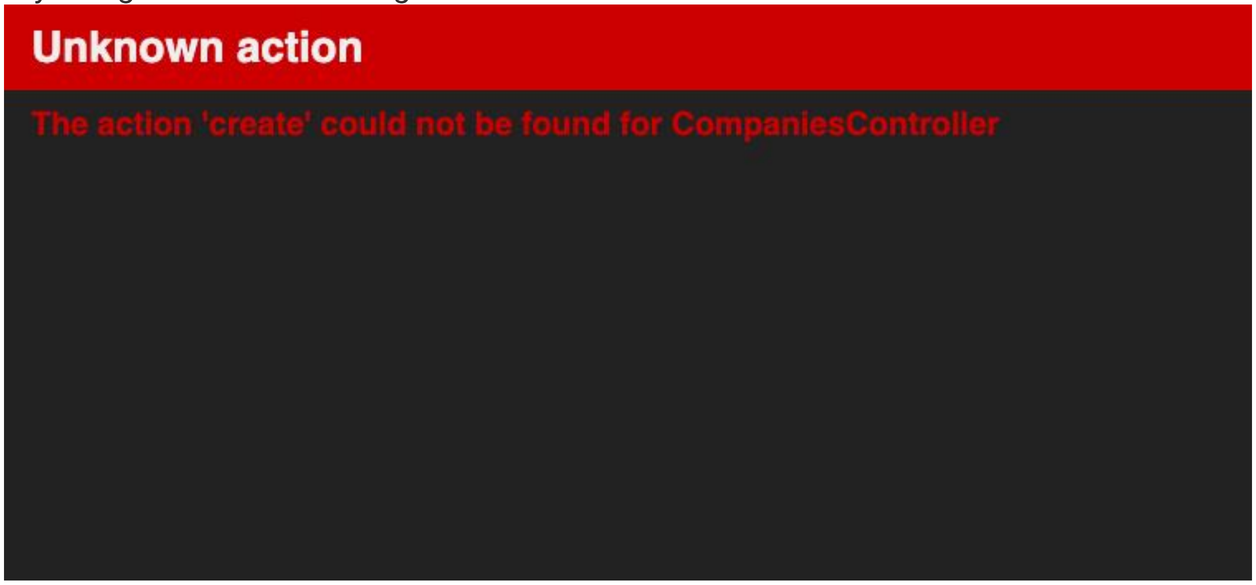
Moving over to the view, we see a fairly vanilla HTML form with the fields we want to save for a company.

```
<form action="/companies" method="post">
  <p>
    <label for="name">Name</label>
    <input type="text" name="name">
  </p>

  <p>
    <label for="city">City</label>
    <input type="text" name="city">

    <label for="state">State</label>
    <input type="text" name="state">
  </p>

  <button>Submit</button>
</form>
```
In app/views/companies/new.html.erb

We can also see that this form is designed to submit to the `create` action. How do we know it's the `create` action? Recall that the `create` action is the combination of the pluralized resource name as the URL (i.e. `/companies`) and the HTTP `POST` method.

The `create` Action

Try filling out and submitting the form!



```
Unknown action

The action 'create' could not be found for CompaniesController
```

That's ok – we don't fear errors! Of course, Rails is letting us know that we don't have a `create` action in our controller, so let's create one:

```
def create

end
```

And let's go back, fill out, and submit the form again... and this time, *nothing happens*.

This is because we've done something a little different this time than we've done before. Remember that, in the past, when we've gotten this error, we solved it by creating a view file – that is, if we were to add a file called `create.html.erb` in our `app/views/companies/` folder, this error would go away. But that's not what we want in this case. *Not all actions have views!* In this case, we want our action to be a method in our controller that only does three things:

- Grab the information the user filled out in the form
- Use the information to create a new `Company` and save it to the database
- Send the user back to the full list of companies

Let's work backwards and add a line of code to our action that will do the last thing – send the user back to the list of companies.

```
def create

  redirect_to "/companies"

end
```

Submit the form again, and we'll notice that we're back at the list of companies. But the new company we tried to create isn't there – of course, this is because we didn't do anything with the data submitted.

Remember how we've previously talked about the *transfer of state* between pages/actions on the web? And how each request-response cycle doesn't really know about the previous ones, or really anything about the order things have happened in the user's journey? The transition from `new` to `create` that we're doing here is really no different. We have to get the data that the `new` action has sent to our `create` action, and that's done in the same way we've done before – through the `params` hash.

Let's comb through our server log (that's the output from the `rails server` command) and look for the `POST`ing of the form. We'll find this:

```
Started POST "/companies" for ...

Processing by CompaniesController#create as HTML

  Parameters: {"name"=>"Netflix", "city"=>"Los Gatos", "state"=>"CA"}
```

Ah! The data the user has typed into the form is, indeed, in the `params` hash. So let's assign the company values using the `params` hash in our controller!

```
def create

  @company = Company.new

  @company["name"] = params["name"]

  @company["city"] = params["city"]

  @company["state"] = params["state"]

  @company.save

  redirect_to "/companies"

end
```

If we go back, fill out the form, and submit... success!

Try it!

What better way to get some practice with creating records than by creating our own social network!

- You'll find that there's a `PostsController` and an `index` action/view set up already in the project.
- Add a `new.html.erb` view and corresponding `new` action in the controller; create a form in `new.html.erb` that will allow the end-user to input a post's `author`, `body`, and `image` (see `db/schema.rb`). `author` is the author's name (e.g. *Brian)*, `body` is the text body of the post (e.g. *Don't these tacos look awesome?*) and `image` is simply the address of an image on the Internet.
- For example, we can head over to https://unsplash.com/ and search for square images of tacos – right-click and grab the URL of the image.
- Add a `create` action to accept values entered into the form and redirect back to the page with all posts.

Handling Relationships

Creating a company is a good example of a single-model form. But what about contacts? Contacts belong to a company, and the `contacts` table has a `contact_id` to fill in – how do we handle a situation like this?

There are a couple of good ways to treat a model form that requires a relationship. The first way would be, as we've done already, to use query string parameters to transfer state from one action to the next. For example, we could create a link on, say, the `companies/show` page that allows our user to *"Create a new contact for this company"*. Let's give that a try.

```
<p>
  <a href="/contacts/new?company_id=<%= @company.id %>">New contact for
<%= @company.name %></a>
</p>
```

In app/views/companies/show.html.erb

This is fairly straightforward to implement and the user experience is not bad, either. Clicking on the link will take us to the `contacts/new` action, transferring the `company_id` of the company for which we want to create a contact. For now, the code in the `new` action simply renders the view with the form:

```
def new
  render :template => "contacts/new"
end
```

In app/controllers/contacts_controller.rb

And then we can add a form in the view with inputs for the contact data like this:

```
<h1>New Contact</h1>

<form action="/contacts" method="post">
  <p>
    <label for="first_name">First Name</label>
    <input type="text" name="first_name" id="first_name">
    <label for="last_name">Last Name</label>
    <input type="text" name="last_name" id="last_name">
  </p>
  <p>
    <label for="email">Email</label>
    <input type="text" name="email" id="email">
    <label for="phone_number">Phone</label>
    <input type="text" name="phone_number" id="phone_number">
  </p>
  <p>
    <label for="company_id">Company ID</label>
    <input type="text" name="company_id" id="company_id">
  </p>
  <button>Submit</button>
</form>
```

app/views/contacts/new.html.erb

And the `create` action in the controller with code assigning each value and redirecting back to the company.

```
def create
  @contact = Contact.new
  @contact["first_name"] = params["first_name"]
  @contact["last_name"] = params["last_name"]
  @contact["email"] = params["email"]
  @contact["phone_number"] = params["phone_number"]
  @contact["company_id"] = params["company_id"]
  @contact.save
  redirect_to "/companies/#{@contact["company_id"]}"
end
```
In app/controllers/contacts_controller.rb

This will all work, but the user experience is lacking. The user won't know what the company's `id` value is; we can't expect them to recognize they just came from `/companies/1` and then enter a **1** as the company id. Instead, we should handle that for them.

Good news, we can! In the link to this new form, we included the company's id in the query string `/contacts/new?company_id=1`. And since it's in the query string, we can access it via the `params` hash: `params["company_id"]`. To pre-populate the company_id input, change the html to this:

```
<p>

  <label for="company_id">Company ID</label>

  <input type="text" name="company_id" id="company_id" value="<%=
params["company_id"] %>">

</p>
```

By adding the `value` attribute to the input, the form field will already have the company id in it for the user. However, it probably shouldn't be a user-editable field. As a text input, the user can see `123` (or whatever the ID of the company is), and has the ability to edit that field. Instead, there is a different type of HTML form field we can use, one that's designed precisely for this purpose – a **hidden field**. Here's what the form HTML looks like with the hidden input:

```
<h1>New Contact</h1>


<form action="/contacts" method="post">
  <p>
    <label for="first_name">First Name</label>
    <input type="text" name="first_name" id="first_name">
    <label for="last_name">Last Name</label>
    <input type="text" name="last_name" id="last_name">
  </p>
  <p>
    <label for="email">Email</label>
    <input type="text" name="email" id="email">
    <label for="phone_number">Phone</label>
    <input type="text" name="phone_number" id="phone_number">
  </p>

  <!-- transfer state from the previous action to the next -->
  <input type="hidden" name="company_id" value="<%=
params["company_id"] %>">

  <button>Submit</button>
</form>
```
app/views/contacts/new.html.erb

Now, the `company_id` will be successfully included each step of the way – we've transferred state all the way from the `<a>` tag to create a new contact for a company, through the form, and finally, to the `create` action.

Finishing the Story – Edit, Update, and Delete

Once again, here's the "cheatsheet" for all of the available actions for a resource, from the last couple of lessons:

Action	Use Case	HTTP Method	Path
index	Read (display) a list of all tacos	GET	/tacos
show	Read (display) information on a single taco	GET	/tacos/123
new	A form to create a new taco	GET	/tacos/new
create	Receives the information from the "new" form and creates the new taco	POST	/tacos
edit	A form to update an existing taco	GET	/tacos/123/edit
update	Received the information from the "edit" form and updates the existing taco	PATCH	/tacos/123
destroy	Processes a request to destroy an existing taco	DELETE	/tacos/123

Let's complete the story of REST. We've done `index`, `show`, `new`, and `create` – only `edit`, `update`, and `destroy` are left.

Edit/Update

The editing and subsequent updating of existing data is very similar to `new` and `create`. The only difference, of course, is that we're dealing with data that's already in the database instead of brand-new records. In fact, the `edit` view is usually going to be very similar to the `new` view. The main differences are:

- the `action` attribute is directing the form to submit to the update route

- the form fields are pre-populated with the existing values using the `value` attribute in the HTML.

```
<h1>Edit Company</h1>

<form action="/companies/<%= @company["id"] %>" method="post">
  <input type="hidden" name="_method" value="patch">

  <p>
    <label for="name">Name</label>
    <input type="text" name="name" value="<%= @company["name"] %>">
  </p>

  <p>
    <label for="city">City</label>
    <input type="text" name="city" value="<%= @company["city"] %>">

    <label for="state">State</label>
    <input type="text" name="state" value="<%= @company["state"] %>">
  </p>

  <button>Save</button>
</form>
```
app/views/companies/edit.html.erb

There is one other important difference. By default, the `form` element in HTML doesn't know about the `PATCH` method. So to trick it, we use a hidden input field that will override the form's request method and send our request to the update action as it should in the REST pattern.

Now, let's examine the controller:
```
  def new
  end

  def create
    @company = Company.new
    @company["name"] = params["name"]
    @company["city"] = params["city"]
    @company["state"] = params["state"]
    @company.save
    redirect_to "/companies"
  end

  def edit
    @company = Company.find_by({ "id" => params["id"] })
  end

  def update
    @company = Company.find_by({ "id" => params["id"] })
    @company["name"] = params["name"]
    @company["city"] = params["city"]
    @company["state"] = params["state"]
    @company.save
    redirect_to "/companies"
  end
```
In app/controllers/companies_controller.rb

There are only a couple of differences between *new/create* and *edit/update,* due to the nature of new vs. existing data.

- In `new`, we're creating a brand-new, empty `Company`, the data to be filled out by the end-user
- In `edit`, we're grabbing the existing `Company` from the database; **the ID is provided in the URL** and is available in the `params` hash as `params["id"]`
- In `create`, we're getting the user-inputted data from the `params` hash and using it to create and save a new `Company`
- In `update`, we're getting the user-inputted data from the `params` hash as well; but first, we're grabbing the existing `Company` from the database based on the provided `id` in the `params` hash, then we're updating the existing record using the data from the `params` hash

Deleting (Destroying) Data

Finally, we've reached the final action of REST – deleting an existing record. Let's add a `destroy` action to our `CompaniesController`:

```
def destroy

  @company = Company.find_by({ "id" => params["id"] })

  @company.destroy

  redirect_to "/companies"

end
```

When this action is invoked, we'll start by finding the existing record in the data, based on the `id` provided in the `params` hash – much like the `edit` or `update` actions. We'll then perform the `destroy` method on the `Company`. **This permanently and irreversibly deletes this record from the database.** Once that's done, we then send the user back to the list of companies.

But, how do we invoke this action? We've talked about how an end-user can only perform `GET` requests via the URL, so we can't create a link to do this or otherwise provide a web address for a user to delete a record. And for good reason! It's a destructive action, and we want to be able to control the user's access to it.

Just like HTTP `POST` and `PATCH` requests, `DELETE` requests can only be performed via a form submission:

```
<form action="/companies/<%= @company["id"] %>" method="post">

  <input type="hidden" name="_method" value="delete">

  <button>Delete Company</button>

</form>
```

And similar to the edit form, we override the form method with a hidden field so that the request follows the REST design pattern.

The Completed Controller

Here is the complete code for our `CompaniesController`, in all its glory:

```
class CompaniesController < ApplicationController



  def index

    @companies = Company.all

  end



  def show

    @company = Company.find_by({ "id" => params["id"] })
```

```
end

def new
end

def create
  @company = Company.new
  @company["name"] = params["name"]
  @company["city"] = params["city"]
  @company["state"] = params["state"]
  @company.save
  redirect_to "/companies"
end

def edit
  @company = Company.find_by({ "id" => params["id"] })
end

def update
```

```
    @company = Company.find_by({ "id" => params["id"] })

    @company["name"] = params["name"]

    @company["city"] = params["city"]

    @company["state"] = params["state"]

    @company.save

    redirect_to "/companies/#{@company["id"]}"

  end



  def destroy

    @company = Company.find_by({ "id" => params["id"] })

    @company.destroy

    redirect_to "/companies"

  end



end
```

Of course, this is a fairly basic implementation. It doesn't include things like data validation, error handling, security, or otherwise any other logic that protects against users doing weird things – which they often do. But it works! We've successfully built an app that completely CRUDs company data and allows for the creation of associated contact data as well.

The best news of all is that, with the exception of possible small changes in business logic, all of our controllers can follow the exact same formula that we've followed here. As we add controllers and views for activities, salespeople, industries, or whatever else might be a part of our CRM world, we can use the same conventions and techniques as we have for companies. That's the beauty of REST, MVC, and software design patterns in general – we don't have to "reinvent the wheel" for every new thing we want to build. We only need to think about our domain model and our resources, and we're well on our way to building our finished product.