Time	~ 2 - 3 hours
Learning Goals	• Learn how to create a Rails app

Now that you have initialized a git repository and created a Rails app, let's now build the URL shortener that you have built in Week 4, but this time using Rails. This will be a guided tutorial to help you get started. Try not to just copy-and-paste code. Make sure you understand what's going on as it will help you with building the PairBnB app.

1 Run bundle install.

2 Run bundle exec rails db:create to create the database.

3 Run rails s in your terminal. Go to your browser and key in localhost:3000 to view your app. You will see a welcome message that says "Welcome aboard! You're riding Ruby on Rails!". If you are not seeing this and are receiving error messages in your terminal, please flag a staff.

4 We will now starting working on the url feature. Let's adopt good practice and **work on a feature branch**. Do git checkout -b feature/url to check out to a new branch. To make sure that you are in the right branch, type git branch and make sure that the * is at the branch name, not the master branch.

5 Let's now create a table to store our long and short URLs. To do this, we will first create a migration file and the Url model. Type rails g model Url long_url:string short_url:string in the terminal. Then go to the folders db/migrate/ and app/models/ to see what has been created for you. Wow, a migration file and the Url model were created with just one simple command. See how magical Rails is!

6 If you refresh your browser now, you will receive an error message that says: ActiveRecord::PendingMigrationError. Oops, we have not performed the migration yet. Let's run bundle exec rails db:migrate (you can still use bundle exec rake db:migrate, so choose one that it's easier for you to remember. The reason why we can use rails instead of rake is given here.). Remember to shutdown your server before running the migration!

7 Once your migration is successful, start the server and refresh your browser. All is well! :)

8 Now in yet another terminal, run rails c. This is similar to the rake console that you have been using over the past 2 weeks. Type Url.all and if you see #<ActiveRecord::Relation []>, good job! We can now test code and query from our database.

9 At this juncture, let's **git add and commit** what we have done so far, i.e. "create a table and model for Url"

10 Let's now set up routes for our Url model. In config/routes.rb, type resources :urls.

To see what this returns, go to the terminal and type bundle exec rails routes. You should see something like this:

```
  Prefix Verb   URI Pattern               Controller#Action
    urls GET    /urls(.:format)           urls#index
         POST   /urls(.:format)           urls#create
 new_url GET    /urls/new(.:format)       urls#new
edit_url GET    /urls/:id/edit(.:format)  urls#edit
     url GET    /urls/:id(.:format)       urls#show
         PATCH  /urls/:id(.:format)       urls#update
         PUT    /urls/:id(.:format)       urls#update
         DELETE /urls/:id(.:format)       urls#destroy
    root GET    /                         welcome#index
```

11 Notice that the 7 actions (index, show, new, create, edit, update, destroy) have been setup for you along with the prefix and URI pattern. If you're curious what .(:format) means, here's a good explanation from Stack Overflow.

12 We would now set urls/index as the root page. In routes.rb, add root 'urls#index' on top of resources :urls (an app can have only one root path)

13 Since we have made some changes in the config folder, we will need to reboot the server to see the changes made. In your terminal where the server is running, type Ctrl-C to close the server. Rerun the server again by doing rails s. If you have run rails c just now, remember to restart it too.

14 Oh, and let's not forget to commit this change. Remember commits should be small. Go ahead and git add and git commit. The commit message could be something like: "set up resources for urls and set url/index as root page".

15 Now if you refresh your browser, you will see an error message that says uninitialized constant UrlController. This is because we have **not generated the Urls controller*** yet! Let's do it now. In your terminal, type rails g controller Urls. Please make sure you have the "s" so that the noun is **plural**. In app/controllers, you will see that a file url_controller.rb has been generated for you. In the app/views folder, a urls folder has also been created. This is where you will place all the view files associated with the Url controller.

16 Refresh your browser again. This time the error message is The action 'index' could not be found for UrlsController. This is because even though the file urls_controller.rb has been generated for us, we still have to define the actions inside the file. Let's go to urls_controller.rb and do the following: def index

17 @urls = Url.all

```
18      end
```

19 Refresh your browser again. Oops, this time we have another error that says Template is missing. This means that we need to **create a view file** that corresponds to the index action. In app/views/urls, create a file called index.html.erb. We will first make a table that displays the long and shortened urls. In urls/index.html.erb, do the following <table>

```
20  <tr>
21  <th>Original URL</th>
22  <th>Shortened URL</th>
23  </tr>
24
25  <%  @urls.each do |url| %>
26     <tr>
27     <td> <a href = <%= url.long_url %> ><%= url.long_url %></a></td>
28     <td> <a href = <%= url.short_url %> ><%= url.short_url %></a> </td>
29     </tr>
30  <% end %>
31  </table>
```

32 Note: Click here to find out why we use .html.erb in Rails instead of .erb as we did in Sinatra.

33 Refresh the browser. You should now see a table with headers displayed on your site.

34 Let's create one or two examples just to make sure that our code is working. We will worry about the validations and shortening method later on. Go to rails c and do the following: Url.create(long_urL: "http://www.google.com", short_url: "123456")

35 Url.create(long_url: "http://www.facebook.com", short_url: "abcdef") Typing Url.all will return you the following: Url Load (0.9ms) SELECT "urls".* FROM "urls"

36 => #<ActiveRecord::Relation [#<Url id: 1, long_url: "http://www.google.com", short_url: "123456", created_at: "2017-02-01 10:17:28", updated_at: "2017-02-01 10:17:28">, #<Url id: 2, long_url: "http://www.facebook.com", short_url: "abcdef", created_at: "2017-02-01 10:17:50", updated_at: "2017-02-01 10:17:50">]> This indicates that our two examples have been successfully created in our database.

37 Now refresh your browser. These 2 examples should appear. Before moving on, let's **git add and commit this change**. The commit message could be something like: ```

38 set up index for url

39

40 1. Set up index action in Urls controller

41 2. Create a view page for index with a table that display the long and short urls

42 3. Create 2 examples in the Url table to test the code

43 ```

44 We wil now work on the show action which will show the desired long url and its shortened url. In url_controller.rb, after the index action, do the following: def show

45 @url = Url.find(params[:id])

46 end Here's a pretty nice explanation on what params are. Later on we will use byebug to see our params in action too!

47 Create a file called show.html.erb in app/views/urls, do the following: <h3>This is urls/show</h3>

48 <%= @url.long_url %>

49 <%= @url.short_url %>

50 <%= link_to 'Back', urls_path %> The last line links us back to the index page. Instead of using href here, we can use the link_to helper. The pathname urls_path was obtained from the **prefix** of the index action when we run bundle exec rails routes. We just need to add _path behind the prefix to get the right

route. Alternatively, you could write <%= link_to 'Back', controller: :urls, action: :index %> to bring you back to the index page. To find out other usages of the link_to helper, check out this blog.

Alright, now that we have our show page set up, we will need to navigate from the index page to the show page. In urls/index.html.erb, add the line <td> <%= link_to 'Show', url_path(url) %> </td> so that your code is now:

```
<table>
<tr>
<th>Original URL</th>
<th>Shortened URL</th>
</tr>

<%  @urls.each do |url| %>
   <tr>
   <td> <a href = <%= url.long_url %> ><%= url.long_url %></a></td>
   <td> <a href = <%= url.short_url %> ><%= url.short_url %></a> </td>
   <td> <%= link_to 'Show', url_path(url) %> </td>
   </tr>
<% end %>
</table>
```

Once again, we used the link_to helper. The url_path is the prefix for urls#show, while the url inside the parenthesis refers to the Url object. Refresh your browser and click on one of the “Show” linka. It should now direct you to the “show” page for that particular long and short URL.

Let’s now play with byebug. In urls_controller.rb, place the byebug right after def show, as below:

```
def show
   byebug
   @url = Url.find(params[:id])
end
```

Now click on one of the “Show” links and go your terminal where the server is running. Your byebug would have been triggered by now. Type in “params” - what do you observe?

Once you are done exploring, it’s time to **git add and commit!** We will leave it to

you to write a good git commit message that will help you remember what you have done when you refer back to this app in the future.

Now to the core of our app - shortening a long URL! To do this, let's set up our validations and methods in the Url model. Go to app/models/url.rb. Copy your validations and shorten method from your Bitly clone here. **Git add and commit.**

We will now create a form so that a user can key in a URL. In app/views/urls, create a file named _form.html.erb, take note of the **underscore**! The underscore here represents that the view file we are creating is a **partial** view file and can be reused instead of typing out the form again and again (imagine having to create the same form for new and edit!). You can find more information on partials here. In this partial view file, do the following: <%= form_for @url do |f| %>

```
  <p>
    <%= f.label "Type your URL here:" %><br>
    <%= f.text_field :long_url %>
  </p>

  <p>
    <%= f.submit "Shorten URL!" %>
  </p>
    <% end %>
```

Here we are using the form helper form_for so that we can bind this form to our Url model. It is also best practice to use form_for if your form takes in values that are attributes of an Active Record object. To have a better understanding of form_for, we recommend you to read the following references: a. Binding a form to an object b. Difference between HTML form and form_for c. Difference between form_tag and form_for d. Why you need the authenticity_token which is created when you use form_for in Rails e. What is the submit value?

Okie dokie! Now that we have set up a form, we will need to create a new and create action, and place the appropriate links to navigate to the form. Let's work on the new action first. In urls_controller.rb, def new

```
  @url = Url.new
```

```
83  end
```

84 In app/views/urls, create a file called new.html.erb and do the following: <h3>This is urls/new</h3>

```
85  <%= render 'form' %>
86  <%= link_to 'Back', urls_path %>
```

Note that because _form.html.erb is a partial, we can use the render method to display the form as part of new.html.erb. Pretty cool! Finally in urls/index.html.erb, let's add a link that brings us to the form: <%= link_to "Shorten a Url!", new_url_path %>

87 Note: in our Bitly clone, we would have placed the form in the index page instead. To demonstrate how to use the new action, we will place a link that navigates to the form instead. Once you are done with the tutorial, free free to figure out how to render the form in the index page!

88 We are almost there! Now that we have a form that allows users to key in a URL, let's define the create action. In urls_controller.rb:

```
def create
89    @url = Url.new(params[:url])
90    @url.shorten
91    if @url.save
92      redirect_to @url
93    else
94      render 'new'
95    end
96  end
```

Now let's test our button. Type a URL in your form and click on the button. Oopsy, an error message that says ActiveModel::ForbiddenAttributesError appears! This means that we will need to whitelist our parameters so that we can use them! To do this, we will make use of strong parameters.

97 **Strong parameters** is a Rails 4 plugin that controls what is being exposed when doing mass assignment. Mass assignment is a good thing because we can store all our attributes in a hash. But it also causes us to be vulnerable to attack from hackers who can easily change the attributes of an Active Record object for undesired purposes. With strong parameters, we can still do mass assignment but

with more control on what should and should not be exposed. In urls_controller.rb, add a private method called url_params which only permits long_url to be exposed:

```
98  private
99  def url_params
100   params.require(:url).permit(:long_url)
101 end
```

Then in the create method, change the line @url = Url.new(params[:url]) to @url = Url.new(url_params). Use byebug if you want to explore more. Note: Check out this blog for more examples on how to use strong_params.

Refresh your browser and check that your long URL gets shortened, and that the shortened URL is also saved in your database. Note that your shortened URL will not redirect to its associated long URL yet. We will take care of this later. Once everything is working as expected, you know the drill by now, time to **git add and commit**

If we want to allow the user to edit the long URL, we can set up the edit and update action. However, we will not do this here and will let you explore how to do edit and update in your PairBnB app. Let's now set up a destroy method. In urls/index.html.erb, add the following line at the appropriate place:

```
<td><%= link_to 'Delete', url_path(url), method: :delete, data: { confirm: 'Are you sure?' } %></td>
```

Note how the path is written here as there is no prefix path for destroy. In urls_controller.rb, add the destroy method:

```
103 def destroy
104   @url = Url.find(params[:id])
105   @url.destroy
106   redirect_to urls_path
107 end
```

Now try to delete a URL. You will notice that a box pops up to ask you if you are sure that you want to delete the URL. This box appears because of the **data attribute** (data: { confirm: 'Are you sure?' }) AND **JavaScript**. Read this documentation and Stack Overflow answer.

Once you can delete a URL, **git add and commit**!

We are almost done with our first Rails app! Let's now make sure our short URL redirects us to the right link. In routes.rb, define a new route called "short":

```
get '/urls/:id/short' => 'urls#short'
```

In urls_controller.rb:

```
def short
```

```
109       url = Url.find(params[:id])
110       redirect_to url.long_url
111   end
```

In urls/index.html.erb, replace <td> <a href = <%= url.short_url %> ><%= url.short_url %> </td> with <td> <%= link_to url.short_url, controller: :urls, action: :short, id: url %> </td>

112 Test that your code is working as desired. Then **git add and commit**.

113 Last but not least, since we have completed the URL feature, we can now **merge our feature branch back to the master branch**. Let's first push our branch to our remote repo by doing git push -u origin feature/url.

114 Go to your repo in Github. You will see that a message saying that you have pushed a branch to the repo. Click on "Compare and pull request" and follow the instructions there.

115 Once the merge is successful, go back to your local machine and checkout back to your master branch by doing git checkout master.

116 Now we would like our **local repo to be up-to-date with the remote repo**. We do this by doing git pull origin master.

117 Congratulations, your local and remote repo are now in sync with each other! If you want to, you can delete your branch by using the command git branch -d feature/url.

Yay! There you are! A URL shortener built using Rails! Let's now move on to learn some front-end stuff!

IMPORTANT NOTE: Different people have different order of doing things when building apps. For example, you need not necessarily start with building a model. You can also start from building the controller. As you build more apps, you will eventually find a sequence of doing things that you are comfortable with.

We understand that there are a lot of new terms and programming jargons and they may feel very foreign to you at the moment. Just remember, keep on practising until it becomes muscle memory to you! The more you do the same thing over and over again, the better you are at it! We know you can!!! =)