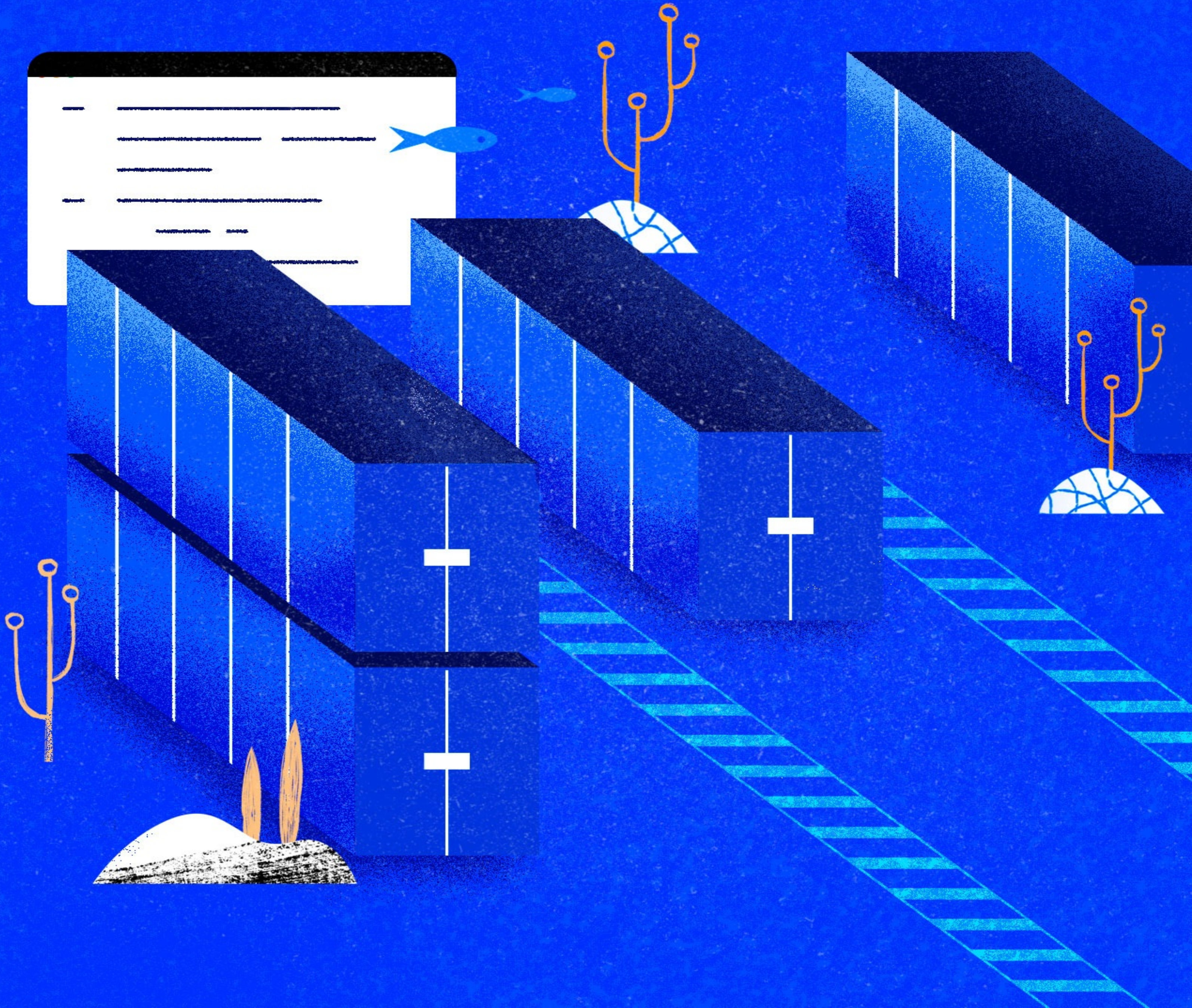
KATHLEEN JUELL

RAILS ON CONTAINERS



ISBN 978-0-9997730-8-6

Rails on Containers

Kathleen Juell

DigitalOcean, New York City, New York, USA

2020-12

Rails on Containers

About DigitalOcean

DigitalOcean is a cloud services platform delivering the simplicity developers love and businesses trust to run production applications at scale. It provides highly available, secure and scalable compute, storage and networking solutions that help developers build great software faster. Founded in 2012 with offices in New York and Cambridge, MA, DigitalOcean offers transparent and affordable pricing, an elegant user interface, and one of the largest libraries of open source resources available. For more information, please visit https://www.digitalocean.com or follow @digitalocean on Twitter.

Preface - Getting Started with this Book

To work with the examples in this book, we recommend that you have a local development environment running Ubuntu 18.04. You can also provision a remote Ubuntu 18.04 server and develop Rails applications that way if you prefer. The first chapter in this book covers all the prerequisites that you will need to develop Rails applications in either a local or remote environment.

When working with Kubernetes, we also recommend that you have a local machine or server with the kubectl command line tool installed.

Introduction

About this Book

This book is designed as an introduction to building and containerizing a Ruby on Rails application. It explains common development tasks that you will encounter when building Rails applications – adding nested resources, a JavaScript framework (Stimulus.js), Bootstrap CSS styles, and Sidekiq and Redis to process background jobs. Once you have an application ready for development, the last part of this book will guide you through containerizing your Rails application for continued development.

Motivation for this Book

Often, resources on development and deployment are relatively independent of one another: guides on containers and Kubernetes rarely cover application development, and tutorials on languages and frameworks are often focused on languages and other nuances rather than on deployment.

This book is designed to be a full-stack introduction to containers and Kubernetes by way of Rails application development. It assumes that readers want an introduction not only to the fundamentals of containerization, but also to the basics of working with Rails and a database backend.

Learning Goals and Outcomes

The goal for this guide is to serve readers interested in Rails application development, as well as readers who would like to learn more about working with containers and container orchestrators. It assumes a shared interest in moving away from highly individuated local environments, in favor of repeatable, reproducible application environments that ensure consistency and ultimately resiliency over time.

How To Build a Ruby on Rails Application

Written by Kathleen Juell

Rails is a web application framework written in Ruby. It takes an opinionated approach to application development, assuming that set conventions best serve developers where there is a common goal. Rails therefore offers conventions for handling routing, stateful data, asset management, and more to provide the baseline functionality that most web applications need.

Rails follows the model-view-controller (MCV) architectural pattern, which separates an application's logic, located in models, from the routing and presentation of application information. This organizational structure — along with other conventions that allow developers to extract code into helpers and partials — ensures that application code isn't repeated unnecessarily.

In this tutorial, you will build a Rails application that will enable users to post information about sharks and their behavior. It will be a good starting point for future application development.

Prerequisites

To follow this tutorial, you will need: - A local machine or development server running Ubuntu 18.04. Your development machine should have a non-root user with administrative privileges and a firewall configured with `ufw`. For instructions on how to set this up, see our [Initial Server Setup with Ubuntu 18.04] tutorial. - [Node.js] and [npm] installed on your local machine or development server. This tutorial uses Node.js version **10.16.3** and npm version **6.9.0**. For guidance on installing Node.js and npm on Ubuntu 18.04, follow the instructions in the "Installing Using a PPA" section of [How To Install Node.js on Ubuntu 18.04]. - Ruby, [rbenv], and Rails installed on your local machine or development server, following Steps 1-4 in [How To Install Ruby on Rails with rbenv on Ubuntu 18.04]. This tutorial uses Ruby **2.5.1**, rbenv **1.1.2**, and Rails **5.2.0**.

Step 1 — Installing SQLite3

Before creating our Rails shark application, we will need to ensure that we have a database to store user data. Rails is configured to use [SQLite] by default, and this is often a good choice in development. Since our application data doesn't require a high level programmatic extensibility, SQLite will meet our needs.

First, update your package index:

```
sudo apt update
```

Next, install the `sqlite3` and `libsqlite3-dev` packages:

```
sudo apt install sqlite3 libsqlite3-dev
```

This will install both SQLite and its required development files.

Check your version to confirm that the installation was successful:

```
sqlite3 --version
```

```
Output
3.22.0 2018-01-22 18:45:57 0c55d179733b46d8d0ba4d88e01a25e1067
7046ee3da1d5b1581e86726f2alt1
```

With SQLite installed, you are ready to begin developing your application.

Step 2 — Creating a New Rails Project

With our database installed, we can create a new Rails project and look at some of the default boilerplate code that Rails gives us with the [`rails new` command].

Create a project called `sharkapp` with the following command:

```
rails new sharkapp
```

You will see a good deal of output telling you what Rails is creating for your new project. The output below highlights some significant files, directories, and commands:

Output

```
     create
     . . .
     create  Gemfile
     . . .
      create  app
      . . .
      create  app/controllers/application_controller.rb
      . . .
      create  app/models/application_record.rb
      . . .
      create  app/views/layouts/application.html.erb
      . . .
      create  config
      create  config/routes.rb
      create  config/application.rb
      . . .
      create  config/environments
      create  config/environments/development.rb
      create  config/environments/production.rb
      create  config/environments/test.rb
      . . .
      create  config/database.yml
      create  db
      create  db/seeds.rb
      . . .
```

```
        run  bundle install
      . . .
Bundle complete! 18 Gemfile dependencies, 78 gems now installe
d.
Use `bundle info [gemname]` to see where a bundled gem is inst
alled.
      . . .
* bin/rake: Spring inserted
* bin/rails: Spring inserted
```

The output highlighted here tells you that Rails has created the following: - `Gemfile`: This file lists the gem dependencies for your application. A [gem](gem) is a Ruby software package, and a Gemfile allows you to manage your project's software needs. - `app`: The `app` directory is where your main application code lives. This includes the models, controllers, views, assets, helpers, and mailers that make up the application itself. Rails gives you some application-level boilerplate for the MCV model to start out in files like `app/models/application_record.rb`, `app/controllers/application_controller.rb`, and `app/views/layouts/application.html.erb`. - `config`: This directory contains your application's configuration settings: - `config/routes.rb`: Your application's route declarations live in this file. - `config/application.rb`: General settings for your application components are located in this file. - `config/environments`: This directory is where configuration settings for your environments live. Rails includes three environments by default: `development`, `production`, and `test`. - `config/database.yml`: Database configuration settings live in this file, which is broken into four sections: `default`, `development`, `production`, and `test`. Thanks to the Gemfile that came with the `rails new` command, which included the `sqlite3` gem, our `config/database.yml` file has its `adapter` parameter set to `sqlite3` already, specifying that we will use an SQLite database with this application. - `db`: This folder includes a directory for database migrations called `migrate`, along with the `schema.rb` and `seeds.rb` files. `schema.db` contains information about your database, while `seeds.rb` is where you can place seed data for the database.

Finally, Rails runs the `bundle install` command to install the dependencies listed in your `Gemfile`.

Once everything is set up, navigate to the `sharkapp` directory:

```
cd sharkapp
```

You can now start the Rails server to ensure that your application is working, using the `rails server` command. If you are working on your local machine, type:

```
rails server
```

Rails binds to `localhost` by default, so you can now access your application by navigating your browser to `locahost:3000`, where you will see the following image:

Rails Landing Page

If you are working on a development server, first ensure that connections are allowed on port `3000`:

```
sudo ufw allow 3000
```

Then start the server with the `--binding` flag, to bind to your server IP:

```
rails server --binding=your_server_ip
```

Navigate to `http://your_server_ip:3000` in your browser, where you will see the Rails welcome message.

Once you have looked around, you can stop the server with `CTRL+C`.

With your application created and in place, you are ready to start building from the Rails boilerplate to create a unique application.

Step 3 — Scaffolding the Application

To create our shark information application, we will need to create a model to manage our application data, views to enable user interaction with that data, and a controller to manage communication between the model and the views. To build these things we will use the `rails generate scaffold` command, which will give us a model, a [database migration](https://edgeguides.rubyonrails.org/active_record_migrations.html) to alter the database schema, a controller, a full set of views to manage [Create, Read, Update, and Delete](https://en.wikipedia.org/wiki/Create,_read,_update_and_delete) (CRUD) operations for the application, and templates for partials, helpers, and tests.

Because the `generate scaffold` command does so much work for us, we'll take a closer look at the resources it creates to understand the work that Rails is doing under the hood.

Our `generate scaffold` command will include the name of our model and the fields we want in our database table. Rails uses [Active Record](https://guides.rubyonrails.org/active_record_basics.html) to manage relationships between application data, constructed as objects with models, and the application database. Each of our models is a [Ruby class](https://ruby-doc.org/core-2.5.3/Class.html), while also inheriting from the `ActiveRecord::Base` class. This means that we can work with our model class in the same way that we would work with a Ruby class, while also pulling in methods from Active Record. Active Record will then ensure that each class is mapped to a table in our database, and each instance of that class to a row in that table.

Type the following command to generate a `Shark` model, controller, and associated views:

```
rails generate scaffold Shark name:string facts:text
```

With `name:string` and `facts:text` we are giving Rails information about the fields we would like in our database table and the type of data they should accept. Both will give us room to input what we would like, though `text` will allow more characters for shark facts.

When you type this command, you will again see a long list of output that explains everything Rails is generating for you. The output below highlights some of the more significant things for our setup:

```
Output
      invoke  active_record
      create    db/migrate/20190804181822_create_sharks.rb
      create    app/models/shark.rb
      . . .
      invoke  resource_route
       route    resources :sharks
      invoke  scaffold_controller
      create    app/controllers/sharks_controller.rb
      invoke    erb
      create      app/views/sharks
      create      app/views/sharks/index.html.erb
      create      app/views/sharks/edit.html.erb
      create      app/views/sharks/show.html.erb
      create      app/views/sharks/new.html.erb
      create      app/views/sharks/_form.html.erb
      . . .
```

Rails has created the model at `app/models/shark.rb` and a database migration to go with it: `db/migrate/20190804181822_create_sharks.rb`. The timestamp on your migration file will differ from what you see here.

It has also created a controller, `app/controllers/sharks_controller.rb`, as well as the views associated with our application's CRUD operations, collected under `app/views/sharks`. Among these views is a partial, `_form.html.erb`, that contains code used across views.

Finally, Rails added a new resourceful route, `resources :sharks`, to `config/routes.rb`. This enables the Rails router to match incoming HTTP requests with the `sharks` controller and its associated views.

Though Rails has done much of the work of building out our application code for us, it is worth taking a look at some files to understand what is happening.

First, let's look at the controller file with the following command:

```
cat app/controllers/sharks_controller.rb
```

```
Output
class SharksController < ApplicationController
  before_action :set_shark, only: [:show, :edit, :update, :des
troy]

  # GET /sharks
  # GET /sharks.json
  def index
    @sharks = Shark.all
  end

  # GET /sharks/1
  # GET /sharks/1.json
  def show
  end

  # GET /sharks/new
  def new
    @shark = Shark.new
  end

  # GET /sharks/1/edit
  def edit
  end

  # POST /sharks
```

```
  # POST /sharks.json
  def create
    @shark = Shark.new(shark_params)

    respond_to do |format|
      if @shark.save
        format.html { redirect_to @shark, notice: 'Shark was s
uccessfully created.' }
        format.json { render :show, status: :created, locatio
n: @shark }
      else
        format.html { render :new }
        format.json { render json: @shark.errors, status: :unp
rocessable_entity }
      end
    end
  end

  # PATCH/PUT /sharks/1
  # PATCH/PUT /sharks/1.json
  def update
    respond_to do |format|
      if @shark.update(shark_params)
        format.html { redirect_to @shark, notice: 'Shark was s
uccessfully updated.' }
        format.json { render :show, status: :ok, location: @sh
ark }
```

```
      else
        format.html { render :edit }
        format.json { render json: @shark.errors, status: :unp
rocessable_entity }
      end
    end
  end

  # DELETE /sharks/1
  # DELETE /sharks/1.json
  def destroy
    @shark.destroy
    respond_to do |format|
      format.html { redirect_to sharks_url, notice: 'Shark was
successfully destroyed.' }
      format.json { head :no_content }
    end
  end

  private
    # Use callbacks to share common setup or constraints betwe
en actions.
    def set_shark
      @shark = Shark.find(params[:id])
    end

    # Never trust parameters from the scary internet, only all
```

```
ow the white list through.
    def shark_params
      params.require(:shark).permit(:name, :facts)
    end
end
```

The controller is responsible for managing how information gets fetched and passed to its associated model, and how it gets associated with particular views. As you can see, our `sharks` controller includes a series of methods that map roughly to standard CRUD operations. However, there are more methods than CRUD functions, to enable efficiency in the case of errors.

For example, consider the `create` method:

~/sharkapp/app/controllers/sharks_controller.rb

```
. . .
  def create
    @shark = Shark.new(shark_params)

    respond_to do |format|
      if @shark.save
        format.html { redirect_to @shark, notice: 'Shark was s
uccessfully created.' }
        format.json { render :show, status: :created, locatio
n: @shark }
      else
        format.html { render :new }
        format.json { render json: @shark.errors, status: :unp
rocessable_entity }
      end
    end
  end
. . .
```

If a new instance of the `Shark` class is successfully saved, `redirect_to` will spawn a new request that is then directed to the controller. This will be a `GET` request, and it will be handled by the `show` method, which will show the user the shark they've just added.

If there is a failure, then Rails will render the `app/views/sharks/new.html.erb` template again rather than making another request to the router, giving users another chance to submit their data.

In addition to the sharks controller, Rails has given us a template for an `index` view, which maps to the `index` method in our controller. We will use this as the root view for our application, so it's worth taking a look at it.

Type the following to output the file:

```
cat app/views/sharks/index.html.erb
```

```
Output
<p id="notice"><%= notice %></p>

<h1>Sharks</h1>

<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Facts</th>
      <th colspan="3"></th>
    </tr>
  </thead>

  <tbody>
    <% @sharks.each do |shark| %>
      <tr>
        <td><%= shark.name %></td>
        <td><%= shark.facts %></td>
        <td><%= link_to 'Show', shark %></td>
        <td><%= link_to 'Edit', edit_shark_path(shark) %></td>
        <td><%= link_to 'Destroy', shark, method: :delete, dat
a: { confirm: 'Are you sure?' } %></td>
      </tr>
    <% end %>
  </tbody>
```

```
</table>

<br>

<%= link_to 'New Shark', new_shark_path %>
```

The `index` view iterates through the instances of our `Shark` class, which have been mapped to the `sharks` table in our database. Using ERB templating, the view outputs each field from the table that is associated with an individual shark instance: `name` and `facts`.

The view then uses the `link_to` helper to create a hyperlink, with the provided string as the text for the link and the provided path as the destination. The paths themselves are made possible through the helpers that became available to us when we defined the `sharks` resourceful route with the `rails generate scaffold` command.

In addition to looking at our `index` view, we can also take a look at the `new` view to see how Rails uses partials in views. Type the following to output the `app/views/sharks/new.html.erb` template:

```
cat app/views/sharks/new.html.erb
```

```
Output
<h1>New Shark</h1>


<%= render 'form', shark: @shark %>


<%= link_to 'Back', sharks_path %>
```

Though this template may look like it lacks input fields for a new shark entry, the reference to `render 'form'` tells us that the template is pulling in the `_form.html.erb` partial, which extracts code that is repeated across views.

Looking at that file will give us a full sense of how a new shark instance gets created:

```
cat app/views/sharks/_form.html.erb
```

Output

```
<%= form_with(model: shark, local: true) do |form| %>
  <% if shark.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(shark.errors.count, "error") %> prohib
ited this shark from being saved:</h2>

      <ul>
      <% shark.errors.full_messages.each do |message| %>
        <li><%= message %></li>
      <% end %>
      </ul>
    </div>
  <% end %>

  <div class="field">
    <%= form.label :name %>
    <%= form.text_field :name %>
  </div>

  <div class="field">
    <%= form.label :facts %>
    <%= form.text_area :facts %>
  </div>

  <div class="actions">
```

```
    <%= form.submit %>
  </div>
<% end %>
```

This template makes use of the `form_with` form helper. Form helpers are designed to facilitate the creation of new objects from user input using the fields and scope of particular models. Here, `form_with` takes `model: shark` as an argument, and the new form builder object that it creates has field inputs that correspond to the fields in the `sharks` table. Thus users have form fields to enter both a shark `name` and shark `facts`.

Submitting this form will create a JSON response with user data that the rest of your application can access by way of the params method, which creates a `ActionController::Parameters` object with that data.

Now that you know what `rails generate scaffold` has produced for you, you can move on to setting the root view for your application.

Step 4 — Creating the Application Root View and Testing Functionality

Ideally, you want the landing page of your application to map to the application's root, so users can immediately get a sense of the application's purpose.

There are a number of ways you could handle this: for example, you could create a `Welcome` controller and an associated `index` view, which would give users a generic landing page that could also link out to different parts of the application. In our case, however, having users land on our `index` sharks view will be enough of an introduction to the application's purpose for now.

To set this up, you will need to modify the routing settings in `config/routes.rb` to specify the root of the application.

Open `config/routes.rb` for editing, using `nano` or your favorite editor:

```
nano config/routes.rb
```

The file will look like this:

```
~/sharkapp/config/routes.rb
Rails.application.routes.draw do
  resources :sharks
  # For details on the DSL available within this file, see http://guides.rubyonrails.org/routing.html
end
```

Without setting something more specific, the default view at `http://localhost:3000` or `http://your_server_ip:3000` will be the default Rails welcome page.

In order to map the root view of the application to the `index` view of the sharks controller, you will need to add the following line to the file:

```
~/sharkapp/config/routes.rb
Rails.application.routes.draw do
  resources :sharks

  root 'sharks#index'
  # For details on the DSL available within this file, see htt
p://guides.rubyonrails.org/routing.html
end
```

Now, when users navigate to your application root, they will see a full listing of sharks, and have the opportunity to create a new shark entry, look at existing entries, and edit or delete given entries.

Save the file and exit your editor when you are finished editing. If you used `nano` to edit the file, you can do so by pressing `CTRL+X`, `Y`, then `ENTER`

You can now run your migrations with the following command:

```
rails db:migrate
```

You will see output confirming the migration.

Start your Rails server once again. If you are working locally, type:

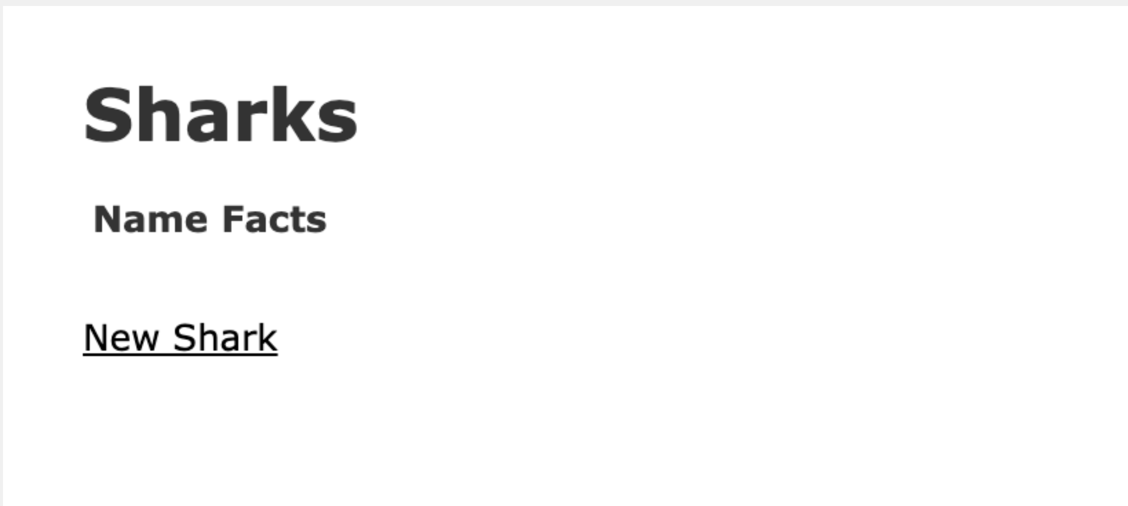
```
rails s
```

On a development server, type:

```
rails s --binding=your_server_ip
```

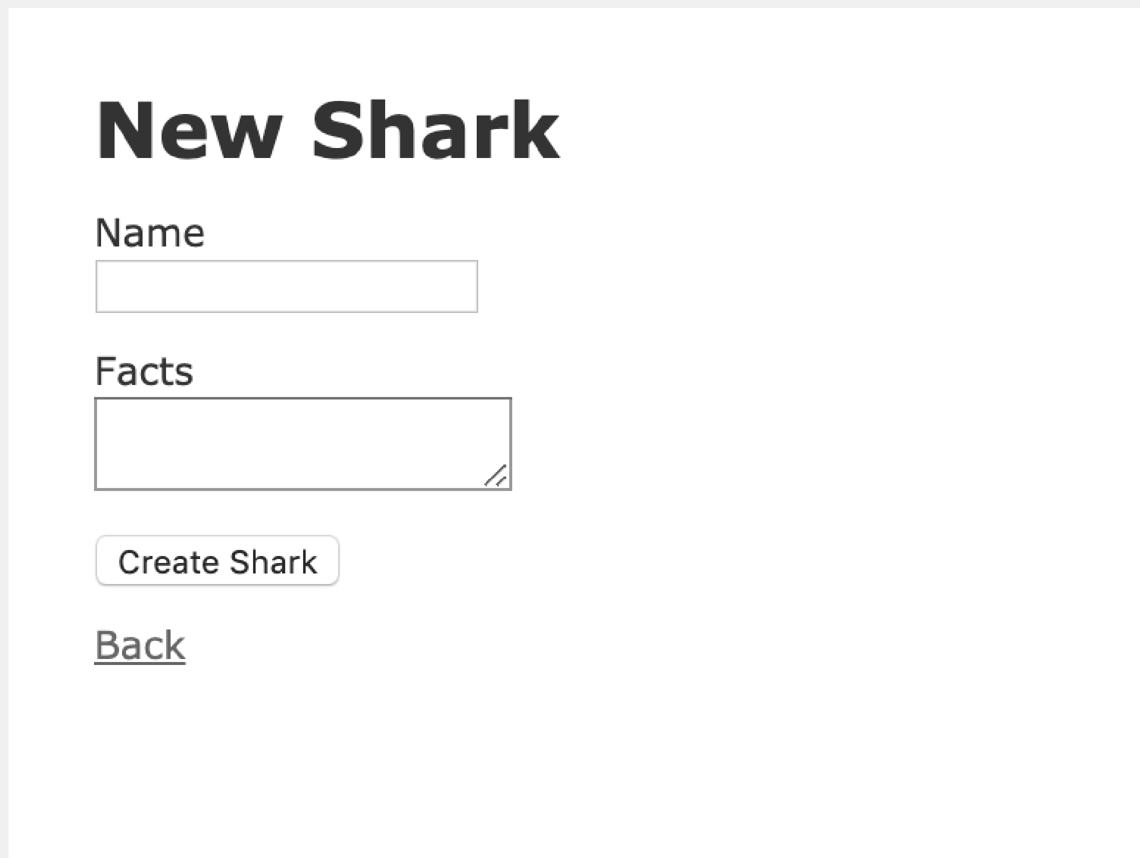
Navigate to `localhost:3000` if you are working locally, or `http://your_server_ip:3000` if you are working on a development server.

Your application landing page will look like this:



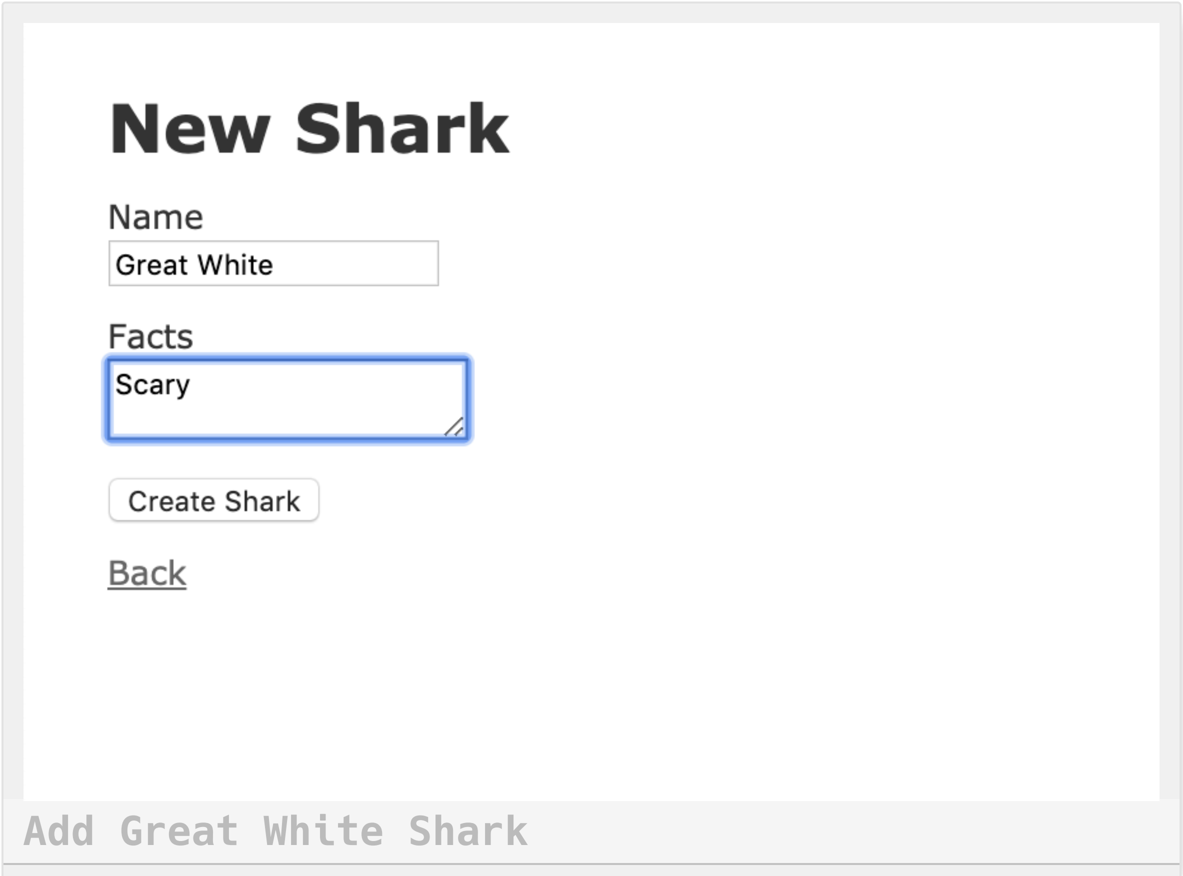
Application Landing Page

To create a new shark, click on the **New Shark** link at the bottom of the page, which will take you to the `sharks/new` route:



Create New Shark

Let's add some demo information to test our application. Input “Great White” into the **Name** field and “Scary” into the **Facts** field:

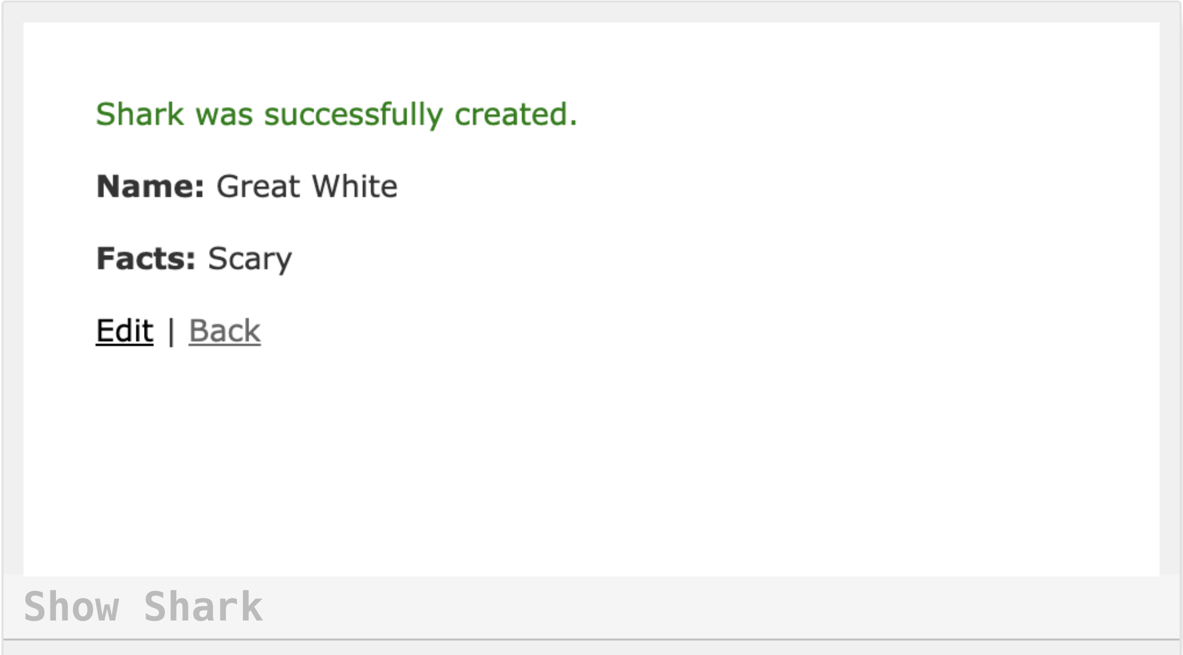


Click on the **Create** button to create the shark.

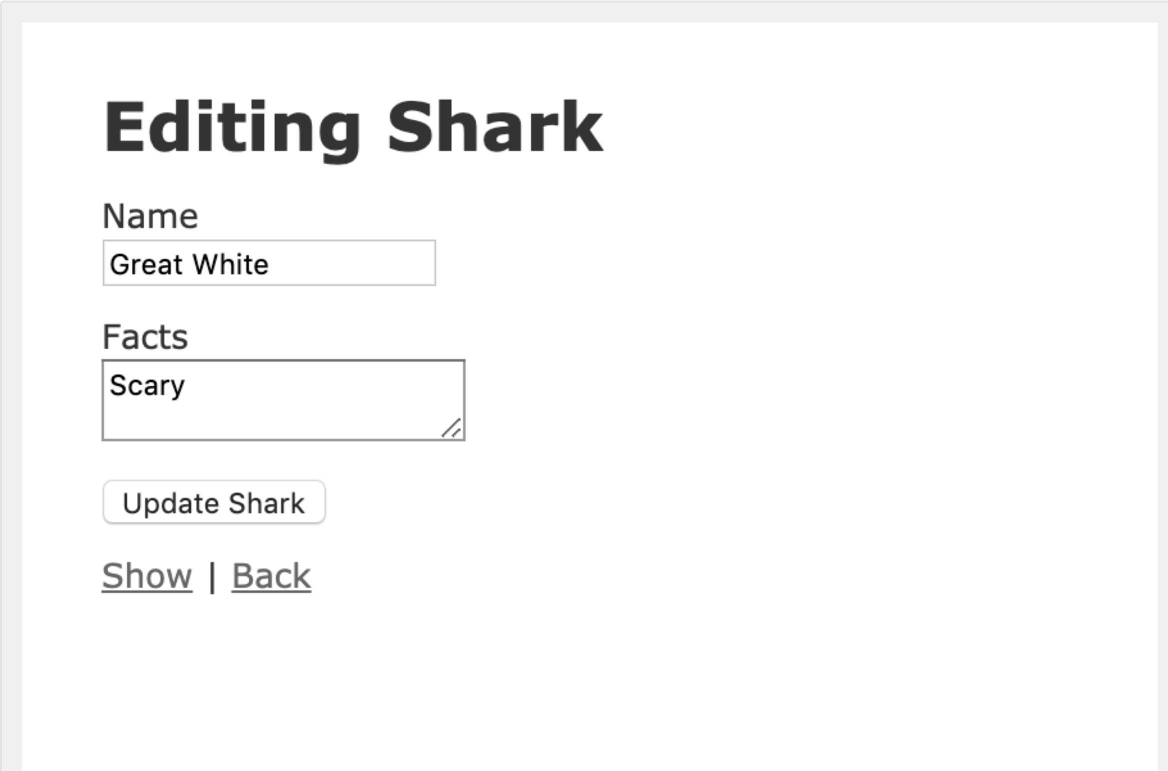
This will direct you to the `show` route, which, thanks to the `before_action` filter, is set with the `set_shark` method, which grabs the `id` of the shark we've just created:

~/sharkapp/app/controllers/sharks_controller.rb

```
class SharksController < ApplicationController
  before_action :set_shark, only: [:show, :edit, :update, :des
troy]

  . . .

  def show
  end

  . . .

  private
    # Use callbacks to share common setup or constraints betwe
en actions.
    def set_shark
      @shark = Shark.find(params[:id])
    end
  . . .
```

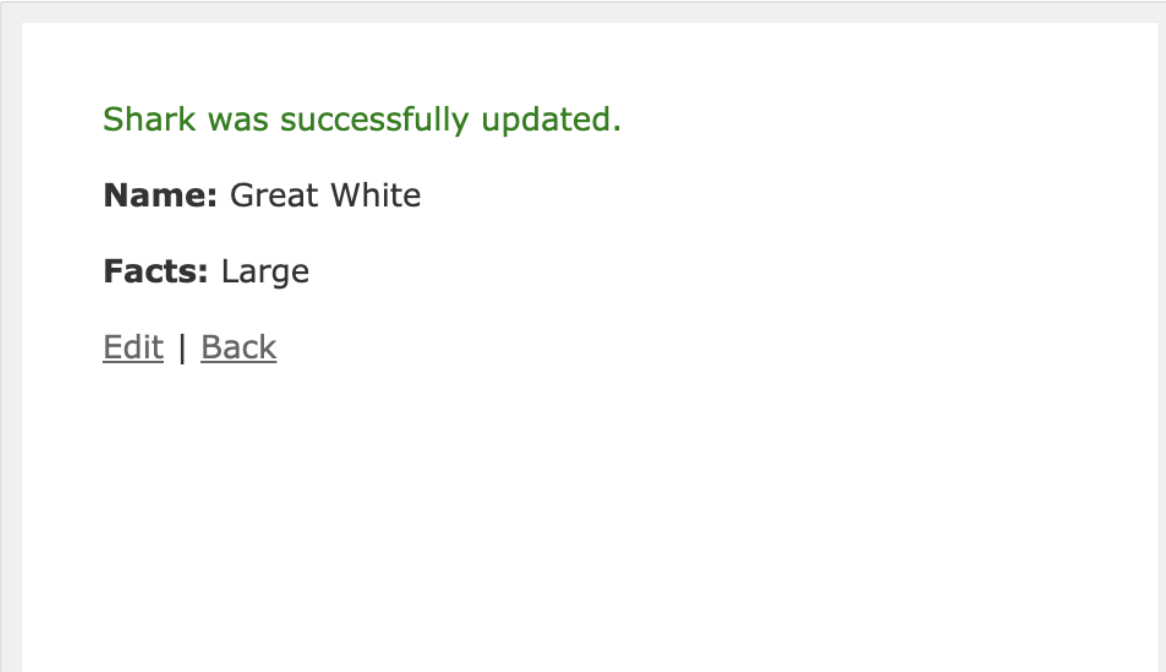


You can test the edit function now by clicking **Edit** on your shark entry. This will take you to the `edit` route for that shark:



Edit Shark

Change the `facts` about the Great White to read “Large” instead of “Scary” and click **Update Shark**. This will take you back to the `show` route:



Updated Shark

Finally, clicking **Back** will take you to your updated `index` view:

New Index View

Now that you have tested your application's basic functionality, you can add some validations and security checks to make everything more secure.

Step 5 — Adding Validations

Your shark application can accept input from users, but imagine a case where a user attempts to create a shark without adding facts to it, or creates an entry for a shark that's already in the database. You can create mechanisms to check data before it gets entered into the database by adding validations to your models. Since your application's logic is located in its models, validating data input here makes more sense than doing so elsewhere in the application.

Note that we will not cover writing validation tests in this tutorial, but you can find out more about testing by consulting the Rails documentation.

If you haven't stopped the server yet, go ahead and do that by typing `CTRL+C`.

Open your `shark.rb` model file:

```
nano app/models/shark.rb
```

Currently, the file tells us that the `Shark` class inherits from `ApplicationRecord`, which in turn inherits from `ActiveRecord::Base`:

```
~/sharkapp/app/models/shark.rb
class Shark < ApplicationRecord
end
```

Let's first add some validations to our `name` field to confirm that the field is filled out and that the entry is unique, preventing duplicate entries:

```
~/sharkapp/app/models/shark.rb
class Shark < ApplicationRecord
  validates :name, presence: true, uniqueness: true
end
```

Next, add a validation for the `facts` field to ensure that it, too, is filled out:

```
~/sharkapp/app/models/shark.rb
class Shark < ApplicationRecord
  validates :name, presence: true, uniqueness: true
  validates :facts, presence: true
end
```

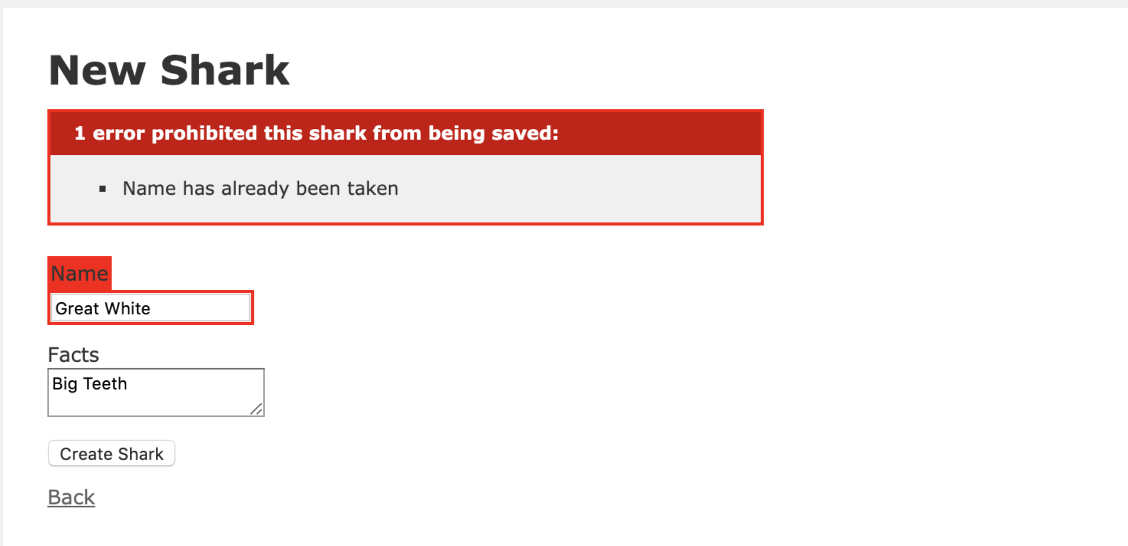
We are less concerned here with the uniqueness of the facts, as long as they are associated with unique shark entries.

Save and close the file when you are finished.

Start up your server once again with either `rails s` or `rails s --binding=your_server_ip`, depending on whether you are working locally or with a development server.

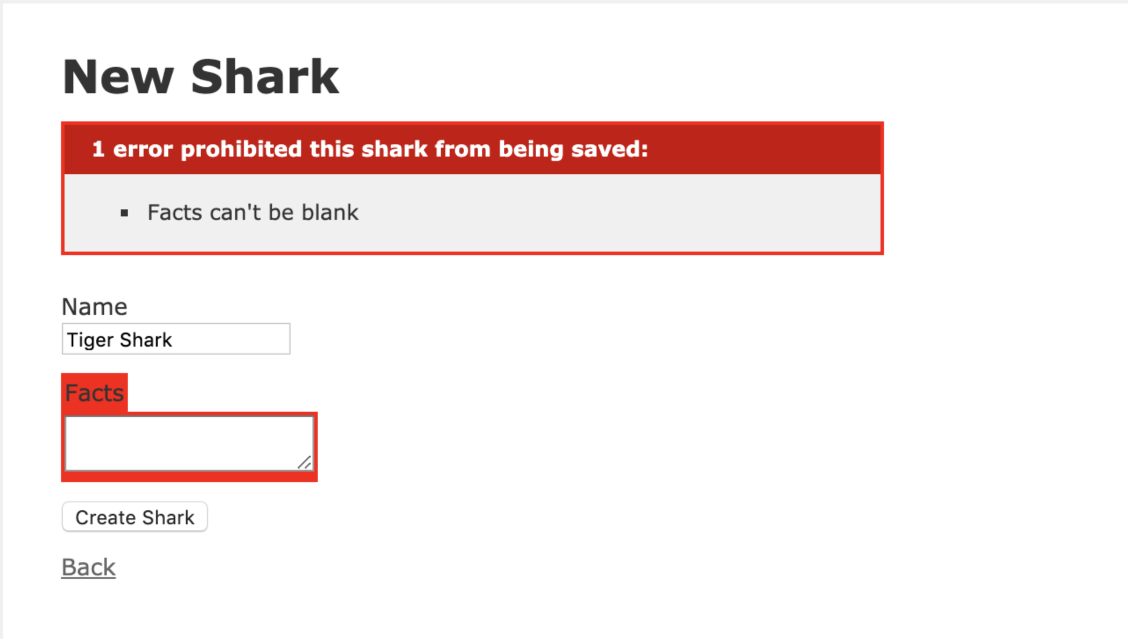
Navigate to your application's root at `http://localhost:3000` or `http://your_server_ip:3000`.

Click on **New Shark**. In the form, add “Great White” to the **Name** field and “Big Teeth” to the **Facts** field, and then click on **Create Shark**. You should see the following warning:



Unique Validation Warning

Now, let's see if we can check our other validation. Click **Back** to return to the homepage, and then **New Shark** once again. In the new form, enter “Tiger Shark” in the **Name** field, and leave **Facts** blank. Clicking **Create Shark** will trigger the following warning:



Fact Presence Warning

With these changes, your application has some validations in place to ensure consistency in the data that's saved to the database. Now you can turn your attention to your application's users and defining who can modify application data.

Step 6 — Adding Authentication

With validations in place, we have some guarantees about the data that's being saved to the database. But what about users? If we don't want any and all users adding to the database, then we should add some authentication measures to ensure that only permitted users can add sharks. In order to do this, we'll use the `http_basic_authenticate_with` method, which will allow us to create a username and password combination to authenticate users.

There are a number of ways to authenticate users with Rails, including working with the `bcrypt` or `devise` gems. For now, however, we will add a method to our application controller that will apply to actions across our application. This will be useful if we add more controllers to the application in the future.

Stop your server again with `CTRL+C`.

Open the file that defines your `ApplicationController`:

```
nano app/controllers/application_controller.rb
```

Inside, you will see the definition for the `ApplicationController` class, which the other controllers in your application inherit from:

~/sharkapp/app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base
end
```

To authenticate users, we'll use a hardcoded username and password with the `http_basic_authenticate_with` method. Add the following code to the file:

~/sharkapp/app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base
  http_basic_authenticate_with name: 'sammy', password: 'shark', except: [:index, :show]
end
```

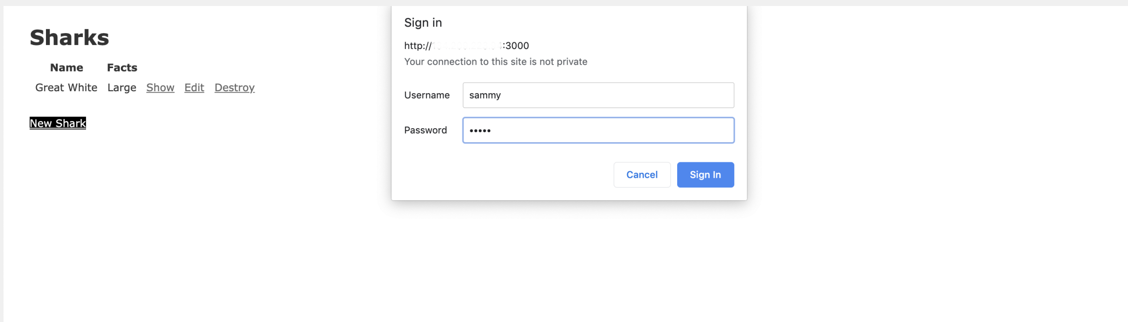
In addition to supplying the username and password here, we've also restricted authentication by specifying the routes where it should **not** be required: `index` and `show`. Another way of accomplishing this would have been to write `only: [:create, :update, :destroy]`. This way, all users will be able to look at all of the sharks and read facts about particular sharks. When it comes to modifying site content, however, users will need to prove that they have access.

In a more robust setup, you would not want to hardcode values in this way, but for the purposes of demonstration, this will allow you to see how you can include authentication for your application's routes. It also lets you see how Rails stores session data by default in cookies: once you authenticate on a specified action, you will not be required to authenticate again in the same session.

Save and close `app/controllers/application_controller.rb` when you are finished editing. You can now test authentication in action.

Start the server with either `rails s` or `rails s --binding=your_server_ip` and navigate to your application at either `http://localhost:3000` or `htt` `p://your_server_ip:3000`.

On the landing page, click on the **New Shark** button. This will trigger the following authentication window:



User Authentication

If you enter the username and password combination you added to `app/controllers/application_controller.rb`, you will be able to securely create a new shark.

You now have a working shark application, complete with data validations and a basic authentication scheme.

Conclusion

The Rails application you created in this tutorial is a jumping off point that you can use for further development. If you are interested in exploring the Rails ecosystem, the project documentation is a great place to start.

You can also learn more about adding nested resources to your project by reading How To Create Nested Resources for a Ruby on Rails Application, which will show you how to build out your application's models and routes.

Additionally, you might want to explore how to set up a more robust frontend for your project with a framework such as React. How To Set Up a Ruby on Rails Project with a React Frontend offers guidance on how to do this.

If you would like to explore different database options, you can also check out How To Use PostgreSQL with Your Ruby on Rails Application on Ubuntu 18.04, which walks through how to work with PostgreSQL instead of SQLite. You can also consult our library of PostgreSQL tutorials to learn more about working with this database.

How To Create Nested Resources for a Ruby on Rails Application

Written by Kathleen Juell

Ruby on Rails is a web application framework written in Ruby that offers developers an opinionated approach to application development. Working with Rails gives developers: - Conventions for handling things like routing, stateful data, and asset management. - A firm grounding in the model-view-controller (MCV) architectural pattern, which separates an application's logic, located in models, from the presentation and routing of application information.

As you add complexity to your Rails applications, you will likely work with multiple models, which represent your application's business logic and interface with your database. Adding related models means establishing meaningful relationships between them, which then affect how information gets relayed through your application's controllers, and how it is captured and presented back to users through views.

In this tutorial, you will build on an existing Rails application that offers users facts about sharks. This application already has a model for handling shark data, but you will add a nested resource for posts about individual sharks. This will allow users to build out a wider body of thoughts and opinions about individual sharks.

Prerequisites

To follow this tutorial, you will need: - A local machine or development server running Ubuntu 18.04. Your development machine should have a non-root user with administrative privileges and a firewall configured with `ufw`. For instructions on how to set this up, see our [Initial Server Setup with Ubuntu 18.04] tutorial. - [Node.js] and [npm] installed on your local machine or development server. This tutorial uses Node.js version <>10.16.3<> and npm version <>6.9.0<>. For guidance on installing Node.js and npm on Ubuntu 18.04, follow the instructions in the “Installing Using a PPA” section of [How To Install Node.js on Ubuntu 18.04]. - Ruby, [rbenv], and Rails installed on your local machine or development server, following Steps 1-4 in [How To Install Ruby on Rails with rbenv on Ubuntu 18.04]. This tutorial uses Ruby <>2.5.1<>, rbenv <>1.1.2<>, and Rails <>5.2.3<>. - SQLite installed, and a basic shark information application created, following the directions in [How To Build a Ruby on Rails Application].

Step 1 — Scaffolding the Nested Model

Our application will take advantage of Active Record [associations] to build out a relationship between `Shark` and `Post` models: posts will belong to particular sharks, and each shark can have multiple posts. Our `Shark` and `Post` models will therefore be related through [`belongs_to`] and [`has_many`] associations.

The first step to building out the application in this way will be to create a `Post` model and related resources. To do this, we can use the `rails generate scaffold` command, which will give us a model, a [database migration] to alter the database schema, a controller, a full set of views to manage standard Create, Read, Update, and Delete (CRUD) operations, and templates for partials, helpers, and tests. We will need to modify these resources, but using the `scaffold` command will save us some time and energy since it generates a structure we can use as a starting point.

First, make sure that you are in the `sharkapp` directory for the Rails project that you created in the prerequisites:

```
cd sharkapp
```

Create your `Post` resources with the following command:

```
rails generate scaffold Post body:text shark:references
```

With `body:text`, we're telling Rails to include a `body` field in the `posts` database table — the table that maps to the `Post` model. We're also including the `:references` keyword, which sets up an association between the `Shark` and `Post` models. Specifically, this will ensure that a foreign key representing each shark entry in the `sharks` database is added to the `posts` database.

Once you have run the command, you will see output confirming the resources that Rails has generated for the application. Before moving on, you can check your database migration file to look at the relationship that now exists between your models and database tables. Use the following command to look at the contents of the file, making sure to substitute the timestamp on your own migration file for what's shown here:

```
cat db/migrate/20190805132506_create_posts.rb
```

You will see the following output:

```
Output
class CreatePosts < ActiveRecord::Migration[5.2]
  def change
    create_table :posts do |t|
      t.text :body
      t.references :shark, foreign_key: true

      t.timestamps
    end
  end
end
```

As you can see, the table includes a column for a shark foreign key. This key will take the form of `model_name_id` — in our case, `shark_id`.

Rails has established the relationship between the models elsewhere as well. Take a look at the newly generated `Post` model with the following command:

```
cat app/models/post.rb
```

```
Output
class Post < ApplicationRecord
  belongs_to :shark
end
```

The `belongs_to` association sets up a relationship between models in which a single instance of the declaring model belongs to a single instance of the named model. In the case of our application, this means that a single post belongs to a single shark.

In addition to setting this relationship, the `rails generate scaffold` command also created routes and views for posts, as it did for our shark resources in [Step 3](https://www.digitalocean.com/community/tutorials/how-to-build-a-ruby-on-rails-application#step-3-%E2%80%94-scaffolding-the-application) of [How To Build a Ruby on Rails Application](https://www.digitalocean.com/community/tutorials/how-to-build-a-ruby-on-rails-application).

This is a useful start, but we will need to configure some additional routing and solidify the Active Record association for the `Shark` model in order for the relationship between our models and routes to work as desired.

Step 2 — Specifying Nested Routes and Associations for the Parent Model

Rails has already set the `belongs_to` association in our `Post` model, thanks to the `:references` keyword in the `rails generate scaffold` command, but in order for that relationship to function properly we will need to specify a `has_many` association in our `Shark` model as well. We will also need to make changes to the default routing that Rails gave us in order to make post resources the children of shark resources.

To add the `has_many` association to the `Shark` model, open `app/models/shark.rb` using `nano` or your favorite editor:

```
nano app/models/shark.rb
```

Add the following line to the file to establish the relationship between sharks and posts:

```
~/sharkapp/app/models/shark.rb
class Shark < ApplicationRecord
  has_many :posts
  validates :name, presence: true, uniqueness: true
  validates :facts, presence: true
end
```

One thing that is worth thinking about here is what happens to posts once a particular shark is deleted. We likely do not want the posts associated with a deleted shark persisting in the database. To ensure that any posts associated with a given shark are eliminated when that shark is deleted, we can include the `dependent` option with the association.

Add the following code to the file to ensure that the `destroy` action on a given shark deletes any associated posts:

```
~/sharkapp/app/models/shark.rb
class Shark < ApplicationRecord
  has_many :posts , dependent: :destroy
  validates :name, presence: true, uniqueness: true
  validates :facts, presence: true
end
```

Once you have finished making these changes, save and close the file. If you are using `nano`, you can do this by pressing `CTRL+X`, `Y`, then `ENTER`.

Next, open your `config/routes.rb` file to modify the relationship between your resourceful routes:

```
nano config/routes.rb
```

Currently, the file looks like this:

```
~/sharkapp/config/routes.rb
Rails.application.routes.draw do
  resources :posts
  resources :sharks

  root 'sharks#index'
  # For details on the DSL available within this file, see htt
p://guides.rubyonrails.org/routing.html
end
```

The current code establishes an independent relationship between our routes, when what we would like to express is a dependent relationship between sharks and their associated posts.

Let's update our route declaration to make `:sharks` the parent of `:posts`. Update the code in the file to look like the following:

```
~/sharkapp/config/routes.rb
Rails.application.routes.draw do
  resources :sharks do
    resources :posts
  end
  root 'sharks#index'
  # For details on the DSL available within this file, see htt
p://guides.rubyonrails.org/routing.html
end
```

Save and close the file when you are finished editing.

With these changes in place, you can move on to updating your `posts` controller.

Step 3 — Updating the Posts Controller

The association between our models gives us methods that we can use to create new post instances associated with particular sharks. To use these methods, we will need to add them our posts controller.

Open the posts controller file:

```
nano app/controllers/posts_controller.rb
```

Currently, the file looks like this:

~/sharkapp/controllers/posts_controller.rb

```
class PostsController < ApplicationController
  before_action :set_post, only: [:show, :edit, :update, :dest
roy]

  # GET /posts
  # GET /posts.json
  def index
    @posts = Post.all
  end

  # GET /posts/1
  # GET /posts/1.json
  def show
  end

  # GET /posts/new
  def new
    @post = Post.new
  end

  # GET /posts/1/edit
  def edit
  end

  # POST /posts
```

```
  # POST /posts.json
  def create
    @post = Post.new(post_params)

    respond_to do |format|
      if @post.save
        format.html { redirect_to @post, notice: 'Post was suc
cessfully created.' }
        format.json { render :show, status: :created, locatio
n: @post }
      else
        format.html { render :new }
        format.json { render json: @post.errors, status: :unpr
ocessable_entity }
      end
    end
  end

  # PATCH/PUT /posts/1
  # PATCH/PUT /posts/1.json
  def update
    respond_to do |format|
      if @post.update(post_params)
        format.html { redirect_to @post, notice: 'Post was suc
cessfully updated.' }
        format.json { render :show, status: :ok, location: @po
st }
```

```
      else
        format.html { render :edit }
        format.json { render json: @post.errors, status: :unpr
ocessable_entity }
      end
    end
  end

  # DELETE /posts/1
  # DELETE /posts/1.json
  def destroy
    @post.destroy
    respond_to do |format|
      format.html { redirect_to posts_url, notice: 'Post was s
uccessfully destroyed.' }
      format.json { head :no_content }
    end
  end

  private
    # Use callbacks to share common setup or constraints betwe
en actions.
    def set_post
      @post = Post.find(params[:id])
    end

    # Never trust parameters from the scary internet, only all
```

```
ow the white list through.
    def post_params
      params.require(:post).permit(:body, :shark_id)
    end
end
```

Like our sharks controller, this controller's methods work with instances of the associated `Post` class. For example, the `new` method creates a new instance of the `Post` class, the `index` method grabs all instances of the class, and the `set_post` method uses `find` and `params` to select a particular post by `id`. If, however, we want our post instances to be associated with particular shark instances, then we will need to modify this code, since the `Post` class is currently operating as an independent entity.

Our modifications will make use of two things: - The methods that became available to us when we added the `belongs_to` and `has_many` associations to our models. Specifically, we now have access to the [`build` method](#) thanks to the `has_many` association we defined in our `Shark` model. This method will allow us to create a collection of post objects associated with a particular shark object, using the `shark_id` foreign key that exists in our `posts` database. - The routes and routing helpers that became available when we created a nested `posts` route. For a full list of example routes that become available when you create nested relationships between resources, see the [Rails documentation](#). For now, it will be enough for us to know that for each specific shark — say `sharks/1` — there will be an associated route for posts related to that shark: `sharks/1/posts`. There will also be routing helpers like `shark_posts_path(@shark)` and `edit_sharks_posts_path(@shark)` that refer to these nested routes.

In the file, we'll begin by writing a method, `get_shark`, that will run before each action in the controller. This method will create a local `@shark` instance variable by finding a shark instance by `shark_id`. With this variable available to us in the file, it will be possible to relate posts to a specific shark in the other methods.

Above the other `private` methods at the bottom of the file, add the following method:

~/sharkapp/controllers/posts_controller.rb

```
. . .
private
  def get_shark
    @shark = Shark.find(params[:shark_id])
  end
  # Use callbacks to share common setup or constraints between actions.
. . .
```

Next, add the corresponding filter to the **top** of the file, before the existing filter:

```
~/sharkapp/controllers/posts_controller.rb
class PostsController < ApplicationController
  before_action :get_shark
```

This will ensure that `get_shark` runs before each action defined in the file.

Next, you can use this `@shark` instance to rewrite the `index` method. Instead of grabbing all instances of the `Post` class, we want this method to return all post instances associated with a particular shark instance.

Modify the `index` method to look like this:

```
~/sharkapp/controllers/posts_controller.rb
. . .
  def index
    @posts = @shark.posts
  end
. . .
```

The `new` method will need a similar revision, since we want a new post instance to be associated with a particular shark. To achieve this, we can make use of the `build` method, along with our local `@shark` instance variable.

Change the `new` method to look like this:

```
~/sharkapp/controllers/posts_controller.rb
. . .
  def new
    @post = @shark.posts.build
  end
. . .
```

This method creates a post object that's associated with the specific shark instance from the `get_shark` method.

Next, we'll address the method that's most closely tied to `new`: `create`. The `create` method does two things: it builds a new post instance using the parameters that users have entered into the `new` form, and, if there are no errors, it saves that instance and uses a route helper to redirect users to where they can see the new post. In the case of errors, it renders the `new` template again.

Update the `create` method to look like this:

```
~/sharkapp/controllers/posts_controller.rb
  def create
    @post = @shark.posts.build(post_params)


        respond_to do |format|
         if @post.save
            format.html { redirect_to shark_posts_path(@shark)
, notice: 'Post was successfully created.' }
            format.json { render :show, status: :created, loca
tion: @post }
         else
            format.html { render :new }
            format.json { render json: @post.errors, status: :
unprocessable_entity }
      end
    end
  end
```

Next, take a look at the `update` method. This method uses a `@post` instance variable, which is not explicitly set in the method itself. Where does this variable come from?

Take a look at the filters at the top of the file. The second, auto-generated `before_action` filter provides an answer:

```
~/sharkapp/controllers/posts_controller.rb
class PostsController < ApplicationController
  before_action :get_shark
  before_action :set_post, only: [:show, :edit, :update, :destroy]
  . . .
```

The `update` method (like `show`, `edit`, and `destroy`) takes a `@post` variable from the `set_post` method. That method, listed under the `get_shark` method with our other `private` methods, currently looks like this:

```
~/sharkapp/controllers/posts_controller.rb
. . .
private
. . .
  def set_post
    @post = Post.find(params[:id])
  end
. . .
```

In keeping with the methods we've used elsewhere in the file, we will need to modify this method so that `@post` refers to a particular instance in the **collection** of posts that's associated with a particular shark. Keep the `build` method in mind here — thanks to the associations between our models, and the methods (like `build`) that are available to us by virtue of those associations, each of our post instances is part of a collection of objects that's associated with a particular shark. So it makes sense that when querying for a particular post, we would query the collection of posts associated with a particular shark.

Update `set_post` to look like this:

```
~/sharkapp/controllers/posts_controller.rb
. . .
private
. . .
  def set_post
    @post = @shark.posts.find(params[:id])
  end
. . .
```

Instead of finding a particular instance of the entire `Post` class by `id`, we instead search for a matching `id` in the collection of posts associated with a particular shark.

With that method updated, we can look at the `update` and `destroy` methods.

The `update` method makes use of the `@post` instance variable from `set_post`, and uses it with the `post_params` that the user has entered in the `edit` form. In the case of success, we want Rails to send the user back to the `ind` ex view of the posts associated with a particular shark. In the case of errors, Rails will render the `edit` template again.

In this case, the only change we will need to make is to the `redirect_to` statement, to handle successful updates. Update it to redirect to `shark_post_path(@shark)`, which will redirect to the `index` view of the selected shark's posts:

~/sharkapp/controllers/posts_controller.rb

```
. . .
  def update
    respond_to do |format|
      if @post.update(post_params)
        format.html { redirect_to shark_post_path(@shark), not
ice: 'Post was successfully updated.' }
        format.json { render :show, status: :ok, location: @po
st }
      else
        format.html { render :edit }
        format.json { render json: @post.errors, status: :unpr
ocessable_entity }
      end
    end
  end
. . .
```

Next, we will make a similar change to the `destroy` method. Update the `redirect_to` method to redirect requests to `shark_posts_path(@shark)` in the case of success:

```
~/sharkapp/controllers/posts_controller.rb
. . .
  def destroy
    @post.destroy
     respond_to do |format|
      format.html { redirect_to shark_posts_path(@shark), notice: 'Post was successfully destroyed.' }
      format.json { head :no_content }
    end
  end
. . .
```

This is the last change we will make. You now have a posts controller file that looks like this:

~/sharkapp/controllers/posts_controller.rb

```
class PostsController < ApplicationController
  before_action :get_shark
  before_action :set_post, only: [:show, :edit, :update, :dest
roy]

  # GET /posts
  # GET /posts.json
  def index
    @posts = @shark.posts
  end

  # GET /posts/1
  # GET /posts/1.json
  def show
  end

  # GET /posts/new
  def new
    @post = @shark.posts.build
  end

  # GET /posts/1/edit
  def edit
  end
```

```
  # POST /posts
  # POST /posts.json
  def create
    @post = @shark.posts.build(post_params)

        respond_to do |format|
         if @post.save
            format.html { redirect_to shark_posts_path(@shar
k), notice: 'Post was successfully created.' }
            format.json { render :show, status: :created, loca
tion: @post }
         else
            format.html { render :new }
            format.json { render json: @post.errors, status: :
unprocessable_entity }
      end
    end
  end

  # PATCH/PUT /posts/1
  # PATCH/PUT /posts/1.json
  def update
    respond_to do |format|
      if @post.update(post_params)
        format.html { redirect_to shark_post_path(@shark), not
ice: 'Post was successfully updated.' }
        format.json { render :show, status: :ok, location: @po
```

```
st }
      else
        format.html { render :edit }
        format.json { render json: @post.errors, status: :unpr
ocessable_entity }
      end
    end
  end

  # DELETE /posts/1
  # DELETE /posts/1.json
  def destroy
    @post.destroy
    respond_to do |format|
      format.html { redirect_to shark_posts_path(@shark), noti
ce: 'Post was successfully destroyed.' }
      format.json { head :no_content }
    end
  end

  private

   def get_shark
     @shark = Shark.find(params[:shark_id])
   end
    # Use callbacks to share common setup or constraints betwe
en actions.
```

```
    def set_post
      @post = @shark.posts.find(params[:id])
    end

    # Never trust parameters from the scary internet, only all
ow the white list through.
    def post_params
      params.require(:post).permit(:body, :shark_id)
    end
end
```

The controller manages how information is passed from the view templates to the database and vice versa. Our controller now reflects the relationship between our `Shark` and `Post` models, in which posts are associated with particular sharks. We can move on to modifying the view templates themselves, which are where users will pass in and modify post information about particular sharks.

Step 4 — Modifying Views

Our view template revisions will involve changing the templates that relate to posts, and also modifying our sharks `show` view, since we want users to see the posts associated with particular sharks.

Let's start with the foundational template for our posts: the `form` partial that is reused across multiple post templates. Open that form now:

```
nano app/views/posts/_form.html.erb
```

Rather than passing only the `post` model to the `form_with` form helper, we will pass both the `shark` and `post` models, with `post` set as a child resource.

Change the first line of the file to look like this, reflecting the relationship between our shark and post resources:

~/sharkapp/views/posts/_form.html.erb

```
<%= form_with(model: [@shark, post], local: true) do |form| %>
. . .
```

Next, **delete** the section that lists the `shark_id` of the related shark, since this is not essential information in the view.

The finished form, complete with our edits to the first line and without the deleted `shark_id` section, will look like this:

```
~/sharkapp/views/posts/_form.html.erb
<%= form_with(model: [@shark, post], local: true) do |form| %>
  <% if post.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(post.errors.count, "error") %> prohibi
ted this post from being saved:</h2>

      <ul>
      <% post.errors.full_messages.each do |message| %>
        <li><%= message %></li>
      <% end %>
      </ul>
    </div>
  <% end %>

  <div class="field">
    <%= form.label :body %>
    <%= form.text_area :body %>
  </div>

  <div class="actions">
    <%= form.submit %>
  </div>
<% end %>
```

Save and close the file when you are finished editing.

Next, open the `index` view, which will show the posts associated with a particular shark:

```
nano app/views/posts/index.html.erb
```

Thanks to the `rails generate scaffold` command, Rails has generated the better part of the template, complete with a table that shows the `body` field of each post and its associated `shark`.

Much like the other code we have already modified, however, this template treats posts as independent entities, when we would like to make use of the associations between our models and the collections and helper methods that these associations give us.

In the body of the table, make the following updates:

First, update `post.shark` to `post.shark.name`, so that the table will include the name field of the associated shark, rather than identifying information about the shark object itself:

```
~/sharkapp/app/views/posts/index.html.erb
. . .
  <tbody>
    <% @posts.each do |post| %>
      <tr>
        <td><%= post.body %></td>
        <td><%= post.shark.name %></td>
. . .
```

Next, change the `Show` redirect to direct users to the `show` view for the associated shark, since they will most likely want a way to navigate back to the original shark. We can make use of the `@shark` instance variable that we set in the controller here, since Rails makes instance variables created in the controller available to all views. We'll also change the text for the link from `Show` to `Show Shark`, so that users will better understand its function.

Update the this line to the following:

```
~/sharkapp/app/views/posts/index.html.erb
. . .
  <tbody>
    <% @posts.each do |post| %>
      <tr>
        <td><%= post.body %></td>
        <td><%= post.shark.name %></td>
        <td><%= link_to 'Show Shark', [@shark] %></td>
```

In the next line, we want to ensure that users are routed the right nested path when they go to edit a post. This means that rather than being directed to `posts/post_id/edit`, users will be directed to `sharks/shark_id/posts/post_id/edit`. To do this, we'll use the `shark_post_path` routing helper and our models, which Rails will treat as URLs. We'll also update the link text to make its function clearer.

Update the `Edit` line to look like the following:

~/sharkapp/app/views/posts/index.html.erb

```
. . .
  <tbody>
    <% @posts.each do |post| %>
      <tr>
        <td><%= post.body %></td>
        <td><%= post.shark.name %></td>
        <td><%= link_to 'Show Shark', [@shark] %></td>
        <td><%= link_to 'Edit Post', edit_shark_post_path(@shark, post) %></td>
```

Next, let's add a similar change to the `Destroy` link, updating its function in the string, and adding our `shark` and `post` resources:

```
~/sharkapp/app/views/posts/index.html.erb
. . .
  <tbody>
    <% @posts.each do |post| %>
      <tr>
        <td><%= post.body %></td>
        <td><%= post.shark.name %></td>
        <td><%= link_to 'Show Shark', [@shark] %></td>
        <td><%= link_to 'Edit Post', edit_shark_post_path(@sha
rk, post) %></td>
        <td><%= link_to 'Destroy Post', [@shark, post], metho
d: :delete, data: { confirm: 'Are you sure?' } %></td>
```

Finally, at the bottom of the form, we will want to update the `New Post` path to take users to the appropriate nested path when they want to create a new post. Update the last line of the file to make use of the `new_shark_post_path(@shark)` routing helper:

```
~/sharkapp/app/views/posts/index.html.erb
. . .
<%= link_to 'New Post', new_shark_post_path(@shark) %>
```

The finished file will look like this:

~/sharkapp/app/views/posts/index.html.erb

```
<p id="notice"><%= notice %></p>


<h1>Posts</h1>


<table>
  <thead>
    <tr>
      <th>Body</th>
      <th>Shark</th>
      <th colspan="3"></th>
    </tr>
  </thead>


  <tbody>
    <% @posts.each do |post| %>
      <tr>
        <td><%= post.body %></td>
        <td><%= post.shark.name %></td>
        <td><%= link_to 'Show Shark', [@shark] %></td>
        <td><%= link_to 'Edit Post', edit_shark_post_path(@sha
rk, post) %></td>
        <td><%= link_to 'Destroy Post', [@shark, post], metho
d: :delete, data: { confirm: 'Are you sure?' } %></td>
      </tr>
    <% end %>
```

```
  </tbody>
</table>

<br>

<%= link_to 'New Post', new_shark_post_path(@shark) %>
```

Save and close the file when you are finished editing.

The other edits we will make to post views won't be as numerous, since our other views use the `form` partial we have already edited. However, we will want to update the `link_to` references in the other post templates to reflect the changes we have made to our `form` partial.

Open `app/views/posts/new.html.erb`:

```
nano app/views/posts/new.html.erb
```

Update the `link_to` reference at the bottom of the file to make use of the `shark_posts_path(@shark)` helper:

```
~/sharkapp/app/views/posts/new.html.erb
. . .
<%= link_to 'Back', shark_posts_path(@shark) %>
```

Save and close the file when you are finished making this change.

Next, open the `edit` template:

```
nano app/views/posts/edit.html.erb
```

In addition to the `Back` path, we'll update `Show` to reflect our nested resources. Change the last two lines of the file to look like this:

```
~/sharkapp/app/views/posts/edit.html.erb
. . .
<%= link_to 'Show', [@shark, @post] %> |
<%= link_to 'Back', shark_posts_path(@shark) %>
```

Save and close the file.

Next, open the `show` template:

```
nano app/views/posts/show.html.erb
```

Make the following edits to the `Edit` and `Back` paths at the bottom of the file:

```
~/sharkapp/app/views/posts/edit.html.erb
. . .
<%= link_to 'Edit', edit_shark_post_path(@shark, @post) %> |
<%= link_to 'Back', shark_posts_path(@shark) %>
```

Save and close the file when you are finished.

As a final step, we will want to update the `show` view for our sharks so that posts are visible for individual sharks. Open that file now:

```
nano app/views/sharks/show.html.erb
```

Our edits here will include adding a `Posts` section to the form and an `Add Post` link at the bottom of the file.

Below the `Facts` for a given shark, we will add a new section that iterates through each instance in the collection of posts associated with this shark, outputting the `body` of each post.

Add the following code below the `Facts` section of the form, and above the redirects at the bottom of the file:

```
~/sharkapp/app/views/sharks/show.html.erb
. . .
<p>
  <strong>Facts:</strong>
  <%= @shark.facts %>
</p>

<h2>Posts</h2>
<% for post in @shark.posts %>
    <ul>
      <li><%= post.body %></li>
  </ul>
<% end %>

<%= link_to 'Edit', edit_shark_path(@shark) %> |
. . .
```

Next, add a new redirect to allow users to add a new post for this particular shark:

```
~/sharkapp/app/views/sharks/show.html.erb
. . .
<%= link_to 'Edit', edit_shark_path(@shark) %> |
<%= link_to 'Add Post', shark_posts_path(@shark) %> |
<%= link_to 'Back', sharks_path %>
```

Save and close the file when you are finished editing.

You have now made changes to your application's models, controllers, and views to ensure that posts are always associated with a particular shark. As a final step, we can add some validations to our `Post` model to guarantee consistency in the data that's saved to the database.

Step 5 — Adding Validations and Testing the Application

In [Step 5](#) of [How To Build a Ruby on Rails Application](#), you added validations to your `Shark` model to ensure uniformity and consistency in the data that gets saved to the `sharks` database. We'll now take a similar step to ensure guarantees for the `posts` database as well.

Open the file where your `Post` model is defined:

```
nano app/models/post.rb
```

Here, we want to ensure that posts are not blank and that they don't duplicate content other users may have posted. To achieve this, add the following line to the file:

```
~/sharkapp/app/models/post.rb
class Post < ApplicationRecord
  belongs_to :shark
  validates :body, presence: true, uniqueness: true
end
```

Save and close the file when you are finished editing.

With this last change in place, you are ready to run your migrations and test the application.

First, run your migrations:

```
rails db:migrate
```

Next, start your server. If you're working locally, you can do so by running:

```
rails s
```

If you are working on a development server, run the following command instead:

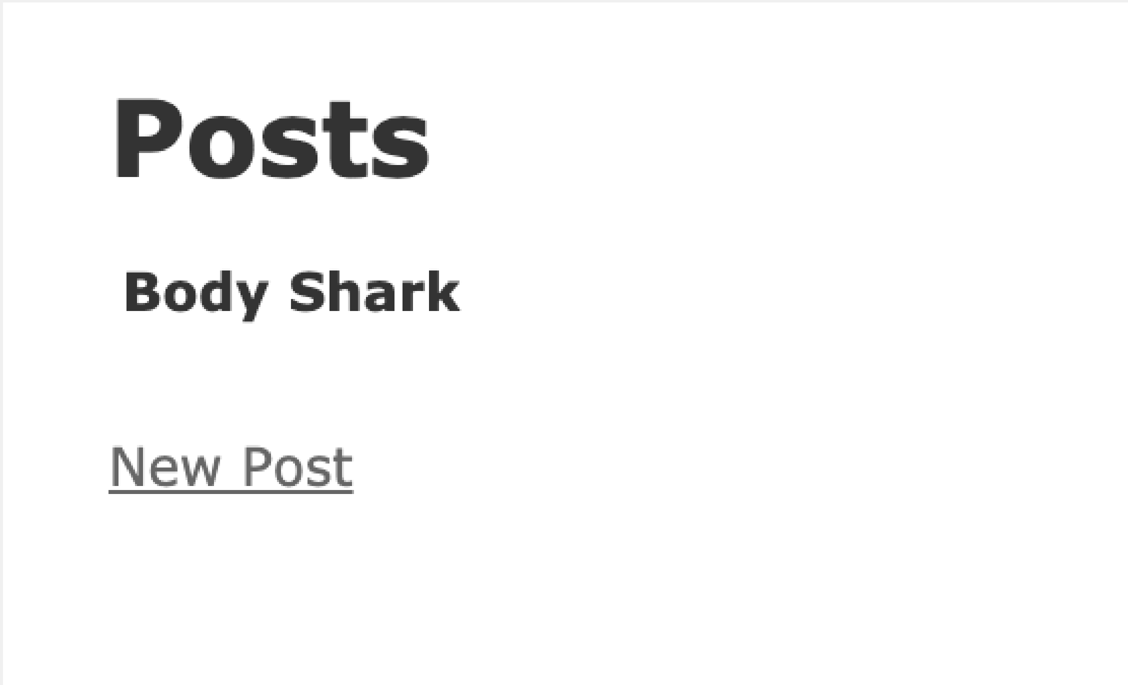
```
rails s --binding=your_server_ip
```

Navigate to your application's root at `http://localhost:3000` or `http://your_server_ip:3000`.

The prerequisite Rails project tutorial walked you through adding and editing a **Great White** shark entry. If you have not added any further sharks, the application landing page will look like this:

Click on **Show** next to the **Great White**'s name. This will take you to the `show` view for this shark. You will see the name of the shark and its facts, and a **Posts** header with no content. Let's add a post to populate this part of the form.

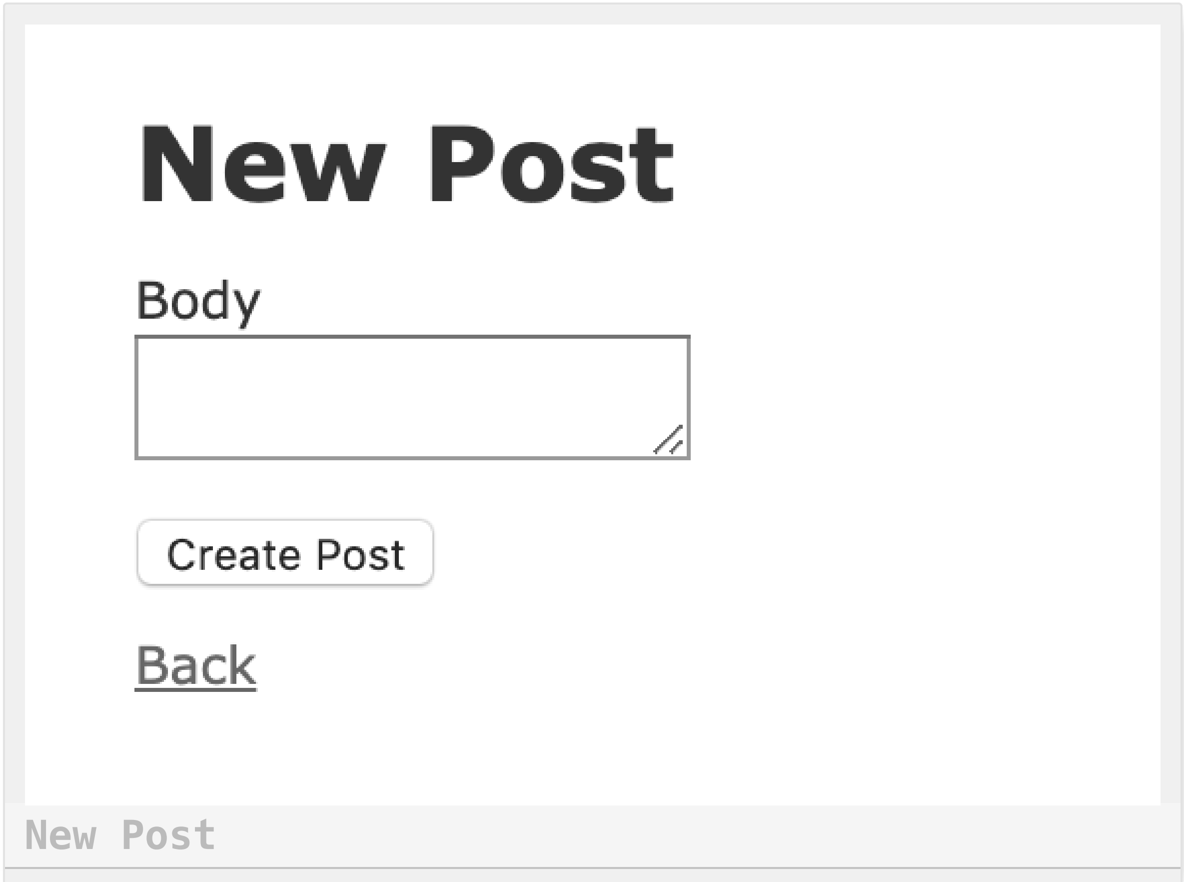
Click on **Add Post** below the **Posts** header. This will bring you to the post `index` view, where you will have the chance to select **New Post**:



Post Index View

Thanks to the authentication mechanisms you put in place in [Step 6](#) of [How To Build a Ruby on Rails Application](#), you may be asked to authenticate with the username and password you created in that Step, depending on whether or not you have created a new session.

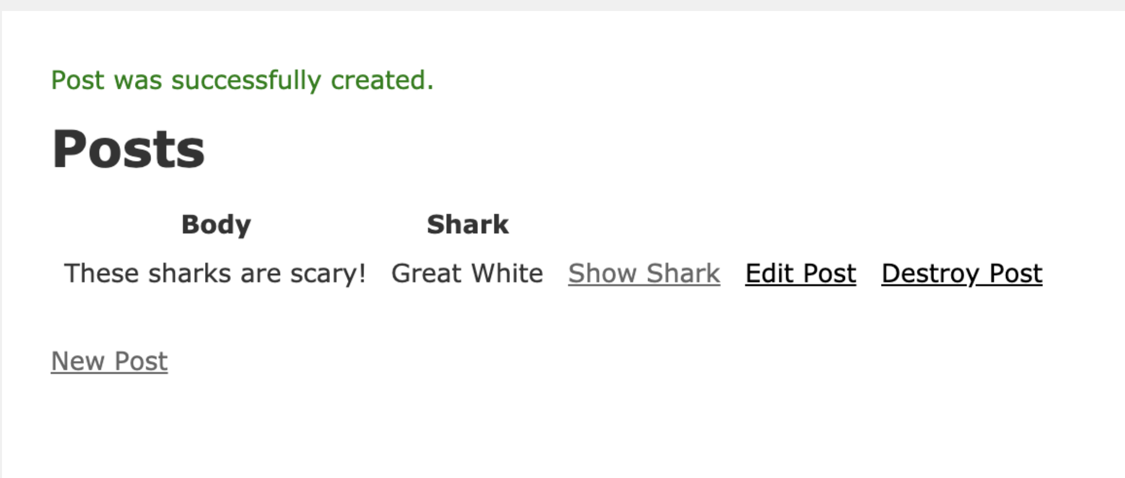
Click on **New Post**, which will bring you to your post `new` template:



In the **Body** field, type, “These sharks are scary!”

New Shark Post

Click on **Create Post**. You will be redirected to the `index` view for all posts that belong to this shark:



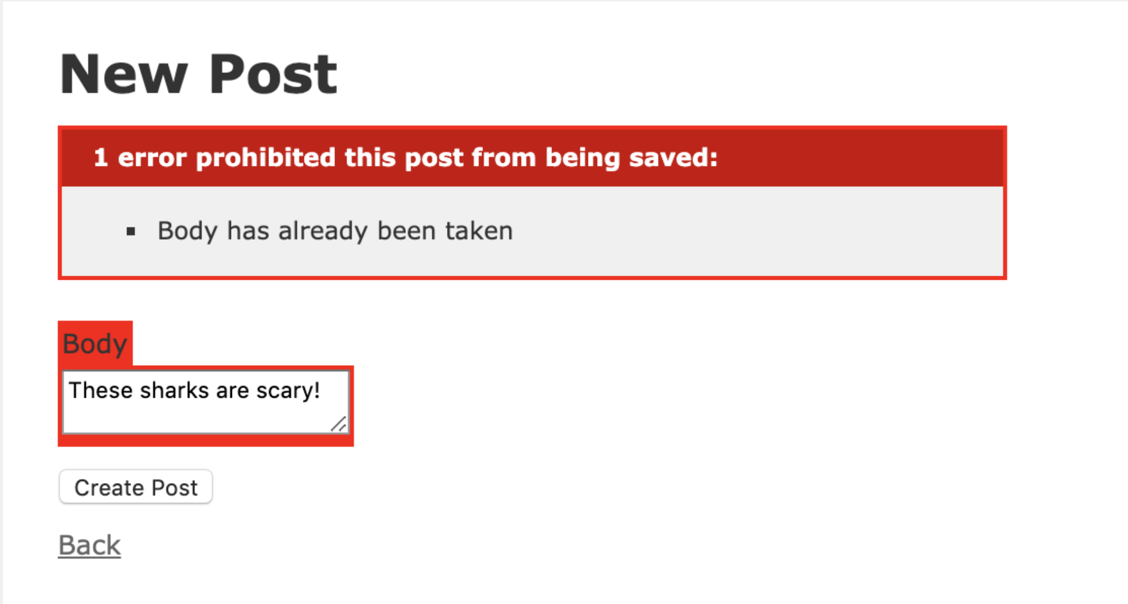
Post Success

With our post resources working, we can now test our data validations to ensure that only desired data gets saved to the database.

From the `index` view, click on **New Post**. In the **Body** field of the new form, try entering "These sharks are scary!" again:

Repeat Shark Post

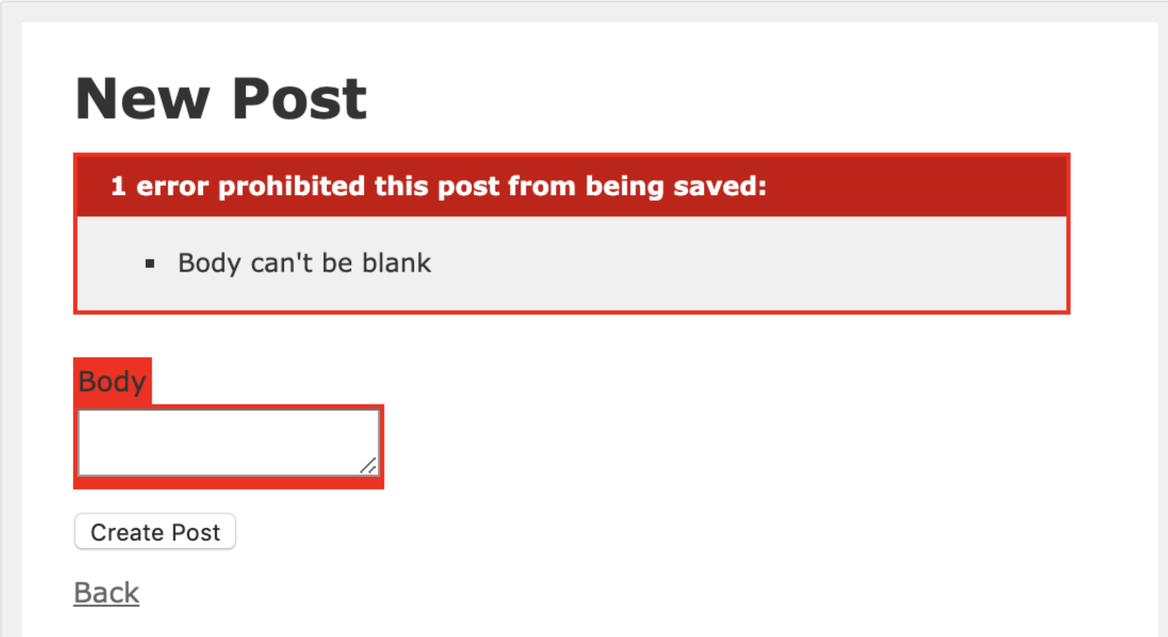
Click on **Create Post**. You will see the following error:



Unique Post Error

Click on **Back** to return to the main posts page.

To test our other validation, click on **New Post** again. Leave the post blank and click **Create Post**. You will see the following error:



Blank Post Error

With your nested resources and validations working properly, you now have a working Rails application that you can use as a starting point for further development.

Conclusion

With your Rails application in place, you can now work on things like styling and developing other front-end components. If you would like to learn more about routing and nested resources, the Rails documentation is a great place to start.

To learn more about integrating front-end frameworks with your application, take a look at How To Set Up a Ruby on Rails Project with a React Frontend.

How To Add Stimulus to a Ruby on Rails Application

Written by Kathleen Juell

If you are working with a Ruby on Rails project, your requirements may include some interactivity with the HTML generated by your view templates. If so, you have a few choices for how to implement this interactivity.

For example, you could implement a JavaScript framework like React or Ember. If your requirements include handling state on the client side, or if you are concerned about performance issues associated with frequent queries to the server, then choosing one of these frameworks may make sense. Many Single Page Applications (SPAs) take this approach.

However, there are several considerations to keep in mind when implementing a framework that manages state and frequent updates on the client side: 1. It's possible for loading and conversion requirements — things like parsing JavaScript, and fetching and converting JSON to HTML — to limit performance. 2. Commitment to a framework may involve writing more code than your particular use case requires, particularly if you are looking for small-scale JavaScript enhancements. 3. State managed on both the client and server side can lead to a duplication of efforts, and increases the surface area for errors.

As an alternative, the team at Basecamp (the same team that wrote Rails) has created Stimulus.js, which they describe as “a modest JavaScript framework for the HTML you already have." Stimulus is meant to enhance a modern Rails application by working with server-side generated HTML. State lives in the Document Object Model (DOM), and the framework offers standard ways of interacting with elements and events in the DOM. It works side by side with Turbolinks (included in Rails 5+ by default) to improve performance and load times with code that is limited and scoped to a clearly defined purpose.

In this tutorial, you will install and use Stimulus to build on an existing Rails application that offers readers information about sharks. The application already has a model for handling shark data, but you will add a nested resource for posts about individual sharks, allowing users to build out a body of thoughts and opinions about sharks. This piece runs roughly parallel to How To Create Nested Resources for a Ruby on Rails Application, except that we will be using JavaScript to manipulate the position and appearance of posts on the page. We will also take a slightly different approach to building out the post model itself.

Prerequisites

To follow this tutorial, you will need: - A local machine or development server running Ubuntu 18.04. Your development machine should have a non-root user with administrative privileges and a firewall configured with `ufw`. For instructions on how to set this up, see our Initial Server Setup with Ubuntu 18.04 tutorial. - Node.js and npm installed on your local machine or development server. This tutorial uses Node.js version <>10.16.3<> and npm version <>6.9.0<>. For guidance on installing Node.js and npm on Ubuntu

18.04, follow the instructions in the **“Installing Using a PPA”** section of How To Install Node.js on Ubuntu 18.04. - Ruby, rbenv, and Rails installed on your local machine or development server, following **Steps 1-4** in How To Install Ruby on Rails with rbenv on Ubuntu 18.04. This tutorial uses Ruby <>2.5.1<>, rbenv <>1.1.2<>, and Rails <>5.2.3<>. - SQLite installed, and a basic shark information application created, following the directions in How To Build a Ruby on Rails Application.

Step 1 — Creating a Nested Model

Our first step will be to create a nested `Post` model, which we will associate with our existing `Shark` model. We will do this by creating Active Record associations between our models: posts will belong to particular sharks, and each shark can have multiple posts.

To get started, navigate to the `sharkapp` directory that you created for your Rails project in the prerequisites:

```
cd sharkapp
```

To create our `Post` model, we'll use the `rails generate` command with the `model` generator. Type the following command to create the model:

```
rails generate model Post body:text shark:references
```

With `body:text`, we're telling Rails to include a `body` field in the `posts` database table — the table that maps to the `Post` model. We're also including the `:references` keyword, which sets up an association between the `Shark` and `Post` models. Specifically, this will ensure that a [foreign key](#) representing each shark entry in the `sharks` database is added to the `posts` database.

Once you have run the command, you will see output confirming the resources that Rails has generated for the application. Before moving on, you can check your database migration file to look at the relationship that now exists between your models and database tables. Use the following command to look at the contents of the file, making sure to substitute the timestamp on your own migration file for what's shown here:

```
cat db/migrate/20190805132506_create_posts.rb
```

You will see the following output:

```
Output
class CreatePosts < ActiveRecord::Migration[5.2]
  def change
    create_table :posts do |t|
      t.text :body
      t.references :shark, foreign_key: true

      t.timestamps
    end
  end
end
```

As you can see, the table includes a column for a shark foreign key. This key will take the form of `model_name_id` — in our case, `shark_id`.

Rails has established the relationship between the models elsewhere as well. Take a look at the newly generated `Post` model with the following command:

```
cat app/models/post.rb
```

```
Output
class Post < ApplicationRecord
  belongs_to :shark
end
```

The `belongs_to` association sets up a relationship between models in which a single instance of the declaring model belongs to a single instance of the named model. In the case of our application, this means that a single post belongs to a single shark.

Though Rails has already set the `belongs_to` association in our `Post` model, we will need to specify a `has_many` association in our `Shark` model as well in order for that relationship to function properly.

To add the `has_many` association to the `Shark` model, open `app/models/shark.rb` using `nano` or your favorite editor:

```
nano app/models/shark.rb
```

Add the following line to the file to establish the relationship between sharks and posts:

```
~/sharkapp/app/models/shark.rb
class Shark < ApplicationRecord
  has_many :posts
  validates :name, presence: true, uniqueness: true
  validates :facts, presence: true
end
```

One thing that is worth thinking about here is what happens to posts once a particular shark is deleted. We likely do not want the posts associated with a deleted shark persisting in the database. To ensure that any posts associated with a given shark are eliminated when that shark is deleted, we can include the `dependent` option with the association.

Add the following code to the file to ensure that the `destroy` action on a given shark deletes any associated posts:

```
~/sharkapp/app/models/shark.rb
class Shark < ApplicationRecord
  has_many :posts, dependent: :destroy
  validates :name, presence: true, uniqueness: true
  validates :facts, presence: true
end
```

Once you have finished making these changes, save and close the file. If you are working with `nano`, do this by pressing `CTRL+X`, `Y`, then `ENTER`.

You now have a model generated for your posts, but you will also need a controller to coordinate between the data in your database and the HTML that's generated and presented to users.

Step 2 — Creating a Controller for a Nested Resource

Creating a posts controller will involve setting a nested resource route in the application's main routing file and creating the controller file itself to specify the methods we want associated with particular actions.

To begin, open your `config/routes.rb` file to establish the relationship between your resourceful routes:

```
nano config/routes.rb
```

Currently, the file looks like this:

```
~/sharkapp/config/routes.rb
Rails.application.routes.draw do
  resources :sharks

  root 'sharks#index'
  # For details on the DSL available within this file, see htt
p://guides.rubyonrails.org/routing.html
end
```

We want to create a dependent relationship relationship between shark and post resources. To do this, update your route declaration to make `:sharks` the parent of `:posts`. Update the code in the file to look like the following:

~/sharkapp/config/routes.rb

```
Rails.application.routes.draw do
  resources :sharks do
    resources :posts
  end
  root 'sharks#index'
  # For details on the DSL available within this file, see htt
p://guides.rubyonrails.org/routing.html
end
```

Save and close the file when you are finished editing.

Next, create a new file called `app/controllers/posts_controller.rb` for the controller:

```
nano app/controllers/posts_controller.rb
```

In this file, we'll define the methods that we will use to create and destroy individual posts. However, because this is a nested model, we'll also want to create a local instance variable, `@shark`, that we can use to associate particular posts with specific sharks.

First, we can create the `PostsController` class itself, along with two `private` methods: `get_shark`, which will allow us to reference a particular shark, and `post_params`, which gives us access to user-submitted information by way of the [params method](#).

Add the following code to the file:

~/sharkapp/app/controllers/posts_controller.rb

```
class PostsController < ApplicationController
  before_action :get_shark


  private


  def get_shark
    @shark = Shark.find(params[:shark_id])
  end


  def post_params
    params.require(:post).permit(:body, :shark_id)
  end
end
```

You now have methods to get the particular shark instances with which your posts will be associated, using the `:shark_id` key, and the data that users are inputting to create posts. Both of these objects will now be available for the methods you will define to handle creating and destroying posts.

Next, above the `private` methods, add the following code to the file to define your `create` and `destroy` methods:

~/sharkapp/app/controllers/posts_controller.rb

```
. . .
  def create
    @post = @shark.posts.create(post_params)
  end

  def destroy
    @post = @shark.posts.find(params[:id])
    @post.destroy
  end
. . .
```

These methods associate `@post` instances with particular `@shark` instances, and use the [collection methods]() that became available to us when we created the `has_many` association between sharks and posts. Methods such as `find` and `create` allow us to target the collection of posts associated with a particular shark.

The finished file will look like this:

```
~/sharkapp/app/controllers/posts_controller.rb
class PostsController < ApplicationController
  before_action :get_shark


  def create
    @post = @shark.posts.create(post_params)
  end


  def destroy
    @post = @shark.posts.find(params[:id])
    @post.destroy
  end


  private


  def get_shark
    @shark = Shark.find(params[:shark_id])
  end


  def post_params
    params.require(:post).permit(:body, :shark_id)
  end
end
```

Save and close the file when you are finished editing.

With your controller and model in place, you can begin thinking about your view templates and how you will organize your application's generated HTML.

Step 3 — Reorganizing Views with Partials

You have created a `Post` model and controller, so the last thing to think about from a Rails perspective will be the views that present and allow users to input information about sharks. Views are also the place where you will have a chance to build out interactivity with Stimulus.

In this step, you will map out your views and partials, which will be the starting point for your work with Stimulus.

The view that will act as the base for posts and all partials associated with posts is the `sharks/show` view.

Open the file:

```
nano app/views/sharks/show.html.erb
```

Currently, the file looks like this:

```
~/sharkapp/app/views/sharks/show.html.erb
<p id="notice"><%= notice %></p>


<p>
  <strong>Name:</strong>
  <%= @shark.name %>
</p>


<p>
  <strong>Facts:</strong>
  <%= @shark.facts %>
</p>


<%= link_to 'Edit', edit_shark_path(@shark) %> |
<%= link_to 'Back', sharks_path %>
```

When we created our `Post` model, we opted not to generate views for our posts, since we will handle them through our `sharks/show` view. So in this view, the first thing we will address is how we will accept user input for new posts, and how we will present posts back to the user.

Note: For an alternative to this approach, please see [How To Create Nested Resources for a Ruby on Rails Application], which sets up post views using the full range of [Create, Read, Update, Delete] (CRUD) methods defined in the posts controller. For a discussion of these methods and how they work, please see [Step 3] of [How To Build a Ruby on Rails Application].

Instead of building all of our functionality into this view, we will use partials — reusable templates that serve a particular function. We will create one partial for new posts, and another to control how posts are displayed back to the user. Throughout, we'll be thinking about how and where we can use Stimulus to manipulate the appearance of posts on the page, since our goal is to control the presentation of posts with JavaScript.

First, below shark facts, add an `<h2>` header for posts and a line to render a partial called `sharks/posts`:

```
~/sharkapp/app/views/sharks/show.html.erb
. . .
<p>
  <strong>Facts:</strong>
  <%= @shark.facts %>
</p>

<h2>Posts</h2>
<%= render 'sharks/posts' %>
. . .
```

This will render the partial with the form builder for new post objects.

Next, below the `Edit` and `Back` links, we will add a section to control the presentation of older posts on the page. Add the following lines to the file to render a partial called `sharks/all`:

```
~/sharkapp/app/views/sharks/show.html.erb
<%= link_to 'Edit', edit_shark_path(@shark) %> |
<%= link_to 'Back', sharks_path %>


<div>
  <%= render 'sharks/all' %>
</div>
```

The `<div>` element will be useful when we start integrating Stimulus into this file.

Once you are finished making these edits, save and close the file. With the changes you've made on the Rails side, you can now move on to installing and integrating Stimulus into your application.

Step 4 — Installing Stimulus

The first step in using Stimulus will be to install and configure our application to work with it. This will include making sure we have the correct dependencies, including the Yarn package manager and Webpacker, the gem that will allow us to work with the JavaScript pre-processor and bundler webpack. With these dependencies in place, we will be able to install Stimulus and use JavaScript to manipulate events and elements in the DOM.

Let's begin by installing Yarn. First, update your package list:

```
sudo apt update
```

Next, add the GPG key for the Debian Yarn repository:

```
curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg | sudo apt-k
ey add -
```

Add the repository to your APT sources:

```
echo "deb https://dl.yarnpkg.com/debian/ stable main" | sudo t
ee /etc/apt/sources.list.d/yarn.list
```

Update the package database with the newly added Yarn packages:

```
sudo apt update
```

And finally, install Yarn:

```
sudo apt install yarn
```

With `yarn` installed, you can move on to adding the `webpacker` gem to your project.

Open your project's Gemfile, which lists the gem dependencies for your project:

```
nano Gemfile
```

Inside the file, you will see Turbolinks enabled by default:

```
~/sharkapp/Gemfile
. . .
# Turbolinks makes navigating your web application faster. Read more: https://github.com/turbolinks/turbolinks
gem 'turbolinks', '~> 5'
. . .
```

Turbolinks is designed to improve performance by optimizing page loads: instead of having link clicks navigate to a new page, Turbolinks intercepts these click events and makes the page request using [Asynchronous JavaScript and HTML (AJAX)](). It then replaces the body of the current page and merges the contents of the `<head>` sections, while the JavaScript `window` and `document` objects and the `<html>` element persist between renders. This addresses one of the main causes of slow page load times: the reloading of CSS and JavaScript resources.

We get Turbolinks by default in our Gemfile, but we will need to add the `webpacker` gem so that we can install and use Stimulus. Below the `turbolinks` gem, add `webpacker`:

```
~/sharkapp/Gemfile
. . .
# Turbolinks makes navigating your web application faster. Rea
d more: https://github.com/turbolinks/turbolinks
gem 'turbolinks', '~> 5'
gem 'webpacker', '~> 4.x'
. . .
```

Save and close the file when you are finished.

Next, add the gem to your project's bundle with the `bundle` command:

```
bundle
```

This will generate a new `Gemfile.lock` file — the definitive record of gems and versions for your project.

Next, install the gem in the context of your bundle with the following `bundle exec` command:

```
bundle exec rails webpacker:install
```

Once the installation is complete, we will need to make one small adjustment to our application's content security file. This is due to the fact that we are working with Rails 5.2+, which is a Content Security Policy (CSP) restricted environment, meaning that the only scripts allowed in the application must be from trusted sources.

Open `config/initializers/content_security_policy.rb`, which is the default file Rails gives us for defining application-wide security policies:

```
nano config/initializers/content_security_policy.rb
```

Add the following lines to the bottom of the file to allow `webpack-dev-server` — the server that serves our application's webpack bundle — as an allowed origin:

~/sharkapp/config/initializers/content_security_policy.rb

```
. . .
Rails.application.config.content_security_policy do |policy|
  policy.connect_src :self, :https, 'http://localhost:3035',
 'ws://localhost:3035' if Rails.env.development?
end
```

This will ensure that the `webpacker-dev-server` is recognized as a trusted asset source.

Save and close the file when you are finished making this change.

By installing `webpacker`, you created two new directories in your project's `app` directory, the directory where your main application code is located. The new parent directory, `app/javascript`, will be where your project's JavaScript code will live, and it will have the following structure:

```
Output
├── javascript
|   ├── controllers
|   |   ├── hello_controller.js
|   |   └── index.js
|   └── packs
|       └── application.js
```

The `app/javascript` directory will contain two child directories: `app/javascript/packs`, which will have your webpack entry points, and `app/javascript/controllers`, where you will define your Stimulus controllers. The `bundle exec` command that we just used will create the `app/javascript/packs` directory, but we will need to install Stimulus for the `app/javascript/controllers` directory to be autogenerated.

With `webpacker` installed, we can now install Stimulus with the following command:

```
bundle exec rails webpacker:install:stimulus
```

You will see output like the following, indicating that the installation was successful:

```
Output
. . .
success Saved lockfile.
success Saved 5 new dependencies.
info Direct dependencies
└─ stimulus@1.1.1
info All dependencies
├─ @stimulus/core@1.1.1
├─ @stimulus/multimap@1.1.1
├─ @stimulus/mutation-observers@1.1.1
├─ @stimulus/webpack-helpers@1.1.1
└─ stimulus@1.1.1
Done in 8.30s.
Webpacker now supports Stimulus.js 🎉
```

We now have Stimulus installed, and the main directories we need to work with it in place. Before moving on to writing any code, we'll need to make a few application-level adjustments to complete the installation process.

First, we'll need to make an adjustment to `app/views/layouts/application.html.erb` to ensure that our JavaScript code is available and that the code defined in our main `webpacker` entry point, `app/javascript/packs/application.js`, runs each time a page is loaded.

Open that file:

```
nano app/views/layouts/application.html.erb
```

Change the following `javascript_include_tag` tag to `javascript;pack_tag` to load `app/javascript/packs/application.js`:

```
~/sharkapp/app/views/layouts/application.html.erb
```

```
. . .
    <%= stylesheet_link_tag    'application', media: 'all', 'data-turbolinks-track': 'reload' %>
    <%= javascript_pack_tag 'application', 'data-turbolinks-track': 'reload' %>
. . .
```

Save and close the file when you have made this change.

Next, open `app/javascript/packs/application.js`:

```
nano app/javascript/packs/application.js
```

Initially, the file will look like this:

```
~/sharkapp/app/javascript/packs/application.js
```

```
. . .
console.log('Hello World from Webpacker')


import "controllers"
```

Delete the boilerplate code that's there, and add the following code to load your Stimulus controller files and boot the application instance:

~/sharkapp/app/javascript/packs/application.js

```
. . .
import { Application } from "stimulus"
import { definitionsFromContext } from "stimulus/webpack-helpe
rs"

const application = Application.start()
const context = require.context("../controllers", true, /\.js
$/)
application.load(definitionsFromContext(context))
```

This code uses webpack helper methods to require the controllers in the `app/javascript/controllers` directory and load this context for use in your application.

Save and close the file when you are finished editing.

You now have Stimulus installed and ready to use in your application. Next, we'll build out the partials that we referenced in our sharks `show` view — `sharks/posts` and `sharks/all` — using Stimulus controllers, targets, and actions.

Step 5 — Using Stimulus in Rails Partials

Our `sharks/posts` partial will use the [`form_with` form helper](#) to create a new post object. It will also make use of Stimulus's three core concepts: controllers, targets, and actions. These concepts work as follows: - Controllers are JavaScript classes that are defined in JavaScript modules and exported as the module's default object. Through controllers, you have access to particular HTML elements and the Stimulus Application instance defined in `app/javascript/packs/application.js`. - Targets allow you to reference particular HTML elements by name, and are associated with particular controllers. - Actions control how DOM events are handled by controllers, and are also associated with particular controllers. They create a connection between the HTML element associated with the controller, the methods defined in the controller, and a DOM event listener.

In our partial, we're first going to build a form as we normally would using Rails. We will then add a Stimulus controller, action, and targets to the form in order to use JavaScript to control how new posts get added to the page.

First, create a new file for the partial:

```
nano app/views/sharks/_posts.html.erb
```

Inside the file, add the following code to create a new post object using the `form_with` helper:

```
~/sharkapp/app/views/sharks/_posts.html.erb
        <%= form_with model: [@shark, @shark.posts.build] do |
form| %>
                <%= form.text_area :body, placeholder: "Your p
ost here" %>
                <br>
                <%= form.submit %>
        <% end %>
```

So far, this form behaves like a typical Rails form, using the `form_with` helper to build a post object with the fields defined for the `Post` model. Thus, the form has a field for the post `:body`, to which we've added a `placeholder` with a prompt for filling in a post.

Additionally, the form is scoped to take advantage of the collection methods that come with the associations between the `Shark` and `Post` models. In this case, the new post object that's created from user-submitted data will belong to the collection of posts associated with the shark we're currently viewing.

Our goal now is to add some Stimulus controllers, events, and actions to control how the post data gets displayed on the page. The user will ultimately submit post data and see it posted to the page thanks to a Stimulus action.

First, we'll add a controller to the form called **`posts`** in a `<div>` element:

```
~/sharkapp/app/views/sharks/_posts.html.erb
<div data-controller="posts">
        <%= form_with model: [@shark, @shark.posts.build] do |
form| %>
                 <%= form.text_area :body, placeholder: "Your
 post here" %>
                 <br>
                 <%= form.submit %>
        <% end %>
</div>
```

Make sure you add the closing `<div>` tag to scope the controller properly.

Next, we'll attach an action to the form that will be triggered by the form submit event. This action will control how user input is displayed on the page. It will reference an `addPost` method that we will define in the posts Stimulus controller:

```
~/sharkapp/app/views/sharks/_posts.html.erb
<div data-controller="posts">
        <%= form_with model: [@shark, @shark.posts.build], dat
a: { action: "posts#addBody" } do |form| %>
        . . .
                 <%= form.submit %>
        <% end %>
</div>
```

We use the `:data` option with `form_with` to submit the Stimulus action as an additional HTML data attribute. The action itself has a value called an action descriptor made up of the following: - **The DOM event to listen for**. Here, we are using the default event associated with form elements, submit, so we do not need to specify the event in the descriptor itself. For more information about common element/event pairs, see the [Stimulus documentation](). - **The controller identifier**, in our case `posts`. - **The method that the event should invoke**. In our case, this is the `addBody` method that we will define in the controller.

Next, we'll attach a data target to the user input defined in the `:body` `<textarea>` element, since we will use this inputted value in the `addBody` method.

Add the following `:data` option to the `:body` `<textarea>` element:

~/sharkapp/app/views/sharks/_posts.html.erb

```
<div data-controller="posts">
        <%= form_with model: [@shark, @shark.posts.build], data: { action: "posts#addBody" } do |form| %>
                <%= form.text_area :body, placeholder: "Your post here", data: { target: "posts.body" } %>
. . .
```

Much like action descriptors, Stimulus targets have target descriptors, which include the controller identifier and the target name. In this case, `posts` is our controller, and `body` is the target itself.

As a last step, we'll add a data target for the inputted `body` values so that users will be able to see their posts as soon as they are submitted.

Add the following `<ul>` element with an `add` target below the form and above the closing `<div>`:

```
~/sharkapp/app/views/sharks/_posts.html.erb
. . .
        <% end %>
  <ul data-target="posts.add">
  </ul>

</div>
```

As with the `body` target, our target descriptor includes both the name of the controller and the target — in this case, `add`.

The finished partial will look like this:

~/sharkapp/app/views/sharks/_posts.html.erb

```
<div data-controller="posts">
        <%= form_with model: [@shark, @shark.posts.build], data: { action: "posts#addBody"} do |form| %>
                <%= form.text_area :body, placeholder: "Your post here", data: { target: "posts.body" } %>
                <br>
                <%= form.submit %>
        <% end %>
  <ul data-target="posts.add">
  </ul>

</div>
```

Once you have made these changes, you can save and close the file.

You have now created one of the two partials you added to the `sharks/show` view template. Next, you'll create the second, `sharks/all`, which will show all of the older posts from the database.

Create a new file named `_all.html.erb` in the `app/views/sharks/` directory:

```
nano app/views/sharks/_all.html.erb
```

Add the following code to the file to iterate through the collection of posts associated with the selected shark:

```
~/sharkapp/app/views/sharks/_all.html.erb
<% for post in @shark.posts  %>
    <ul>

        <li class="post">
            <%= post.body %>
        </li>

    </ul>
    <% end %>
```

This code uses a for loop to iterate through each post instance in the collection of post objects associated with a particular shark.

We can now add some Stimulus actions to this partial to control the appearance of posts on the page. Specifically, we will add actions that will control upvotes and whether or not posts are visible on the page

Before we do that, however, we will need to add a gem to our project so that we can work with [Font Awesome]() icons, which we'll use to register upvotes. Open a second terminal window, and navigate to your `sharkapp` project directory.

Open your Gemfile:

```
[environment second]
nano Gemfile
```

Below your `webpacker` gem, add the following line to include the [font-awesome-rails gem](#) in the project:

```
~/sharkapp/Gemfile
[environment second]
. . .
gem 'webpacker', '~> 4.x'
gem 'font-awesome-rails', '~>4.x'
. . .
```

Save and close the file.

Next, install the gem:

```
[environment second]
bundle install
```

Finally, open your application's main stylesheet, `app/assets/stylesheets/application.css`:

```
[environment second]
nano app/assets/stylesheets/application.css
```

Add the following line to include Font Awesome's styles in your project:

```
~/sharkapp/app/assets/stylesheets/application.css
```

```
[environment second]
. . .
*
 *= require_tree .
 *= require_self
 *= require font-awesome
 */
```

Save and close the file. You can now close your second terminal window.

Back in your `app/views/sharks/_all.html.erb` partial, you can now add two [`button_tags`]() with associated Stimulus actions, which will be triggered on click events. One button will give users the option to upvote a post and the other will give them the option to remove it from the page view.

Add the following code to `app/views/sharks/_all.html.erb`:

```
~/sharkapp/app/views/sharks/_all.html.erb
<% for post in @shark.posts  %>
    <ul>


        <li class="post">
            <%= post.body %>
            <%= button_tag "Remove Post", data: { controller: "posts", action: "posts#remove" } %>
            <%= button_tag "Upvote Post", data: { controller: "posts", action: "posts#upvote" } %>
        </li>


    </ul>
    <% end %>
```

Button tags also take a `:data` option, so we've added our posts Stimulus controller and two actions: `remove` and `upvote`. Once again, in the action descriptors, we only need to define our controller and method, since the default event associated with button elements is click. Clicking on each of these buttons will trigger the respective `remove` and `upvote` methods defined in our controller.

Save and close the file when you have finished editing.

The final change we will make before moving on to defining our controller is to set a data target and action to control how and when the `sharks/all` partial will be displayed.

Open the `show` template again, where the initial call to render `sharks/all` is currently defined:

```
nano app/views/sharks/show.html.erb
```

At the bottom of the file, we have a `<div>` element that currently looks like this:

```
~/sharkapp/app/views/sharks/show.html.erb
. . .
<div>
  <%= render 'sharks/all' %>
</div>
```

First, add a controller to this `<div>` element to scope actions and targets:

```
~/sharkapp/app/views/sharks/show.html.erb
. . .
<div data-controller="posts">
  <%= render 'sharks/all' %>
</div>
```

Next, add a button to control the appearance of the partial on the page. This button will trigger a `showAll` method in our posts controller.

Add the button below the `<div>` element and above the `render` statement:

```
~/sharkapp/app/views/sharks/show.html.erb
. . .
<div data-controller="posts">


<button data-action="posts#showAll">Show Older Posts</button>


  <%= render 'sharks/all' %>
```

Again, we only need to identify our `posts` controller and `showAll` method here — the action will be triggered by a click event.

Next, we will add a data target. The goal of setting this target is to control the appearance of the partial on the page. Ultimately, we want users to see older posts only if they have opted into doing so by clicking on the `Show Older Posts` button.

We will therefore attach a data target called `show` to the `sharks/all` partial, and set its default style to [`visibility:hidden`](). This will hide the partial unless users opt in to seeing it by clicking on the button.

Add the following `<div>` element with the `show` target and `style` definition below the button and above the partial render statement:

```
~/sharkapp/app/views/sharks/show.html.erb
. . .
<div data-controller="posts">


<button data-action="posts#showAll">Show Older Posts</button>


<div data-target="posts.show" style="visibility:hidden">
  <%= render 'sharks/all' %>
</div>
```

Be sure to add the closing `</div>` tag.

The finished `show` template will look like this:

~/sharkapp/app/views/sharks/show.html.erb

```
<p id="notice"><%= notice %></p>

<p>
  <strong>Name:</strong>
  <%= @shark.name %>
</p>

<p>
  <strong>Facts:</strong>
  <%= @shark.facts %>
</p>

<h2>Posts</h2>

<%= render 'sharks/posts' %>

<%= link_to 'Edit', edit_shark_path(@shark) %> |
<%= link_to 'Back', sharks_path %>

<div data-controller="posts">

<button data-action="posts#showAll">Show Older Posts</button>

<div data-target="posts.show" style="visibility:hidden">
  <%= render 'sharks/all' %>
```

```
</div>

</div>
```

Save and close the file when you are finished editing.

With this template and its associated partials finished, you can move on to creating the controller with the methods you've referenced in these files.

Step 6 — Creating the Stimulus Controller

Installing Stimulus created the `app/javascript/controllers` directory, which is where webpack is loading our application context from, so we will create our posts controller in this directory. This controller will include each of the methods we referenced in the previous step: - `addBody()`, to add new posts. - `showAll()`, to show older posts. - `remove()`, to remove posts from the current view. - `upvote()`, to attach an upvote icon to posts.

Create a file called `posts_controller.js` in the `app/javascript/controllers` directory:

```
nano app/javascript/controllers/posts_controller.js
```

First, at the top of the file, extend Stimulus's built-in `Controller` class:

~/sharkapp/app/javascript/controllers/posts_controller.js

```
import { Controller } from "stimulus"


export default class extends Controller {
}
```

Next, add the following target definitions to the file:

~/sharkapp/app/javascript/controllers/posts_controller.js

```
. . .
export default class extends Controller {
    static targets = ["body", "add", "show"]
}
```

Defining targets in this way will allow us to access them in our methods with the `this.target-nameTarget` property, which gives us the first matching target element. So, for example, to match the `body` data target defined in our targets array, we would use `this.bodyTarget`. This property allows us to manipulate things like input values or css styles.

Next, we can define the `addBody` method, which will control the appearance of new posts on the page. Add the following code below the target definitions to define this method:

```
~/sharkapp/app/javascript/controllers/posts_controller.js
```

```
. . .
export default class extends Controller {
    static targets = [ "body", "add", "show"]

    addBody() {
        let content = this.bodyTarget.value;
        this.addTarget.insertAdjacentHTML('beforebegin', "<li>" + content + "</li>");
    }
}
```

This method defines a `content` variable with the let keyword and sets it equal to the post input string that users entered into the posts form. It does this by virtue of the `body` data target that we attached to the `<textarea>` element in our form. Using `this.bodyTarget` to match this element, we can then use the value property that is associated with that element to set the value of `content` as the post input users have entered.

Next, the method adds this post input to the `add` target we added to the `<ul>` element below the form builder in the `sharks/posts` partial. It does this using the `Element.insertAdjacentHTML()` method, which will insert the content of the new post, set in the `content` variable, before the `add` target element. We've also enclosed the new post in an `<li>` element, so that new posts appear as bulleted list items.

Next, below the `addBody` method, we can add the `showAll` method, which will control the appearance of older posts on the page:

~/sharkapp/app/javascript/controllers/posts_controller.js

```
. . .
export default class extends Controller {
. . .
    addBody() {
        let content = this.bodyTarget.value;
        this.addTarget.insertAdjacentHTML('beforebegin', "<li>" + content + "</li>");
    }

    showAll() {
        this.showTarget.style.visibility = "visible";
    }

}
```

Here, we again use the `this.target-nameTarget` property to match our `show` target, which is attached to the `<div>` element with the `sharks/all` partial. We gave it a default style, `"visibility:hidden"`, so in this method, we simply change the style to `"visible"`. This will show the partial to users who have opted into seeing older posts.

Below `showAll`, we'll next add an `upvote` method, to allow users to "upvote" posts on the page by attaching the free Font Awesome `check-circle` icon to a particular post.

Add the following code to define this method:

~/sharkapp/app/javascript/controllers/posts_controller.js

```
. . .
export default class extends Controller {
. . .

    showAll() {
        this.showTarget.style.visibility = "visible";
    }

    upvote() {
        let post = event.target.closest(".post");
        post.insertAdjacentHTML('beforeend', '<i class="fa fa-check-circle"></i>');
    }

}
```

Here, we're creating a `post` variable that will target the closest `<li>` element with the class `post` — the class we attached to each `<li>` element in our loop iteration in `sharks/all`. This will target the closest post and add the `check-circle` icon just inside `<li>` element, after its last child.

Next, we'll use a similar method to hide posts on the page. Add the following code below the `upvote` method to define a `remove` method:

~/sharkapp/app/javascript/controllers/posts_controller.js

```
. . .
export default class extends Controller {
. . .

    upvote() {
        let post = event.target.closest(".post");
        post.insertAdjacentHTML('beforeend', '<i class="fa fa-check-circle"></i>');
    }

    remove() {
        let post = event.target.closest(".post");
        post.style.visibility = "hidden";
    }
}
```

Once again, our `post` variable will target the closest `<li>` element with the class `post`. It will then set the visibility property to `"hidden"` to hide the post on the page.

The finished controller file will now look like this:

~/sharkapp/app/javascript/controllers/posts_controller.js

```
import { Controller } from "stimulus"

export default class extends Controller {

    static targets = ["body", "add", "show"]

    addBody() {
        let content = this.bodyTarget.value;
        this.addTarget.insertAdjacentHTML('beforebegin', "<li
>" + content + "</li>");
    }

    showAll() {
        this.showTarget.style.visibility = "visible";
    }

    upvote() {
        let post = event.target.closest(".post");
        post.insertAdjacentHTML('beforeend', '<i class="fa fa-
check-circle"></i>');
    }

    remove() {
        let post = event.target.closest(".post");
```

```
        post.style.visibility = "hidden";
    }
}
```

Save and close the file when you are finished editing.

With your Stimulus controller in place, you can move on to making some final changes to the `index` view and testing your application.

Step 7 — Modifying the Index View and Testing the Application

With one final change to the sharks `index` view you will be ready to test out your application. The `index` view is the root of the application, which you set in [Step 4](#) of [How To Build a Ruby on Rails Application](#).

Open the file:

```
nano app/views/sharks/index.html.erb
```

In place of the `link_to` helpers that were autogenerated for us to display and destroy sharks, we'll use `button_to` helpers. This will help us work with generated HTML code instead of the default Rails JavaScript assets, which we specified we would no longer use in Step 1, when we changed `javascript_include_tag` to `javascript_pack_tag` in `app/views/layouts/application.html.erb`.

Replace the existing `link_to` helpers in the file with the following `button_to` helpers:

~/sharkapp/app/views/sharks/index.html.erb

```
. . .
  <tbody>
    <% @sharks.each do |shark| %>
      <tr>
        <td><%= shark.name %></td>
        <td><%= shark.facts %></td>
        <td><%= button_to 'Show', shark_path(:id => shark.id), :method => :get %></td>
        <td><%= button_to 'Edit', edit_shark_path(:id => shark.id), :method => :get %></td>
        <td><%= button_to 'Destroy', shark_path(:id => shark.id), :method => :delete %></td>
      </tr>
    <% end %>
  </tbody>
. . .
```

These helpers accomplish much the same things as their `link_to` counterparts, but the `Destroy` helper now relies on generated HTML rather than Rails's default JavaScript.

Save and close the file when you are finished editing.

You are now ready to test your application.

First, run your database migrations:

```
rails db:migrate
```

Next, start your server. If you are working locally, you can do this with the following command:

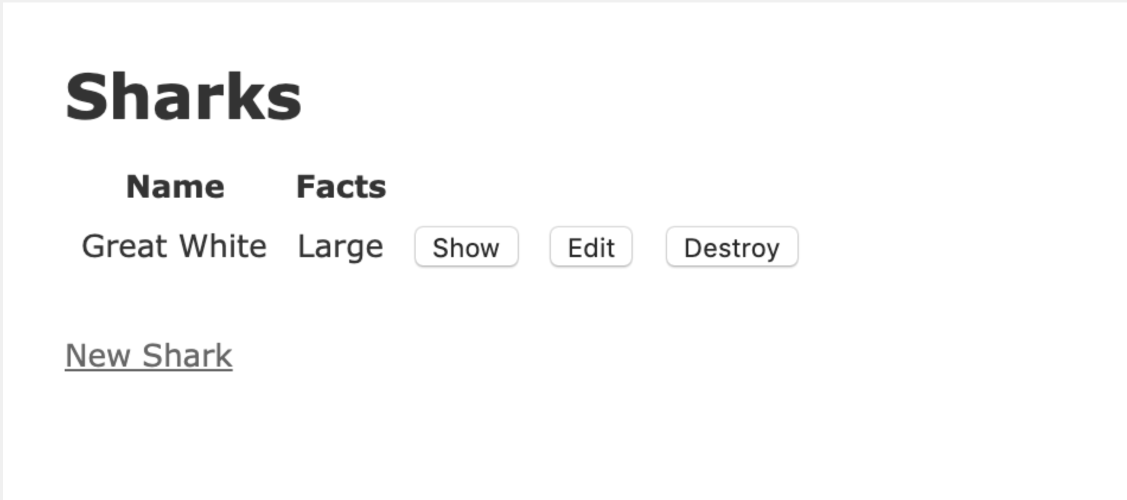
```
rails s
```

If you are working on a development server, you can start the application with:

```
rails s --binding=your_server_ip
```

Navigate to the application landing page in your browser. If you are working locally, this will be `localhost:3000`, or `http://your_server_ip:3000` if you are working on a server.

You will see the following landing page:



Application Landing Page

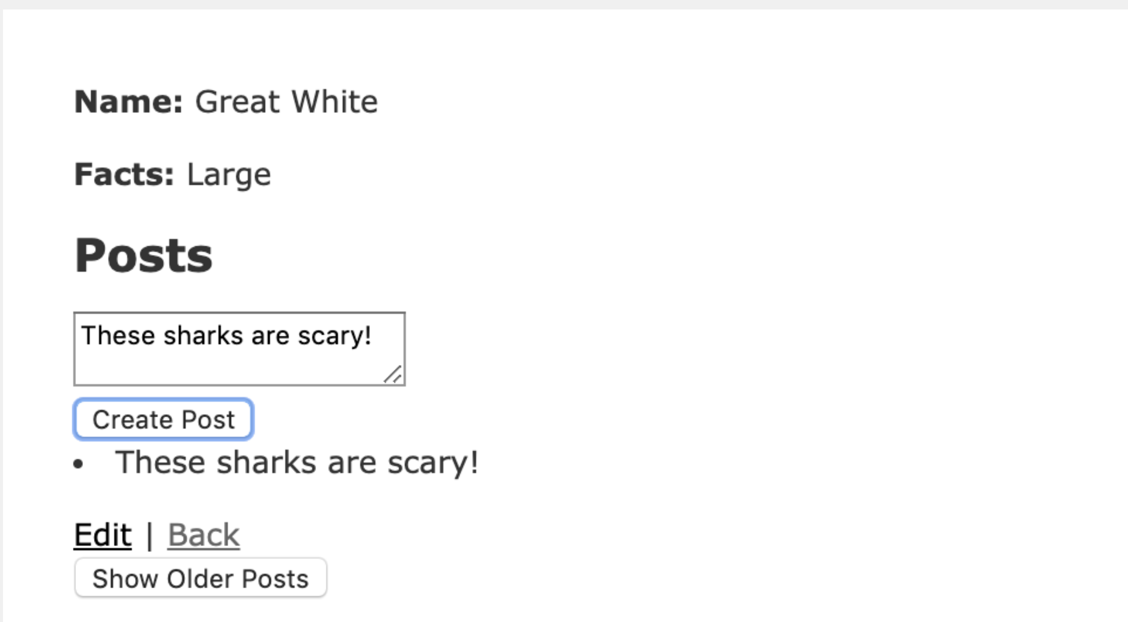
Clicking on **Show** will take you to the `show` view for this shark. Here you will see a form to fill out a post:

Shark Show Page

In the post form, type "These sharks are scary!":

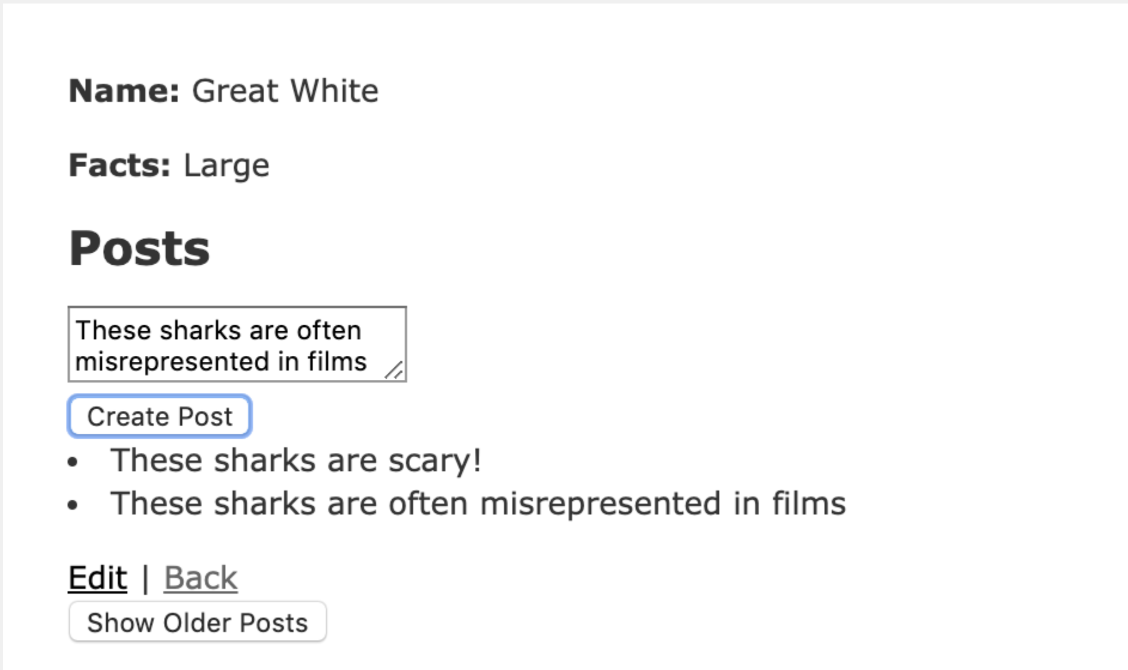
Filled in Post

Click on **Create Post**. You will now see the new post on the page:



New Post Added to Page

You can add another new post, if you would like. This time, type "These sharks are often misrepresented in films" and click **Create Post**:



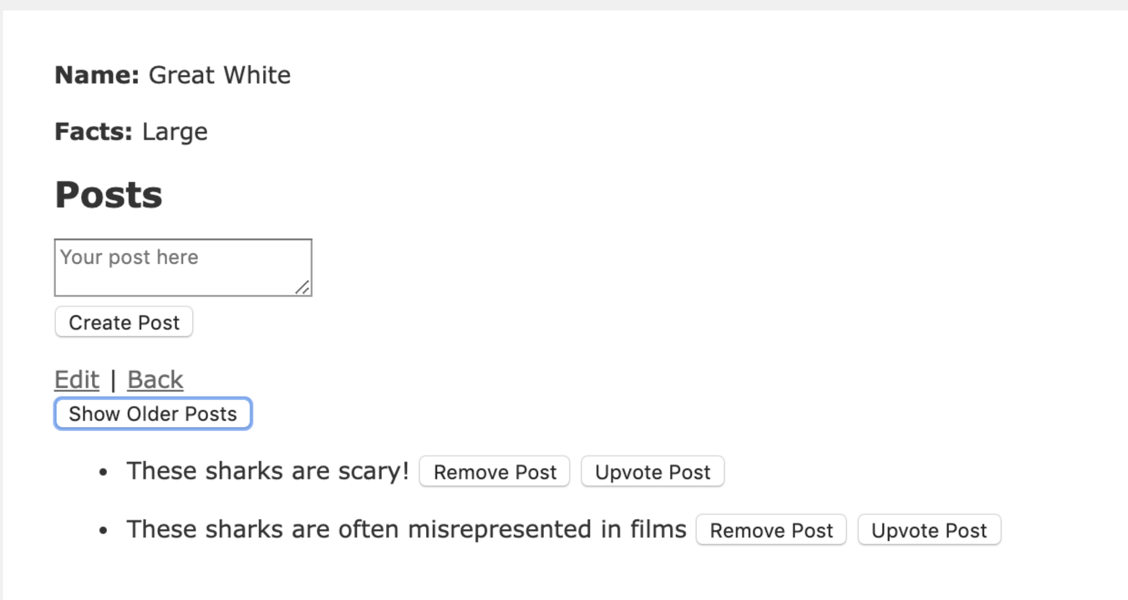
Second Post Added to Page

In order to test the functionality of the **Show Older Posts** feature, we will need to leave this page, since our Great White does not currently have any posts that are older than the ones we've just added.

Click **Back** to get to the main page, and then **Show** to return to the Great White landing page:

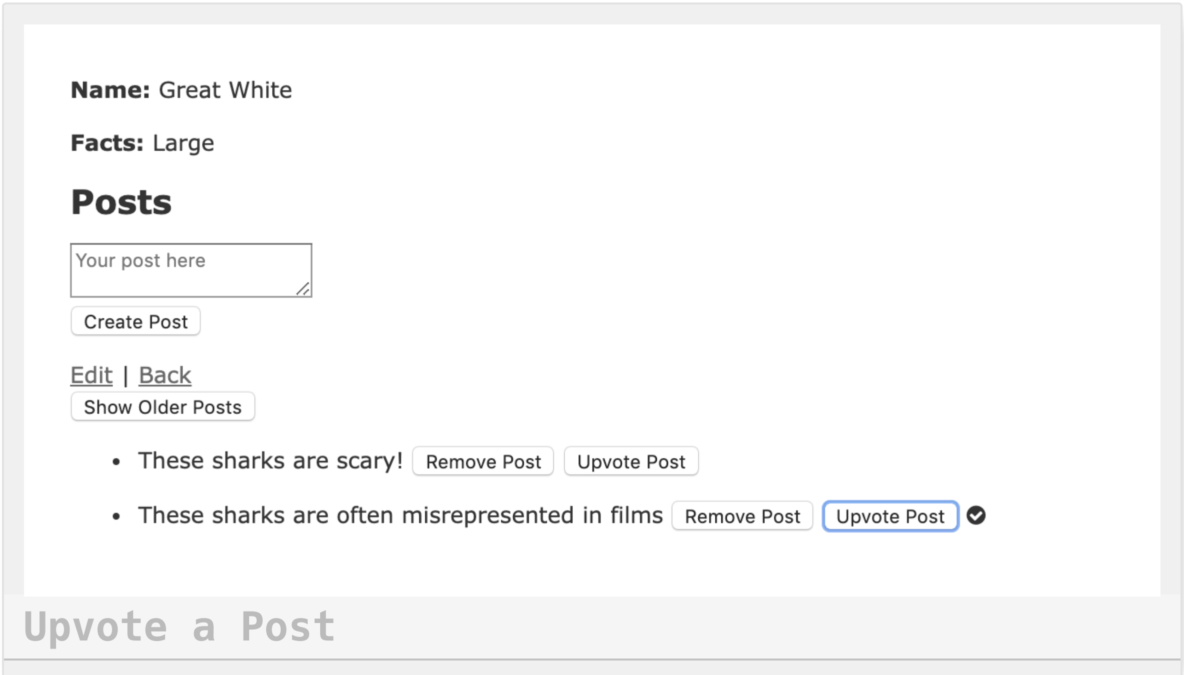
Shark Show Page

Clicking on **Show Older Posts** will now show you the posts you created:

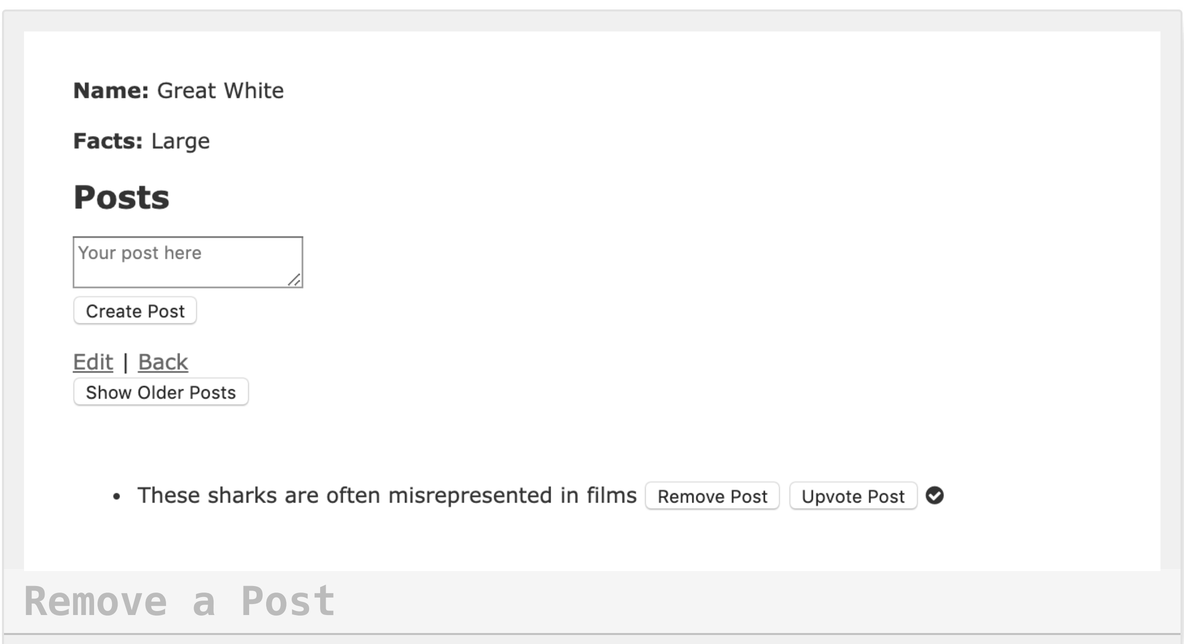


Show Older Posts

You can now upvote a post by clicking on the **Upvote Post** button:



Similarly, clicking **Remove Post** will hide the post:



You have now confirmed that you have a working Rails application that uses Stimulus to control how nested post resources are displayed on individual shark pages. You can use this as the jumping off point for future development and experimentation with Stimulus.

Conclusion

Stimulus represents a possible alternative to working with rails-ujs, JQuery, and frameworks like React and Vue.

As discussed in the introduction, Stimulus makes the most sense when you need to work directly with HTML generated by the server. It is lightweight, and aims to make code – particularly HTML – self-explanatory to the highest degree possible. If you don't need to manage state on the client side, then Stimulus may be a good choice.

If you are interested in how to create nested resources without a Stimulus integration, you can consult How To Create Nested Resources for a Ruby on Rails Application.

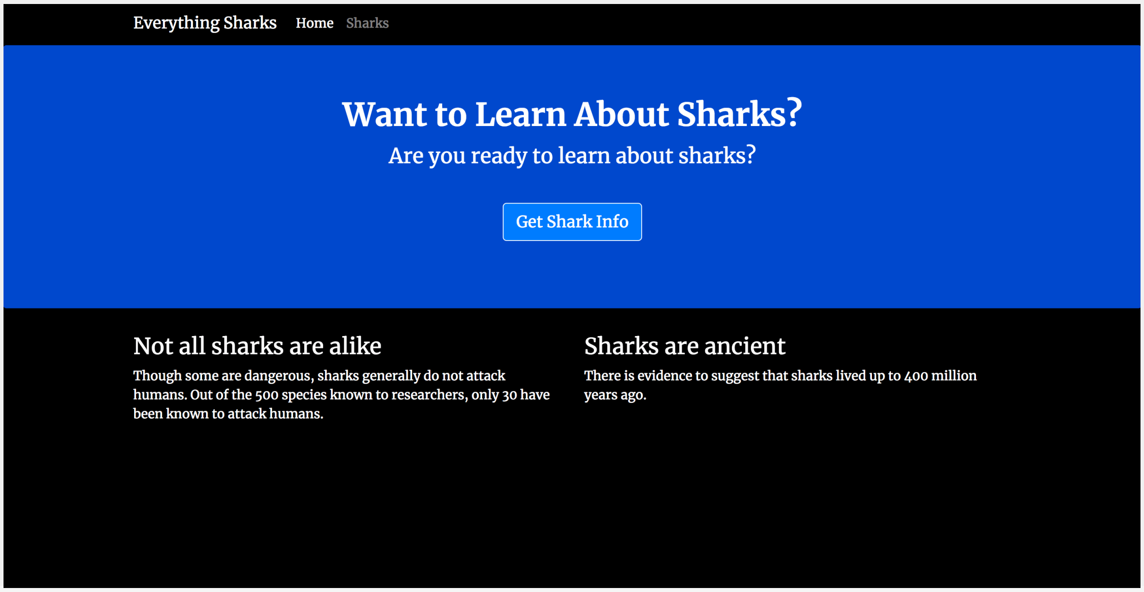
For more information on how you would integrate React with a Rails application, see How To Set Up a Ruby on Rails Project with a React Frontend.

How To Add Bootstrap to a Ruby on Rails Application

Written by Kathleen Juell

If you are developing a Ruby on Rails application, you may be interested in adding styles to your project to facilitate user engagement. One way to do this is by adding Bootstrap, an HTML, CSS, and JavaScript framework designed to simplify the process of making web projects responsive and mobile ready. By implementing Bootstrap in a Rails project, you can integrate its layout conventions and components into your application to make user interactions with your site more engaging.

In this tutorial, you will add Bootstrap to an existing Rails project that uses the webpack bundler to serve its JavaScript and CSS assets. The goal will be to create a visually appealing site that users can interact with to share information about sharks:



Application Landing Page

Prerequisites

To follow this tutorial, you will need: - A local machine or development server running Ubuntu 18.04. Your development machine should have a non-root user with administrative privileges and a firewall configured with `ufw`. For instructions on how to set this up, see our Initial Server Setup with Ubuntu 18.04 tutorial. - Node.js and npm installed on your local machine or development server. This tutorial uses Node.js version <>10.16.3<> and npm version <>6.9.0<>. For guidance on installing Node.js and npm on Ubuntu 18.04, follow the instructions in the **"Installing Using a PPA"** section of How To Install Node.js on Ubuntu 18.04. - Ruby, rbenv, and Rails installed on your local machine or development server, following **Steps 1-4** in How To Install Ruby on Rails with rbenv on Ubuntu 18.04. This tutorial uses Ruby <>2.5.1<>, rbenv <>1.1.2<>, and Rails <>5.2.3<>. - SQLite installed, following

Step 1 of How To Build a Ruby on Rails Application. This tutorial uses SQLite 3 <>3.22.0<>.

Step 1 — Cloning the Project and Installing Dependencies

Our first step will be to clone the rails-stimulus repository from the DigitalOcean Community GitHub account. This repository includes the code from the setup described in How To Add Stimulus to a Ruby on Rails Application, which described how to add Stimulus.js to an existing Rails 5 project.

Clone the repository into a directory called `rails-bootstrap`:

```
git clone https://github.com/do-community/rails-stimulus.git rails-bootstrap
```

Navigate to the `rails-bootstrap` directory:

```
cd rails-bootstrap
```

In order to work with the project code, you will first need to install the project's dependencies, which are listed in its Gemfile. Use the following command to install the required gems:

```
bundle install
```

Next, you will install your Yarn dependencies. Because this Rails 5 project has been modified to serve assets with webpack, its JavaScript dependencies are now managed by Yarn. This means that it's necessary to install and verify the dependencies listed in the project's `package.json` file.

Run the following command to install these dependencies:

```
yarn install --check-files
```

The `--check-files` flag checks to make sure that any files already installed in the `node_modules` directory have not been removed.

Next, run your database migrations:

```
rails db:migrate
```

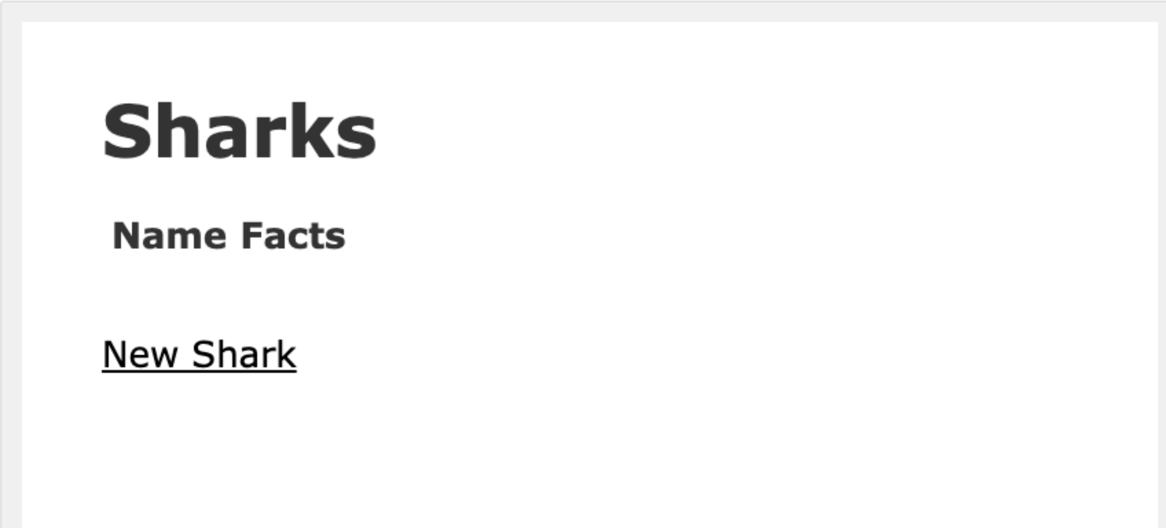
Once your migrations have finished, you can test the application to ensure that it is working as expected. Start your server with the following command if you are working locally:

```
rails s
```

If you are working on a development server, you can start the application with:

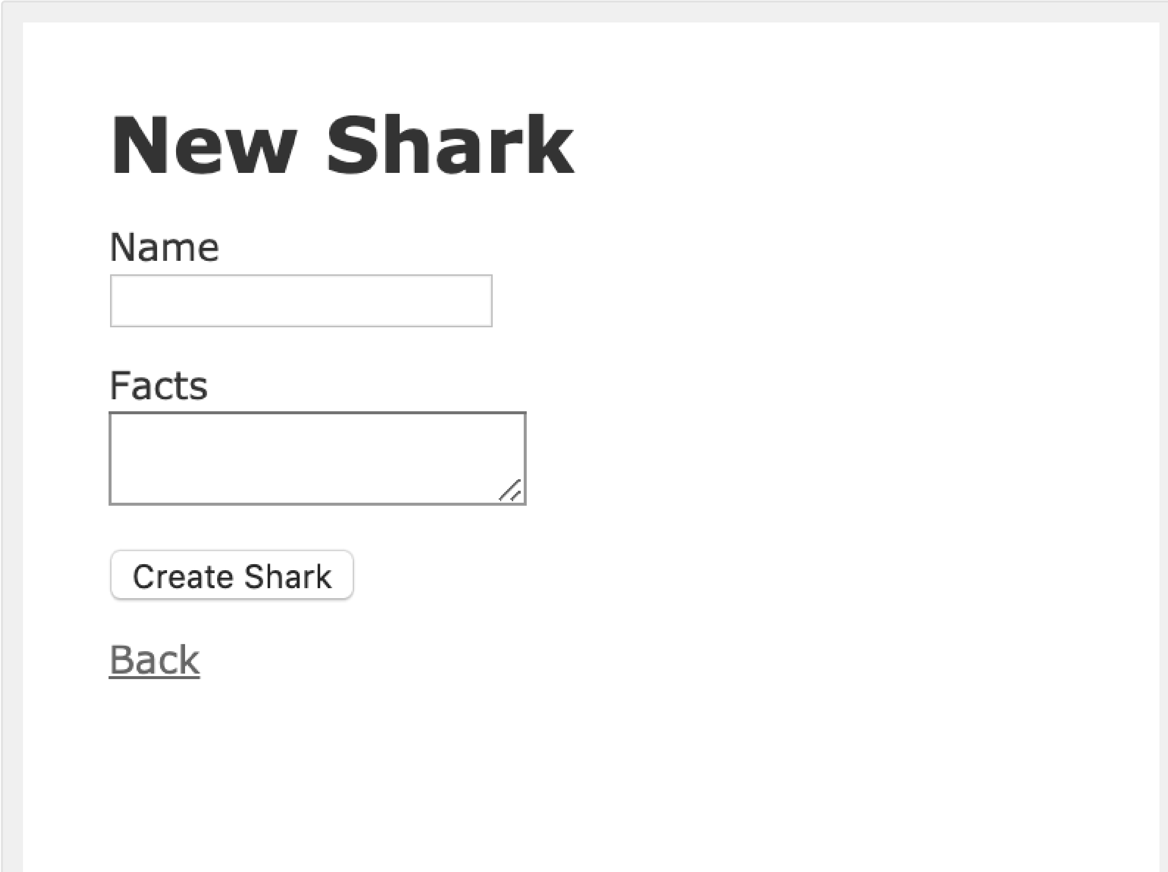
```
rails s --binding=your_server_ip
```

Navigate to `localhost:3000` or `http://your_server_ip:3000`. You will see the following landing page:



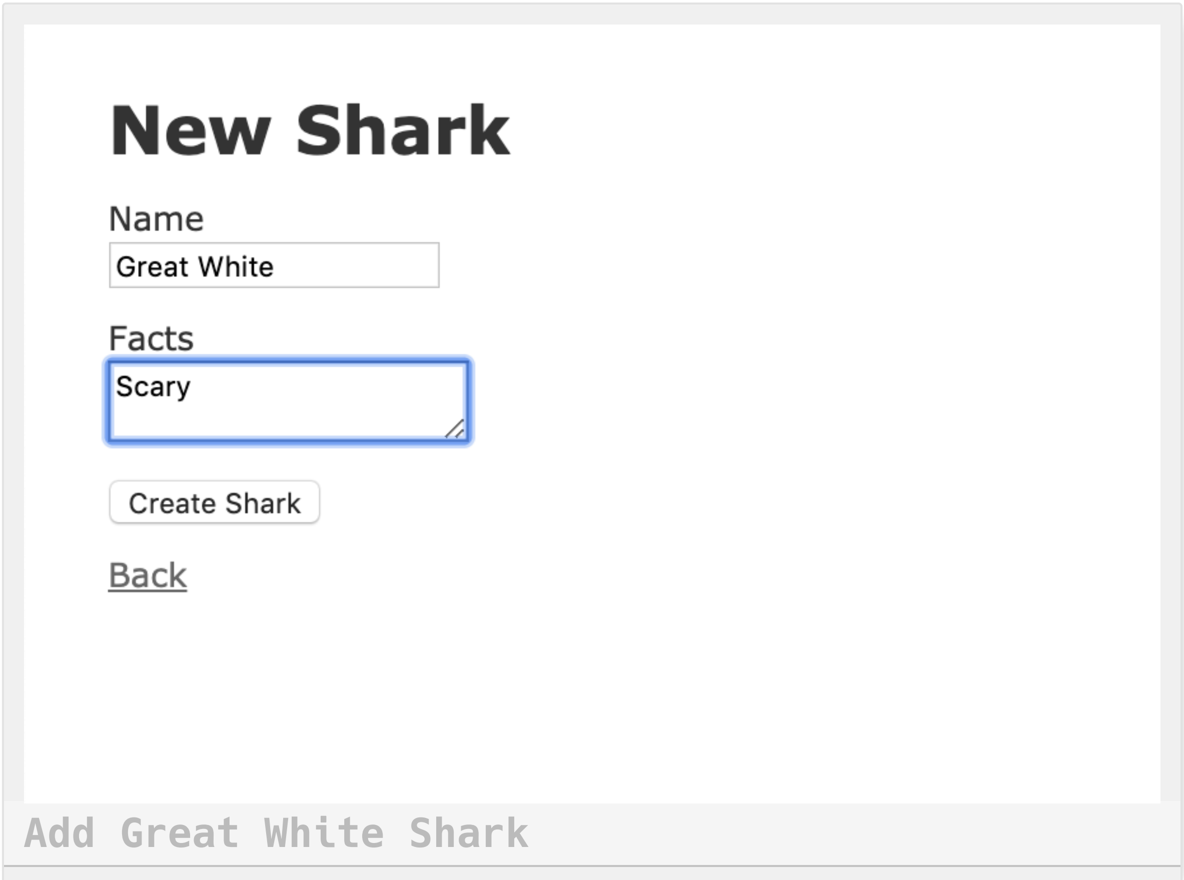
Application Landing Page

To create a new shark, click on the **New Shark** link at the bottom of the page, which will take you to the `sharks/new` route. You will be prompted for a username (**sammy**) and password (**shark**), thanks to the project's [authentication settings](). The `new` view looks like this:

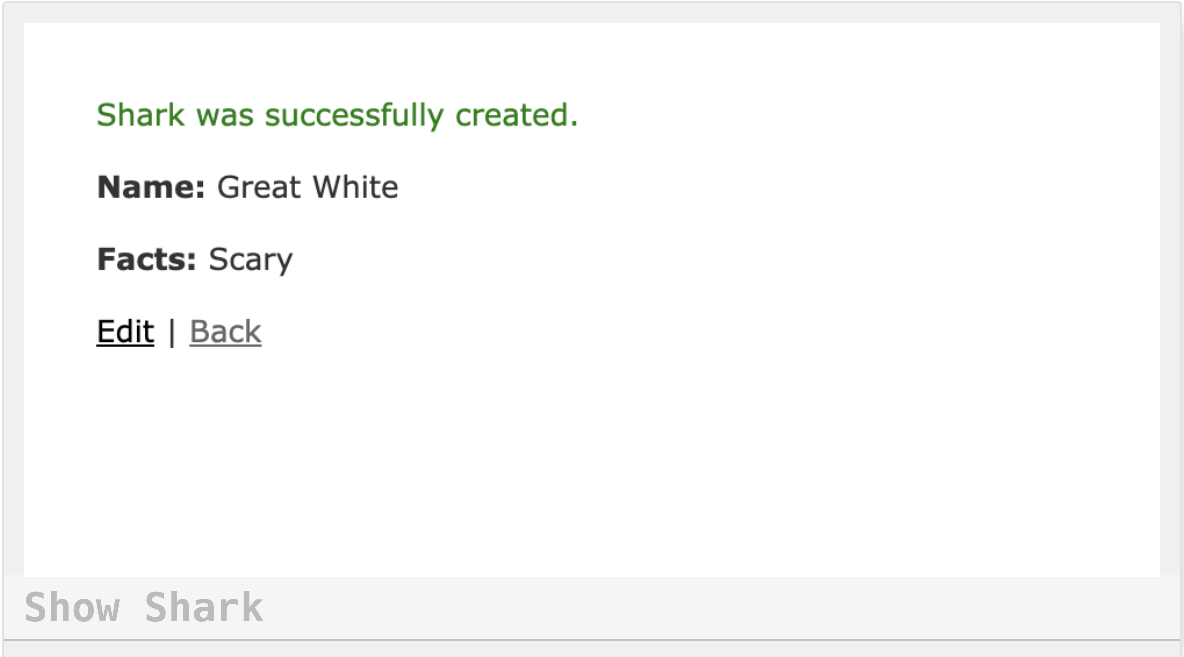


Create New Shark

To verify that the application is working, we can add some demo information to it. Input “Great White” into the **Name** field and “Scary” into the **Facts** field:



Click on the **Create Shark** button to create the shark:



You have now installed the necessary dependencies for your project and tested its functionality. Next, you can make a few changes to the Rails application so that users encounter a main landing page before navigating to the shark information application itself.

Step 2 — Adding a Main Landing Page and Controller

The current application sets the root view to the main shark information page, the `index` view for the `sharks` controller. While this works to get users to the main application, it may be less desirable if we decide to develop the application in the future and add other capabilities and features. We can reorganize the application to have the root view set to a `home` controller, which will include an `index` view. From there, we can link out to other parts of the application.

To create the `home` controller, you can use the [`rails generate`](https://guides.rubyonrails.org/command_line.html#rails-generate) command with the `controller` generator. In this case, we will specify that we want an `index` view for our main landing page:

```
rails generate controller home index
```

With the controller created, you'll need to modify the root view in the project's `config/routes.rb` file — the file that specifies the application's route declarations — since the root view is currently set to the sharks `index` view.

Open the file:

```
nano config/routes.rb
```

Find the following line:

```
~/rails-bootstrap/config/routes.rb
. . .
root 'sharks#index'
. . .
```

Change it to the following:

```
~/rails-bootstrap/config/routes.rb
. . .
root 'home#index'
. . .
```

This will set the `home` controller's `index` view as the root of the application, making it possible to branch off to other parts of the application from there.

Save and close the file when you are finished editing.

With these changes in place, you are ready to move on to adding Bootstrap to the application.

Step 3 — Installing Bootstrap and Adding Custom Styles

In this step, you will add Bootstrap to your project, along with the tool libraries that it requires to function properly. This will involve importing libraries and plugins into the application's webpack entry point and environment files. It will also involve creating a custom style sheet in your application's `app/javascript` directory, the directory where the project's JavaScript assets live.

First, use `yarn` to install Bootstrap and its required dependencies:

```
yarn add bootstrap jquery popper.js
```

Many of Bootstrap's components require JQuery and Popper.js, along with Bootstrap's own custom plugins, so this command will ensure that you have the libraries you need.

Next, open your main webpack configuration file, `config/webpack/environment.js` with `nano` or your favorite editor:

```
nano config/webpack/environment.js
```

Inside the file, add the webpack library, along with a [`ProvidePlugin`](https://webpack.js.org/plugins/provide-plugin/) that tells Bootstrap how to interpret JQuery and Popper variables.

Add the following code to the file:

```
~/rails-bootstrap/config/webpack/environment.js
const { environment } = require('@rails/webpacker')
const webpack = require("webpack")

environment.plugins.append("Provide", new webpack.ProvidePlugin({
  $: 'jquery',
  jQuery: 'jquery',
  Popper: ['popper.js', 'default']
}))

module.exports = environment
```

The `ProvidePlugin` helps us avoid the multiple `import` or `require` statements we would normally use when working with JQuery or Popper modules. With this plugin in place, webpack will automatically load the correct modules and point the named variables to each module's loaded exports.

Save and close the file when you are finished editing.

Next, open your main webpack entry point file, `app/javascript/packs/application.js`:

```
nano app/javascript/packs/application.js
```

Inside the file, add the following `import` statements to import Bootstrap and the custom `scss` styles file that you will create next:

```
. . .
[label ~/rails-bootstrap/app/javascript/packs/application.js]
import { Application } from "stimulus"
import { definitionsFromContext } from "stimulus/webpack-helpers"

import "bootstrap"
import "../stylesheets/application"
. . .
```

Save and close the file when you are finished editing.

Next, create a `stylesheets` directory for your application style sheet:

```
mkdir app/javascript/stylesheets
```

Open the custom styles file:

```
nano app/javascript/stylesheets/application.scss
```

This is an `scss` file, which uses [Sass](#) instead of [CSS](#). Sass, or Syntactically Awesome Style Sheets, is a CSS extension language that lets developers integrate programming logic and conventions like shared variables into styling rules.

In the file, add the following statements to import the custom Bootstrap `scss` styles and Google fonts for the project:

```
~/rails-bootstrap/app/javascript/stylesheets/application.scss
@import "~bootstrap/scss/bootstrap";
@import url('https://fonts.googleapis.com/css?family=Merriweather:400,700');
```

Next, add the following custom variable definitions and styles for the application:

~/rails-bootstrap/app/javascript/stylesheets/application.scss

```
. . .
$white: white;
$black: black;

.navbar {
        margin-bottom: 0;
        background: $black;
}
body {
        background: $black;
        color: $white;
        font-family: 'Merriweather', sans-serif;
}
h1,
h2 {
        font-weight: bold;
}
p {
        font-size: 16px;
        color: $white;
}
a:visited {
        color: $black;
```

```
}
.jumbotron {
        background: #0048CD;
        color: $white;
        text-align: center;
        p {
                color: $white;
                font-size: 26px;
        }
}
.link {
        color: $white;
}
.btn-primary {
        color: $white;
        border-color: $white;
        margin-bottom: 5px;
}
.btn-sm {
        background-color: $white;
        display: inline-block;
}
img,
video,
audio {
        margin-top: 20px;
        max-width: 80%;
```

```
}
caption {


        float: left;
        clear: both;


}
```

Save and close the file when you are finished editing.

You have added Bootstrap to your project, along with some custom styles. Now you can move on to integrating Bootstrap layout conventions and components into your application files.

Step 4 — Modifying the Application Layout

Our first step in integrating Bootstrap conventions and components into the project will be adding them to the main application layout file. This file sets the elements that will be included with each rendered view template for the application. In this file, we'll make sure our webpack entry point is defined, while also adding references to a shared navigation headers partial and some logic that will allow us to render a layout for the views associated with the shark application.

First, open `app/views/layouts/application.html.erb`, your application's main layout file:

```
nano app/views/layouts/application.html.erb
```

Currently, the file looks like this:

```
~/rails-bootstrap/app/views/layouts/application.html.erb
<!DOCTYPE html>
<html>
  <head>
    <title>Sharkapp</title>
    <%= csrf_meta_tags %>
    <%= csp_meta_tag %>

    <%= stylesheet_link_tag 'application', media: 'all', 'data-turbolinks-track': 'reload' %>
    <%= javascript_pack_tag 'application', 'data-turbolinks-track': 'reload' %>
  </head>

  <body>
    <%= yield %>
  </body>
</html>
```

The code renders things like [cross-site request forgery protection parameters and tokens] for dynamic forms, a [csp-nonce] for per-session nonces that allows in-line script tags, and the application's style sheets and javascript assets. Note that rather than having a `javascript_link_tag`, our code includes a `javascript_pack_tag`, which tells Rails to load our main webpack entry point at `app/javascript/packs/application.js`.

In the `<body>` of the page, a `yield` statement tells Rails to insert the content from a view. In this case, because our application root formerly mapped to the `index` shark view, this would have inserted the content from that view. Now, however, because we have changed the root view, this will insert content from the `home` controller's `index` view.

This raises a couple of questions: Do we want the home view for the application to be the same as what users see when they view the shark application? And if we want these views to be somewhat different, how do we implement that?

The first step will be deciding what should be replicated across all application views. We can leave everything included under the `<header>` in place, since it is primarily tags and metadata that we want to be present on all application pages. Within this section, however, we can also add a few things that will customize all of our application views.

First, add the `viewport` meta tag that Bootstrap recommends for responsive behaviors:

~/rails-bootstrap/app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width, initial
-scale=1.0">
    <title>Sharkapp</title>
    <%= csrf_meta_tags %>
    <%= csp_meta_tag %>
. . .
```

Next, replace the existing `title` code with code that will render the application title in a more dynamic way:

```
~/rails-bootstrap/app/views/layouts/application.html.erb
```

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width, initial
-scale=1.0">
    <title><%= content_for?(:title) ? yield(:title) : "About S
harks" %></title>
    <%= csrf_meta_tags %>
    <%= csp_meta_tag %>
. . .
```

Add a `<meta>` tag to include a description of the site:

```
~/rails-bootstrap/app/views/layouts/application.html.erb
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title><%= content_for?(:title) ? yield(:title) : "About Sharks" %></title>
    <meta name="description" content="<%= content_for?(:description) ? yield(:description) : "About Sharks" %>">
    <%= csrf_meta_tags %>
    <%= csp_meta_tag %>
. . .
```

With this code in place, you can add a navigation partial to the layout. Ideally, each of our application's pages should include a navbar component at the top of the page, so that users can easily navigate from one part of the site to another.

Under the `<body>` tag, add a `<header>` tag and the following render statement:

```
~/rails-bootstrap/app/views/layouts/application.html.erb
  <body>
    <header>
      <%= render 'layouts/navigation' %>
    </header>

    <%= yield %>
. . .
```

This `<header>` tag allows you to organize your page content, separating the navbar from the main page contents.

Finally, you can add a `<main>` element tag and some logic to control which view, and thus which layout, the application will render. This code uses the [`content_for` method] to reference a content identifier that we will associate with our sharks layout in the next step.

Replace the existing `yield` statement with the following content:

```
~/rails-bootstrap/app/views/layouts/application.html.erb
. . .
  <body>
    <header>
      <%= render 'layouts/navigation' %>
    </header>
    <main role="main">
    <%= content_for?(:content) ? yield(:content) : yield %>
    </main>
  </body>
</html>
```

Now, if the `:content` block is set, the application will yield the associated layout. Otherwise, thanks to the ternary operator, it will do an implicit yield of the view associated with the `home` controller.

Once you have made these changes, save and close the file.

With the application-wide layout set, you can move on to creating the shared navbar partial and the sharks layout for your shark views.

Step 5 — Creating the Shared Partial and Specific Layouts

In addition to the changes you made to the application layout in the previous Step, you will want to create the shared navbar partial, the sharks layout that you referenced in `app/views/layouts/application.html.erb`, and a view for the application landing page. You can also add Bootstrap styles to your application's current `link_to` elements in order to take advantage of built-in Bootstrap styles.

First, open a file for the shared navbar partial:

```
nano app/views/layouts/_navigation.html.erb
```

Add the following code to the file to create the navbar:

~/rails-bootstrap/app/views/layouts/_navigation.html.erb

```
<nav class="navbar navbar-dark navbar-static-top navbar-expand
-md">
    <div class="container">
        <button type="button" class="navbar-toggler collapsed"
data-toggle="collapse" data-target="#bs-example-navbar-collaps
e-1" aria-expanded="false"> <span class="sr-only">Toggle navig
ation</span>
        </button> <%= link_to "Everything Sharks", root_path,
 class: 'navbar-brand' %>
        <div class="collapse navbar-collapse" id="bs-example-n
avbar-collapse-1">
            <ul class="nav navbar-nav mr-auto">
            <li class='nav-item'><%= link_to 'Home', home_inde
x_path, class: 'nav-link' %></li>
            <li class='nav-item'><%= link_to 'Sharks', sharks_
path, class: 'nav-link' %></li>


                </li>
            </ul>
        </div>
    </div>
</nav>
```

This navbar creates a link for the application root using the `link_to` method, which maps to the application root path. The navbar also includes two additional links: one to the `Home` path, which maps to the `home` controller's `index` view, and another to the shark application path, which maps to the `shark` `index` view.

Save and close the file when you are finished editing.

Next, open a file in the `layouts` directory for the sharks layout:

```
nano app/views/layouts/sharks.html.erb
```

Before adding layout features, we will need to ensure that the content of the layout is set as the `:content` block that we reference in the main application layout. Add the following lines to the file to create the block:

```
~/rails-bootstrap/app/views/layouts/sharks.html.erb
<% content_for :content do %>
<% end %>
```

The code we're about to write in this block will be rendered inside the `:content` block in the `app/views/layouts/application.html.erb` file whenever a sharks view is requested by a controller.

Next, inside the block itself, add the following code to create a jumbotron component and two containers:

~/rails-bootstrap/app/views/layouts/sharks.html.erb

```
<% content_for :content do %>
    <div class="jumbotron text-center">
        <h1>Shark Info</h1>
    </div>
    <div class="container">
        <div class="row">
            <div class="col-lg-6">
                <p>
                    <%= yield %>
                </p>
            </div>
            <div class="col-lg-6">
                <p>

                    <div class="caption">You can always count
 on some sharks to be friendly and welcoming!</div>
                    <img src="https://assets.digitalocean.com/
articles/docker_node_image/sammy.png" alt="Sammy the Shark">
                </p>

            </div>
        </div>
    </div>
    <% end %>
```

The first container includes a `yield` statement that will insert the content from the `shark` controller's views, while the second includes a reminder that certain sharks are always friendly and welcoming.

Finally, at the bottom of the file, add the following `render` statement to render the application layout:

```
~/rails-bootstrap/app/views/layouts/sharks.html.erb
```

```
. . .
            </div>
        </div>
    </div>
    <% end %>
        <%= render template: "layouts/application" %>
```

This sharks layout will provide the content for the named `:content` block in the main application layout; it will then render the application layout itself to ensure that our rendered application pages have everything we want at the application-wide level.

Save and close the file when you are finished editing.

We now have our partials and layouts in place, but we haven't yet created the view that users will see when they navigate to the application home page, the `home` controller's `index` view.

Open that file now:

```
nano app/views/home/index.html.erb
```

The structure of this view will match the layout we defined for shark views, with a main jumbotron component and two containers. Replace the boilerplate code in the file with the following:

~/rails-bootstrap/app/views/home/index.html.erb

```
<div class="jumbotron">
    <div class="container">
        <h1>Want to Learn About Sharks?</h1>
        <p>Are you ready to learn about sharks?</p>
        <br>
        <p>
            <%= button_to 'Get Shark Info', sharks_path, :meth
od => :get,  :class => "btn btn-primary btn-lg"%>
        </p>
    </div>
</div>
<div class="container">
    <div class="row">
        <div class="col-lg-6">
            <h3>Not all sharks are alike</h3>
            <p>Though some are dangerous, sharks generally do
 not attack humans. Out of the 500 species known to researcher
s, only 30 have been known to attack humans.
            </p>
        </div>
        <div class="col-lg-6">
            <h3>Sharks are ancient</h3>
            <p>There is evidence to suggest that sharks lived
 up to 400 million years ago.
            </p>
```

```
        </div>
    </div>
</div>
```

Now, when landing on the home page of the application, users will have a clear way to navigate to the shark section of the application, by clicking on the **Get Shark Info** button. This button points to the `shark_path` — the helper that maps to the routes associated with the `sharks` controller.

Save and close the file when you are finished editing.

Our last task will be to transform some of the `link_to` methods in our application into buttons that we can style using Bootstrap. We will also add a way to navigate back to the home page from the sharks `index` view.

Open the sharks `index` view to start:

```
nano app/views/sharks/index.html.erb
```

At the bottom of the file, locate the `link_to` method that directs to the `new` shark view:

~/rails-bootstrap/app/views/sharks/index.html.erb

```
. . .
<%= link_to 'New Shark', new_shark_path %>
```

Modify the code to turn this link into a button that uses Bootstrap's `"btn btn-primary btn-sm"` class:

```
~/rails-bootstrap/app/views/sharks/index.html.erb
```

```
. . .
<%= link_to 'New Shark', new_shark_path, :class => "btn btn-primary btn-sm" %>
```

Next, add a link to the application home page:

```
~/rails-bootstrap/app/views/sharks/index.html.erb
```

```
. . .
<%= link_to 'New Shark', new_shark_path, :class => "btn btn-primary btn-sm" %> <%= link_to 'Home', home_index_path, :class => "btn btn-primary btn-sm" %>
```

Save and close the file when you are finished editing.

Next, open the `new` view:

```
nano app/views/sharks/new.html.erb
```

Add the button styles to the `link_to` method at the bottom of the file:

```
~/rails-bootstrap/app/views/sharks/new.html.erb
. . .
<%= link_to 'Back', sharks_path, :class => "btn btn-primary bt
n-sm" %>
```

Save and close the file.

Open the `edit` view:

```
nano app/views/sharks/edit.html.erb
```

Currently, the `link_to` methods are arranged like this:

```
~/rails-bootstrap/app/views/sharks/edit.html.erb
. . .
<%= link_to 'Show', @shark %> |
<%= link_to 'Back', sharks_path %>
```

Change their arrangement on the page and add the button styles, so that the code looks like this:

```
~/rails-bootstrap/app/views/sharks/edit.html.erb
. . .
<%= link_to 'Show', @shark, :class => "btn btn-primary btn-sm"
%> <%= link_to 'Back', sharks_path, :class => "btn btn-primary
btn-sm" %>
```

Save and close the file.

Finally, open the `show` view:

```
nano app/views/sharks/show.html.erb
```

Find the following `link_to` methods:

```
~/rails-bootstrap/app/views/sharks/show.html.erb
. . .
<%= link_to 'Edit', edit_shark_path(@shark) %> |
<%= link_to 'Back', sharks_path %>
. . .
```

Change them to look like this:

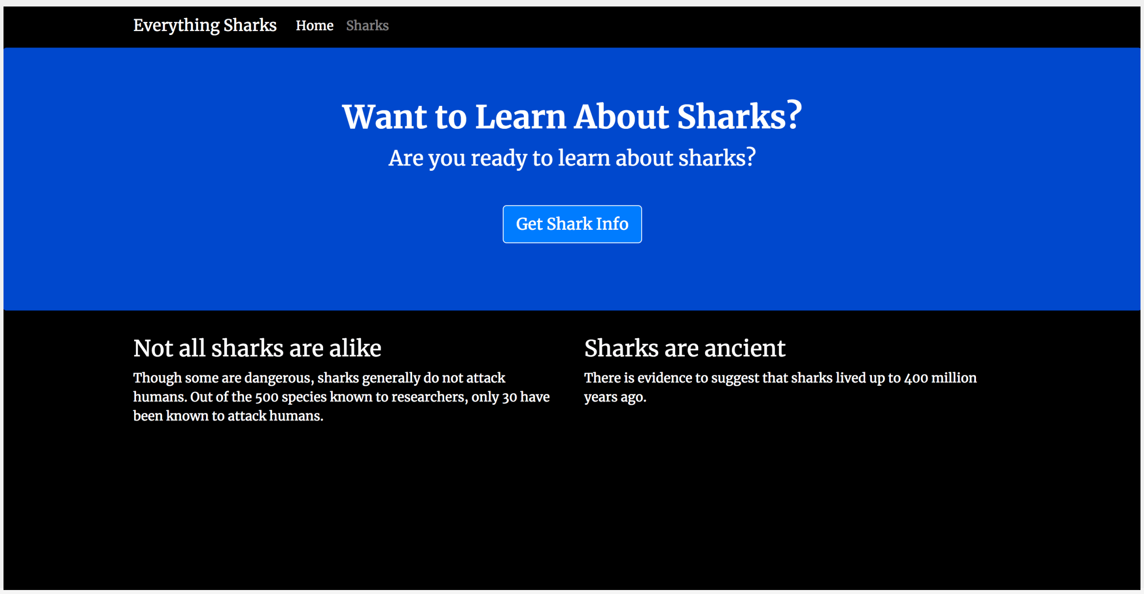
```
~/rails-bootstrap/app/views/sharks/show.html.erb
. . .
<%= link_to 'Edit', edit_shark_path(@shark), :class => "btn bt
n-primary btn-sm" %> <%= link_to 'Back', sharks_path, :class =
> "btn btn-primary btn-sm" %>
. . .
```

Save and close the file.

You are now ready to test the application.

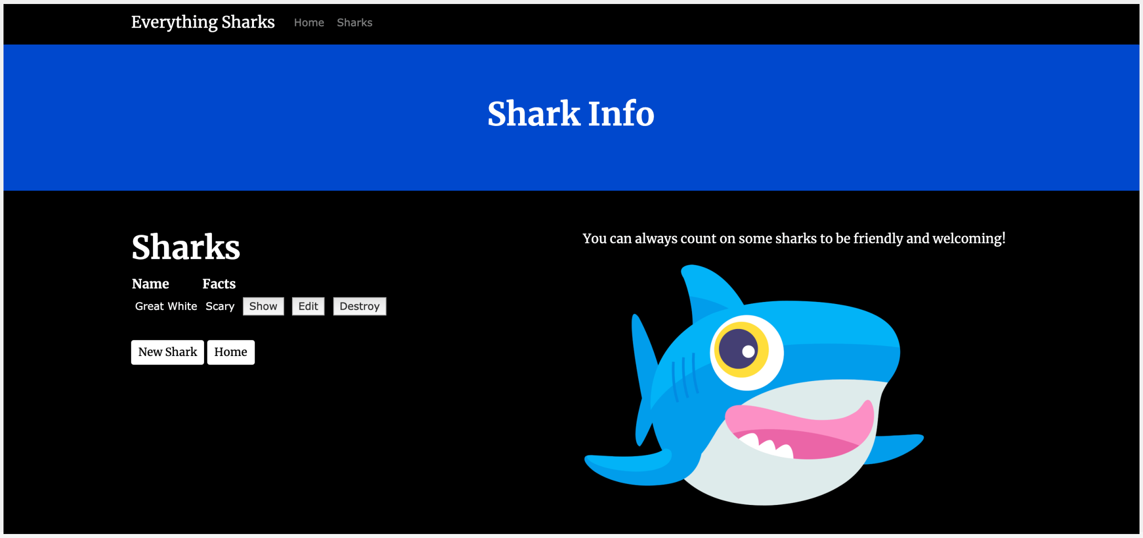
Start your server with the appropriate command: - `rails s` if you are working locally - `rails s --binding=your_server_ip` if you are working with a development server

Navigate to `localhost:3000` or `http://your_server_ip:3000`, depending on whether you are working locally or on a server. You will see the following landing page:



Application Landing Page

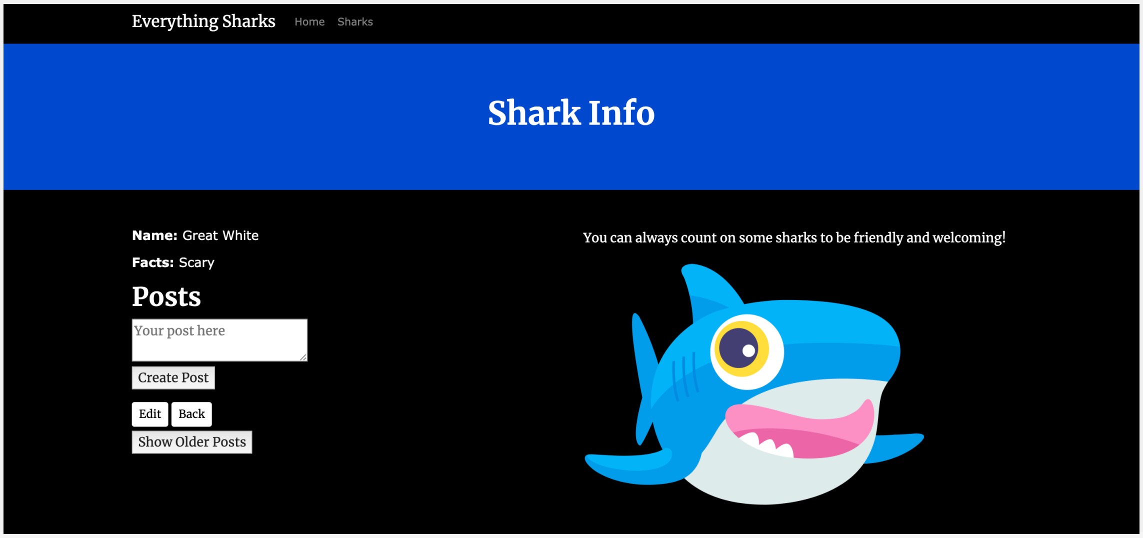
Click on **Get Shark Info**. You will see the following page:



Sharks Index Page

You can now edit your shark, or add facts and posts, using the methods described in [How To Add Stimulus to a Ruby on Rails Application](). You can also add new sharks to the conversation.

As you navigate to other shark views, you will see that the shark layout is always included:



Sharks Show Page

You now have Bootstrap integrated into your Rails application. From here, you can move forward by adding new styles and components to your application to make it more appealing to users.

Conclusion

You now have Bootstrap integrated into your Rails application, which will allow you to create responsive and visually appealing styles to enhance your users' experience of the project.

To learn more about Bootstrap features and what they offer, please see the Bootstrap documentation. You can also look at the documentation for Sass, to get a sense of how you can use it to enhance and extend your CSS styles and logic.

If you are interested in seeing how Bootstrap integrates with other frameworks, please see How To Build a Weather App with Angular, Bootstrap, and the APIXU API. You can also learn about how it integrates with Rails and React by reading How To Set Up a Ruby on Rails Project with a React Frontend.

How To Add Sidekiq and Redis to a Ruby on Rails Application

Written by Kathleen Juell

When developing a Ruby on Rails application, you may find you have application tasks that should be performed asynchronously. Processing data, sending batch emails, or interacting with external APIs are all examples of work that can be done asynchronously with background jobs. Using background jobs can improve your application's performance by offloading potentially time-intensive tasks to a background processing queue, freeing up the original request/response cycle.

Sidekiq is one of the more widely used background job frameworks that you can implement in a Rails application. It is backed by Redis, an in-memory key-value store known for its flexibility and performance. Sidekiq uses Redis as a job management store to process thousands of jobs per second.

In this tutorial, you will add Redis and Sidekiq to an existing Rails application. You will create a set of Sidekiq worker classes and methods to handle: - A batch upload of endangered shark information to the application database from a CSV file in the project repository. - The removal of this data.

When you are finished, you will have a demo application that uses workers and jobs to process tasks asynchronously. This will be a good foundation for you to add workers and jobs to your own application, using this tutorial as a jumping off point.

Prerequisites

To follow this tutorial, you will need: - A local machine or development server running Ubuntu 18.04. Your development machine should have a non-root user with administrative privileges and a firewall configured with `ufw`. For instructions on how to set this up, see our [Initial Server Setup with Ubuntu 18.04]() tutorial. - [Node.js]() and [npm]() installed on your local machine or development server. This tutorial uses Node.js version <>10.17.0<> and npm version <>6.11.3<>. For guidance on installing Node.js and npm on Ubuntu 18.04, follow the instructions in the **“Installing Using a PPA”** section of [How To Install Node.js on Ubuntu 18.04](). - The [Yarn package manager]() installed on your local machine or development server. You can following the installation [instructions]() in the official documentation. - Ruby, [rbenv](), and Rails installed on your local machine or development server, following **Steps 1-4** in [How To Install Ruby on Rails with rbenv on Ubuntu 18.04](). This tutorial uses Ruby <>2.5.1<>, rbenv <>1.1.2<>, and Rails <>5.2.3<>. - SQLite installed, following **Step 1** of [How To Build a Ruby on Rails Application](). This tutorial uses SQLite 3 <>3.22.0<>. - Redis installed, following **Steps 1-3** of [How To Install and Secure Redis on Ubuntu 18.04](). This tutorial uses Redis <>4.0.9<>.

Step 1 — Cloning the Project and Installing Dependencies

Our first step will be to clone the rails-bootstrap repository from the DigitalOcean Community GitHub account. This repository includes the code from the setup described in How To Add Bootstrap to a Ruby on Rails Application, which explains how to add Bootstrap to an existing Rails 5 project.

Clone the repository into a directory called `rails-sidekiq`:

```
git clone https://github.com/do-community/rails-bootstrap.git
rails-sidekiq
```

Navigate to the `rails-sidekiq` directory:

```
cd rails-sidekiq
```

In order to work with the code, you will first need to install the project's dependencies, which are listed in its Gemfile. You will also need to add the sidekiq gem to the project to work with Sidekiq and Redis.

Open the project's Gemfile for editing, using `nano` or your favorite editor:

```
nano Gemfile
```

Add the gem anywhere in the main project dependencies (above development dependencies):

~/rails-sidekiq/Gemfile

```
. . .
# Reduces boot times through caching; required in config/boot.
rb
gem 'bootsnap', '>= 1.1.0', require: false
gem 'sidekiq', '~>6.0.0'

group :development, :test do
. . .
```

Save and close the file when you are finished adding the gem.

Use the following command to install the gems:

```
bundle install
```

You will see in the output that the `redis` gem is also installed as a requirement for `sidekiq`.

Next, you will install your Yarn dependencies. Because this Rails 5 project has been modified to serve assets with webpack, its JavaScript dependencies are now managed by Yarn. This means that it's necessary to install and verify the dependencies listed in the project's `package.json` file.

Run `yarn install` to install these dependencies:

```
yarn install
```

Next, run your database migrations:

```
rails db:migrate
```

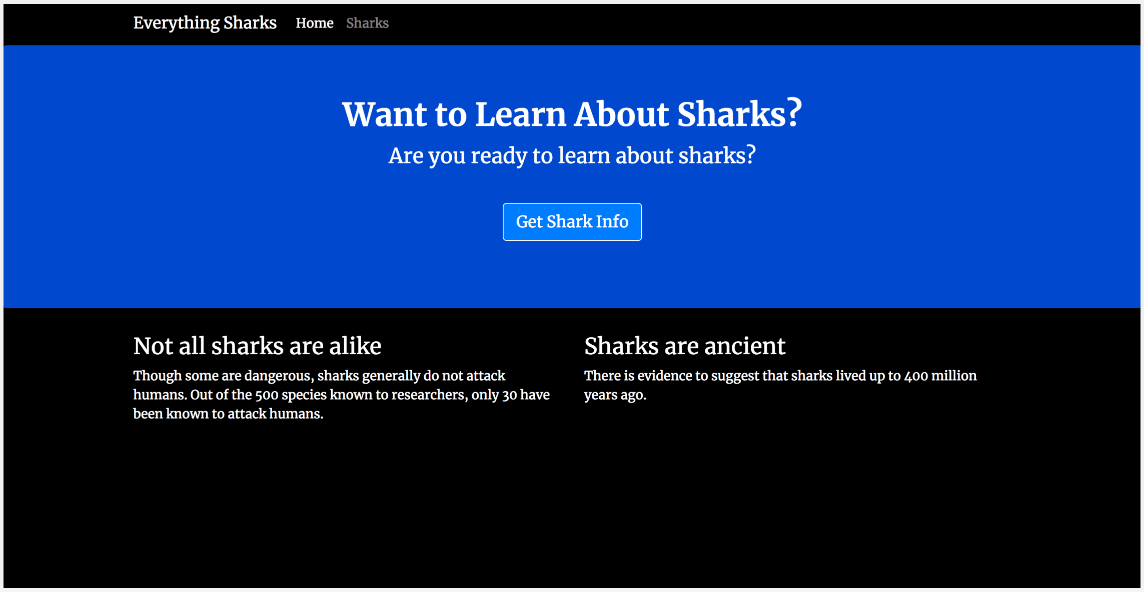
Once your migrations have finished, you can test the application to ensure that it is working as expected. Start your server in the context of your local bundle with the following command if you are working locally:

```
bundle exec rails s
```

If you are working on a development server, you can start the application with:

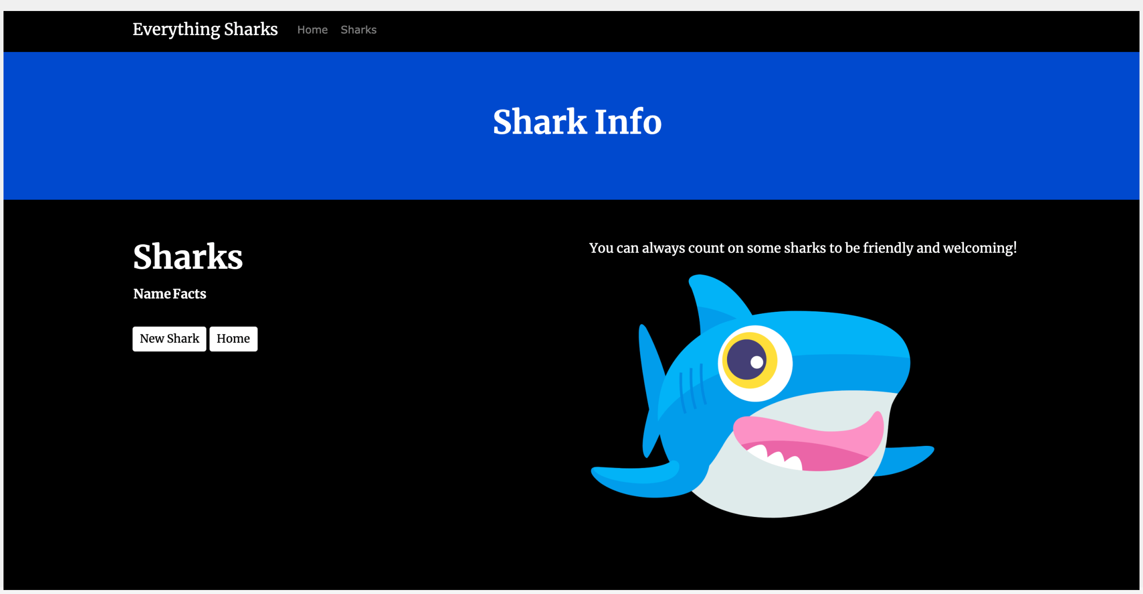
```
bundle exec rails s --binding=your_server_ip
```

Navigate to `localhost:3000` or `http://your_server_ip:3000`. You will see the following landing page:



Application Landing Page

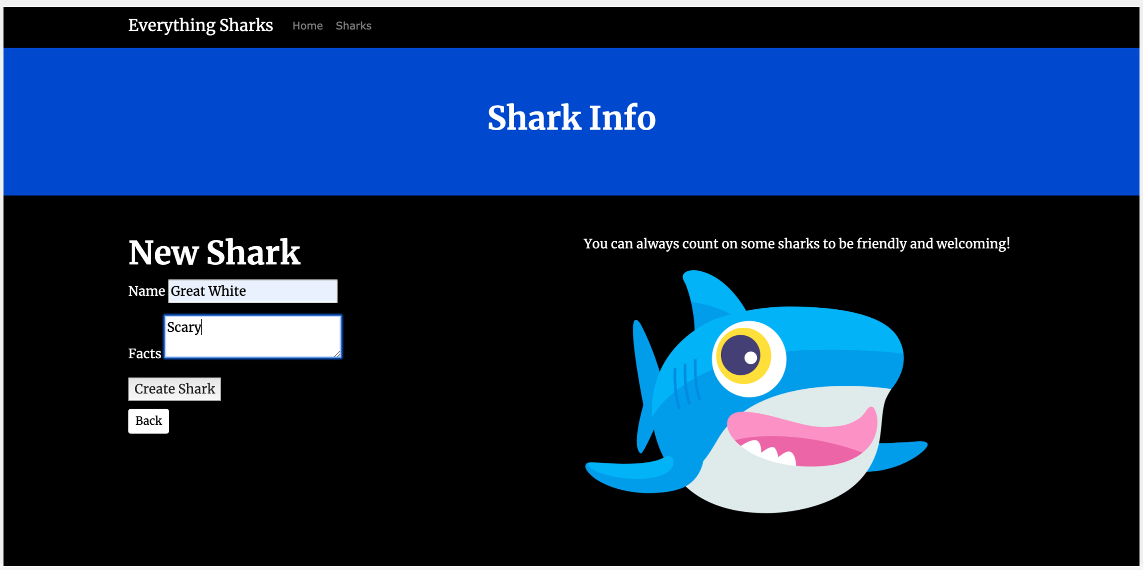
To create a new shark, click on the **Get Shark Info** button, which will take you to the `sharks/index` route:



Sharks Index Route

To verify that the application is working, we can add some demo information to it. Click on **New Shark**. You will be prompted for a username (**sammy**) and password (**shark**), thanks to the project's authentication settings.

On the **New Shark** page, input "Great White" into the **Name** field and "Scary" into the **Facts** field:



Shark Create

Click on the **Create Shark** button to create the shark. Once you see that your shark has been created, you can kill the server with `CTRL+C`.

You have now installed the necessary dependencies for your project and tested its functionality. Next, you can make a few changes to the Rails application to work with your endangered sharks resources.

Step 2 — Generating a Controller for Endangered Shark Resources

To work with our endangered shark resources, we will add a new model to the application and a controller that will control how information about endangered sharks is presented to users. Our ultimate goal is to make it possible for users to upload a large batch of information about endangered sharks without blocking our application's overall functionality, and to delete that information when they no longer need it.

First, let's create an `Endangered` model for our endangered sharks. We'll include a string field in our database table for the shark name, and another string field for the [International Union for the Conservation of Nature (IUCN) categories](#) that determine the degree to which each shark is at risk.

Ultimately, our model structure will match the columns in the CSV file that we will use to create our batch upload. This file is located in the `db` directory, and you can check its contents with the following command:

```
cat db/sharks.csv
```

The file contains a list of 73 endangered sharks and their IUCN statuses - **vu** for vulnerable, **en** for endangered, and **cr** for critically endangered.

Our `Endangered` model will correlate with this data, allowing us to create new `Endangered` instances from this CSV file. Create the model with the following command:

```
rails generate model Endangered name:string iucn:string
```

Next, generate an `Endangered` controller with an `index` action:

```
rails generate controller endangered index
```

This will give us a starting point to build out our application's functionality, though we will also need to add custom methods to the controller file that Rails has generated for us.

Open that file now:

```
nano app/controllers/endangered_controller.rb
```

Rails has provided us with a skeletal outline that we can begin to fill in.

First, we'll need to determine what routes we require to work with our data. Thanks to the `generate controller` command, we have an `index` method to begin with. This will correlate to an `index` view, where we will present users with the option to upload endangered sharks.

However, we will also want to deal with cases where users may have already uploaded the sharks; they will not need an upload option in this case. We will somehow need to assess how many instances of the `Endangered` class already exist, since more than one indicates that the batch upload has already occurred.

Let's start by creating a `set_endangered` `private` method that will grab each instance of our `Endangered` class from the database. Add the following code to the file:

~/rails-sidekiq/app/controllers/endangered_controller.rb

```
class EndangeredController < ApplicationController
  before_action :set_endangered, only: [:index, :data]

  def index
  end

  private

    def set_endangered
      @endangered = Endangered.all
    end

end
```

Note that the `before_action` filter will ensure that the value of `@endangered` is only set for the `index` and `data` routes, which will be where we handle the endangered shark data.

Next, add the following code to the `index` method to determine the correct path for users visiting this part of the application:

```
~/rails-sidekiq/app/controllers/endangered_controller.rb
```

```
class EndangeredController < ApplicationController
  before_action :set_endangered, only: [:index, :data]

  def index
    if @endangered.length > 0
      redirect_to endangered_data_path
    else
      render 'index'
    end
  end
. . .
```

If there are more than 0 instances of our `Endangered` class, we will redirect users to the `data` route, where they can view information about the sharks they've created. Otherwise, they will see the `index` view.

Next, below the `index` method, add a `data` method, which will correlate to a `data` view:

```
~/rails-sidekiq/app/controllers/endangered_controller.rb
. . .
  def index
    if @endangered.length > 0
      redirect_to endangered_data_path
    else
      render 'index'
    end
  end

  def data
  end
. . .
```

Next, we will add a method to handle the data upload itself. We'll call this method `upload`, and it will call a Sidekiq worker class and method to perform the data upload from the CSV file. We will create the definition for this worker class, `AddEndangeredWorker`, in the next step.

For now, add the following code to the file to call the Sidekiq worker to perform the upload:

```
~/rails-sidekiq/app/controllers/endangered_controller.rb
. . .
  def data
  end

  def upload
    csv_file = File.join Rails.root, 'db', 'sharks.csv'
    AddEndangeredWorker.perform_async(csv_file)
    redirect_to endangered_data_path, notice: 'Endangered shar
ks have been uploaded!'
  end
. . .
```

By calling the `perform_async` method on the `AddEndangeredWorker` class, using the CSV file as an argument, this code ensures that the shark data and upload job get passed to Redis. The Sidekiq workers that we will set up monitor the job queue and will respond when new jobs arise.

After calling `perform_async`, our `upload` method redirects to the `data` path, where users will be able to see the uploaded sharks.

Next, we'll add a `destroy` method to destroy the data. Add the following code below the `upload` method:

```
~/rails-sidekiq/app/controllers/endangered_controller.rb
. . .
  def upload
    csv_file = File.join Rails.root, 'db', 'sharks.csv'
    AddEndangeredWorker.perform_async(csv_file)
    redirect_to endangered_data_path, notice: 'Endangered sharks have been uploaded!'
  end

  def destroy
    RemoveEndangeredWorker.perform_async
    redirect_to root_path
  end
. . .
```

Like our `upload` method, our `destroy` method includes a `perform_async` call on a `RemoveEndangeredWorker` class – the other Sidekiq worker that we will create. After calling this method, it redirects users to the root application path.

The finished file will look like this:

~/rails-sidekiq/app/controllers/endangered_controller.rb

```
class EndangeredController < ApplicationController
  before_action :set_endangered, only: [:index, :data]

  def index
    if @endangered.length > 0
      redirect_to endangered_data_path
    else
      render 'index'
    end
  end

  def data
  end

  def upload
    csv_file = File.join Rails.root, 'db', 'sharks.csv'
    AddEndangeredWorker.perform_async(csv_file)
    redirect_to endangered_data_path, notice: 'Endangered shar
ks have been uploaded!'
  end

  def destroy
    RemoveEndangeredWorker.perform_async
    redirect_to root_path

```

```
  end

  private

    def set_endangered
      @endangered = Endangered.all
    end

end
```

Save and close the file when you are finished editing.

As a final step in solidifying our application's routes, we will modify the code in `config/routes.rb`, the file where our route declarations live.

Open that file now:

```
nano config/routes.rb
```

The file currently looks like this:

~/rails-sidekiq/config/routes.rb

```
Rails.application.routes.draw do
  get 'endangered/index'
  get 'home/index'
  resources :sharks do
          resources :posts
  end
  root 'home#index'
  # For details on the DSL available within this file, see htt
p://guides.rubyonrails.org/routing.html
end
```

We will need to update the file to include the routes that we've defined in our controller: `data`, `upload`, and `destroy`. Our `data` route will match with a GET request to retrieve the shark data, while our `upload` and `destroy` routes will map to POST requests that upload and destroy that data.

Add the following code to the file to define these routes:

```
~/rails-sidekiq/config/routes.rb
Rails.application.routes.draw do
  get 'endangered/index'
  get 'endangered/data', to: 'endangered#data'
  post 'endangered/upload', to: 'endangered#upload'
  post 'endangered/destroy', to: 'endangered#destroy'
  get 'home/index'
  resources :sharks do
          resources :posts
  end
  root 'home#index'
  # For details on the DSL available within this file, see htt
p://guides.rubyonrails.org/routing.html
end
```

Save and close the file when you are finished editing.

With your `Endangered` model and controller in place, you can now move on to defining your Sidekiq worker classes.

Step 3 — Defining Sidekiq Workers

We have called `perform_async` methods on our Sidekiq workers in our controller, but we still need to create the workers themselves.

First, create a `workers` directory for the workers:

```
mkdir app/workers
```

Open a file for the `AddEndangeredWorker` worker:

```
nano app/workers/add_endangered_worker.rb
```

In this file, we will add code that will allow us to work with the data in our CSV file. First, add code to the file that will create the class, include the Ruby CSV library, and ensure that this class functions as a Sidekiq Worker:

~/rails-sidekiq/app/workers/add_endangered_worker.rb

```
class AddEndangeredWorker
  require 'csv'
  include Sidekiq::Worker
  sidekiq_options retry: false

end
```

We're also including the `retry: false` option to ensure that Sidekiq does not retry the upload in the case of failure.

Next, add the code for the `perform` function:

```
~/rails-sidekiq/app/workers/add_endangered_worker.rb
```

```ruby
class AddEndangeredWorker
  require 'csv'
  include Sidekiq::Worker
  sidekiq_options retry: false

  def perform(csv_file)
    CSV.foreach(csv_file, headers: true) do |shark|
    Endangered.create(name: shark[0], iucn: shark[1])
  end
 end

end
```

The `perform` method receives arguments from the `perform_async` method defined in the controller, so it's important that the argument values are aligned. Here, we pass in `csv_file`, the variable we defined in the controller, and we use the `foreach` method from the CSV library to read the values in the file. Setting `headers: true` for this loop ensures that the first row of the file is treated as a row of headers.

The block then reads the values from the file into the columns we set for our `Endangered` model: `name` and `iucn`. Running this loop will create `Endangered` instances for each of the entries in our CSV file.

Once you have finished editing, save and close the file.

Next, we will create a worker to handle deleting this data. Open a file for the `RemoveEndangeredWorker` class:

```
nano app/workers/remove_endangered_worker.rb
```

Add the code to define the class, and to ensure that it uses the CSV library and functions as a Sidekiq Worker:

```
~/rails-sidekiq/app/workers/remove_endangered_worker.rb
class RemoveEndangeredWorker
  include Sidekiq::Worker
  sidekiq_options retry: false


end
```

Next, add a `perform` method to handle the destruction of the endangered shark data:

```
~/rails-sidekiq/app/workers/remove_endangered_worker.rb
```

```ruby
class RemoveEndangeredWorker
  include Sidekiq::Worker
  sidekiq_options retry: false

  def perform
    Endangered.destroy_all
  end

end
```

The `perform` method calls `destroy_all` on the `Endangered` class, which will remove all instances of the class from the database.

Save and close the file when you are finished editing.

With your workers in place, you can move on to creating a layout for your `endangered` views, and templates for your `index` and `data` views, so that users can upload and view endangered sharks.

Step 4 — Adding Layouts and View Templates

In order for users to enjoy their endangered shark information, we will need to address two things: the layout for the views defined in our `endangered` controller, and the view templates for the `index` and `data` views.

Currently, our application makes use of an application-wide layout, located at `app/views/layouts/application.html.erb`, a navigation partial, and a layout for `sharks` views. The application layout checks for a content block, which allows us to load different layouts based on which part of the application our user is engaging with: for the `home` `index` page, they will see one layout, and for any views relating to individual sharks, they will see another.

We can repurpose the `sharks` layout for our `endangered` views since this format will also work for presenting shark data in bulk.

Copy the `sharks` layout file over to create an `endangered` layout:

```
cp app/views/layouts/sharks.html.erb app/views/layouts/endange
red.html.erb
```

Next, we'll work on creating the view templates for our `index` and `data` views.

Open the `index` template first:

```
nano app/views/endangered/index.html.erb
```

Delete the boilerplate code and add the following code instead, which will give users some general information about the endangered categories and present them with the option to upload information about endangered sharks:

~/rails-sidekiq/app/views/endangered/index.html.erb

```
<p id="notice"><%= notice %></p>

<h1>Endangered Sharks</h1>

<p>International Union for Conservation of Nature (ICUN) statu
ses: <b>vu:</b> Vulnerable, <b>en:</b> Endangered, <b>cr:</b>
 Critically Endangered </p>

<br>

  <%= form_tag endangered_upload_path do %>
  <%= submit_tag "Import Endangered Sharks" %>
  <% end %>

  <br>

<%= link_to 'New Shark', new_shark_path, :class => "btn btn-pr
imary btn-sm" %> <%= link_to 'Home', home_index_path, :class =
> "btn btn-primary btn-sm" %>
```

A `form_tag` makes the data upload possible by pointing a post action to the `endangered_upload_path` – the route we defined for our uploads. A submit button, created with the `submit_tag`, prompts users to `"Import Endangered Sharks"`.

In addition to this code, we've included some general information about ICUN codes, so that users can interpret the data they will see.

Save and close the file when you are finished editing.

Next, open a file for the `data` view:

```
nano app/views/endangered/data.html.erb
```

Add the following code, which will add a table with the endangered shark data:

~/rails-sidekiq/app/views/endangered/data.html.erb

```
<p id="notice"><%= notice %></p>

<h1>Endangered Sharks</h1>

<p>International Union for Conservation of Nature (ICUN) statu
ses: <b>vu:</b> Vulnerable, <b>en:</b> Endangered, <b>cr:</b>
 Critically Endangered </p>

<div class="table-responsive">
<table class="table table-striped table-dark">
  <thead>
    <tr>
      <th>Name</th>
      <th>IUCN Status</th>
      <th colspan="3"></th>
    </tr>
  </thead>

  <tbody>
    <% @endangered.each do |shark| %>
      <tr>
        <td><%= shark.name %></td>
        <td><%= shark.iucn %></td>
      </tr>
```

```
    <% end %>
  </tbody>
</table>
</div>

<br>

  <%= form_tag endangered_destroy_path do %>
  <%= submit_tag "Delete Endangered Sharks" %>
  <% end %>

  <br>

<%= link_to 'New Shark', new_shark_path, :class => "btn btn-pr
imary btn-sm" %> <%= link_to 'Home', home_index_path, :class =
> "btn btn-primary btn-sm" %>
```

This code includes the ICUN status codes once again, and a Bootstrap table for the outputted data. By looping through our `@endangered` variable, we output the name and ICUN status of each shark to the table.

Below the table, we have another set of `form_tags` and `submit_tags`, which post to the `destroy` path by offering users the option to `"Delete Endangered Sharks"`.

Save and close the file when you are finished editing.

The last modification we'll make to our views will be in the `index` view associated with our `home` controller. You may recall that this view is set as the root of the application in `config/routes.rb`.

Open this file for editing:

```
nano app/views/home/index.html.erb
```

Find the column in the row that states `Sharks are ancient`:

```
~/rails-sidekiq/app/views/home/index.html.erb
. . .
        <div class="col-lg-6">
            <h3>Sharks are ancient</h3>
            <p>There is evidence to suggest that sharks lived
 up to 400 million years ago.
            </p>
        </div>
    </div>
</div>
```

Add the following code to the file:

```
~/rails-sidekiq/app/views/home/index.html.erb
. . .
        <div class="col-lg-6">
            <h3>Sharks are ancient and SOME are in danger</h3>
            <p>There is evidence to suggest that sharks lived
 up to 400 million years ago. Without our help, some could dis
appear soon.</p>
            <p><%= button_to 'Which Sharks Are in Danger?', en
dangered_index_path, :method => :get,  :class => "btn btn-prim
ary btn-sm"%>
            </p>
        </div>
    </div>
</div>
```

We've included a call to action for users to learn more about endangered sharks, by first sharing a strong message, and then adding a `button_to` helper that submits a GET request to our `endangered` `index` route, giving users access to that part of the application. From there, they will be able to upload and view endangered shark information.

Save and close the file when you are finished editing.

With your code in place, you are ready to start the application and upload some sharks!

Step 5 — Starting Sidekiq and Testing the Application

Before we start the application, we'll need to run migrations on our database and start Sidekiq to enable our workers. Redis should already be running on the server, but we can check to be sure. With all of these things in place, we'll be ready to test the application.

First, check that Redis is running:

```
systemctl status redis
```

You should see output like the following:

```
Output
● redis-server.service - Advanced key-value store
   Loaded: loaded (/lib/systemd/system/redis-server.service; e
nabled; vendor preset: enabled)
   Active: active (running) since Tue 2019-11-12 20:37:13 UTC;
1 weeks 0 days ago
```

Next, run your database migrations:

```
rails db:migrate
```

You can now start Sidekiq in the context of your current project bundle by using the `bundle exec sidekiq` command:

```
bundle exec sidekiq
```

You will see output like this, indicating that Sidekiq is ready to process jobs:

Output

```
               m,
               `$b
          .ss,  $$:         .,d$
          `$$P,d$P'    .,md$P"'
           ,$$$$$b/md$$$P^'
         .d$$$$$$/$$$P'
         $$^' `"/$$$'       ____  _     _      _    _
         $:     ,$$:      / ___|(_) __| | ___| | _(_) __ _
         `b     :$$       \___ \| |/ _` |/ _ \ |/ / |/ _` |
                $$:        ___) | | (_| |  __/   <| | (_| |
                $$        |____/|_|\__,_|\___|_|\_\_|\__, |
              .d$$                                      |_|


2019-11-19T21:43:00.540Z pid=17621 tid=gpiqiesdl INFO: Running
in ruby 2.5.1p57 (2018-03-29 revision 63029) [x86_64-linux]
2019-11-19T21:43:00.540Z pid=17621 tid=gpiqiesdl INFO: See LIC
ENSE and the LGPL-3.0 for licensing details.
2019-11-19T21:43:00.540Z pid=17621 tid=gpiqiesdl INFO: Upgrade
to Sidekiq Pro for more features and support: http://sidekiq.o
rg
2019-11-19T21:43:00.540Z pid=17621 tid=gpiqiesdl INFO: Booting
Sidekiq 6.0.3 with redis options {:id=>"Sidekiq-server-PID-176
```

```
21", :url=>nil}
2019-11-19T21:43:00.543Z pid=17621 tid=gpiqiesdl INFO: Startin
g processing, hit Ctrl-C to stop
```

Open a second terminal window, navigate to the `rails-sidekiq` directory, and start your application server.

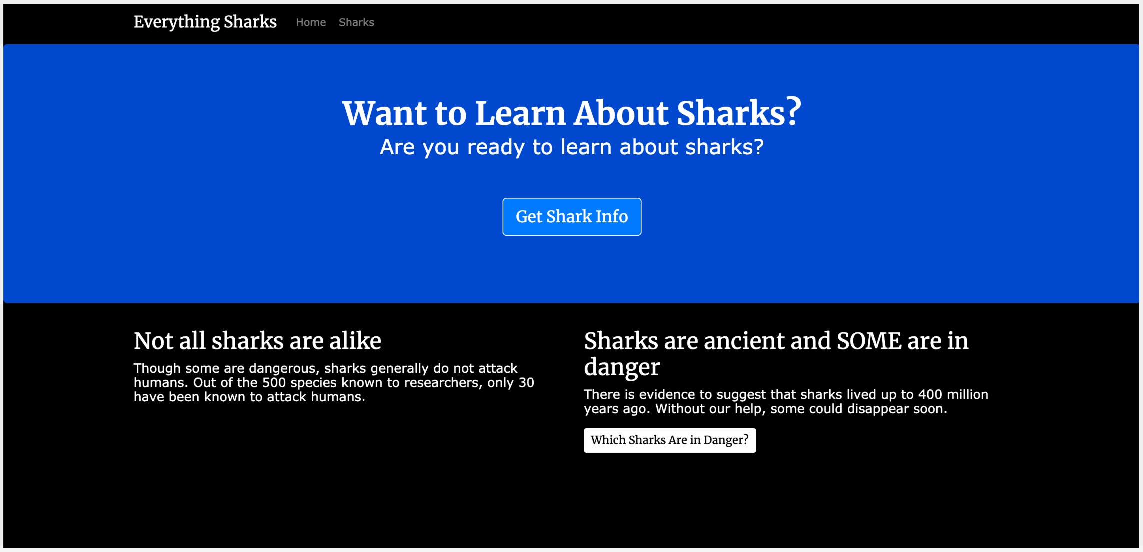
If you are running the application locally, use the following command:

```
[environment second]
bundle exec rails s
```

If you are working with a development server, run the following:

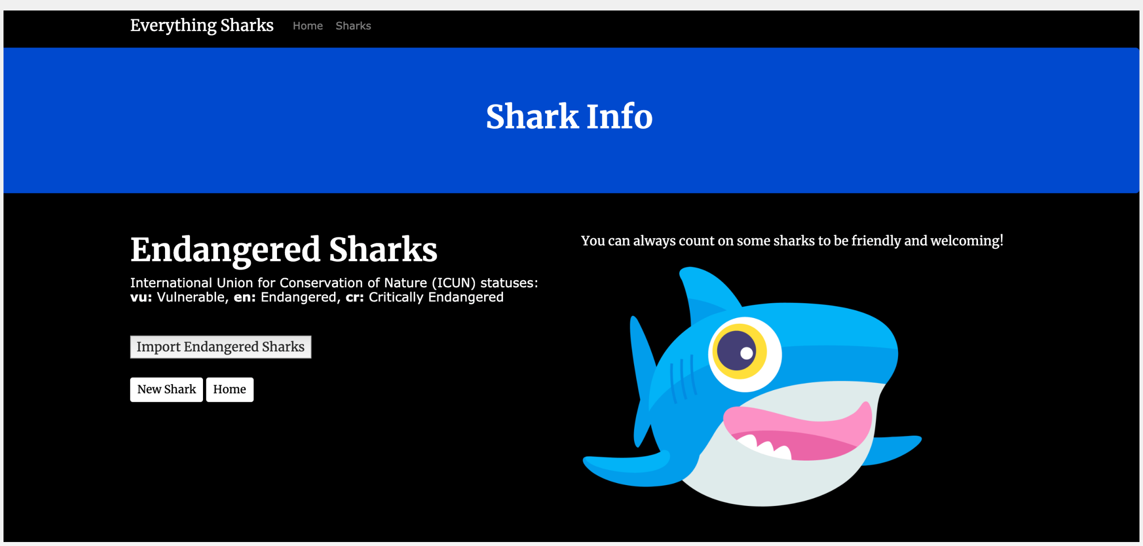
```
[environment second]
bundle exec rails s --binding=your_server_ip
```

Navigate to `localhost:3000` or `http://your_server_ip:3000` in the browser. You will see the following landing page:



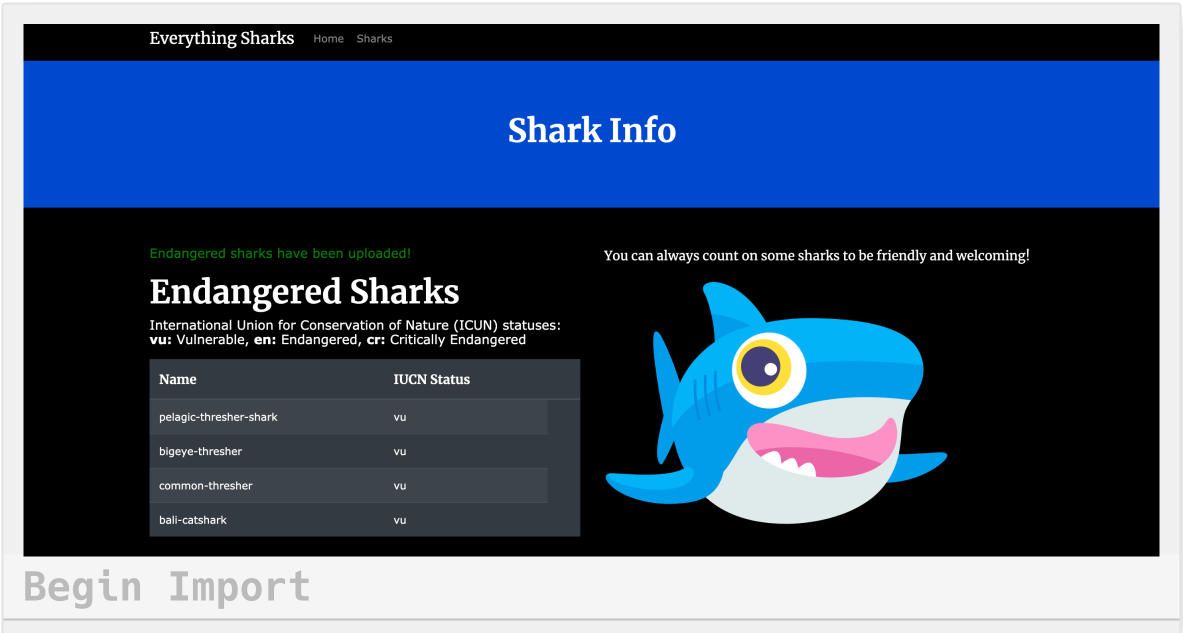
Sidekiq App Home

Click on the **Which Sharks Are in Danger?** button. Since you have not uploaded any endangered sharks, this will take you to the `endangered index` view:

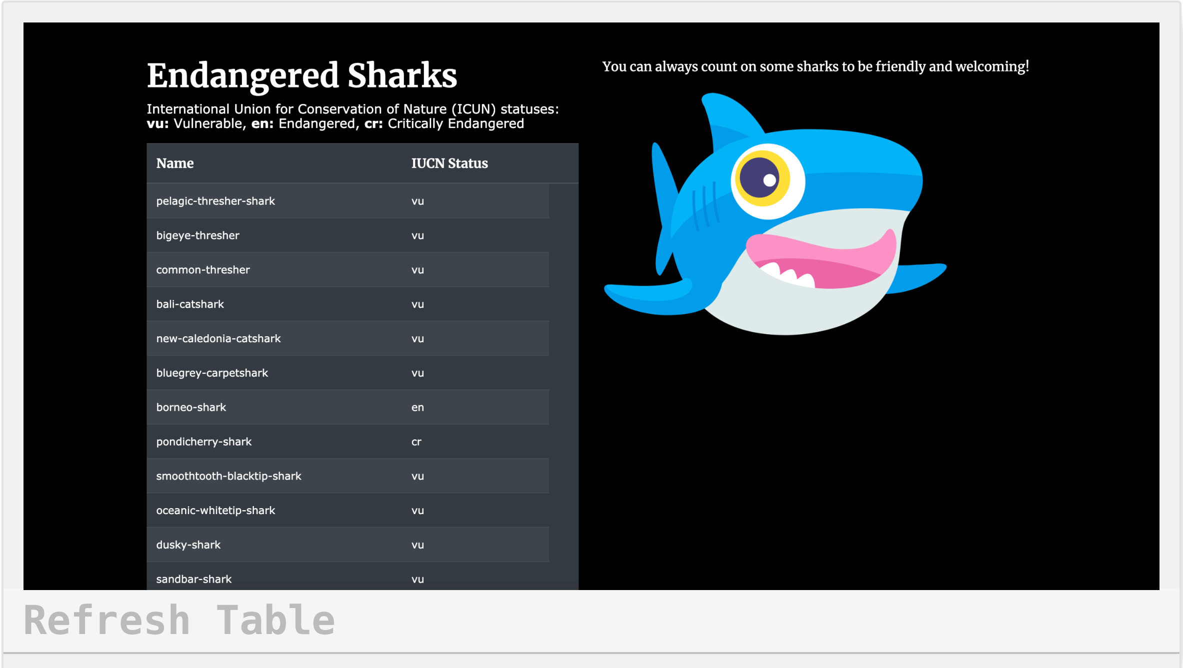


Endangered Index View

Click on **Import Endangered Sharks** to import the sharks. You will see a status message telling you that the sharks have been imported:

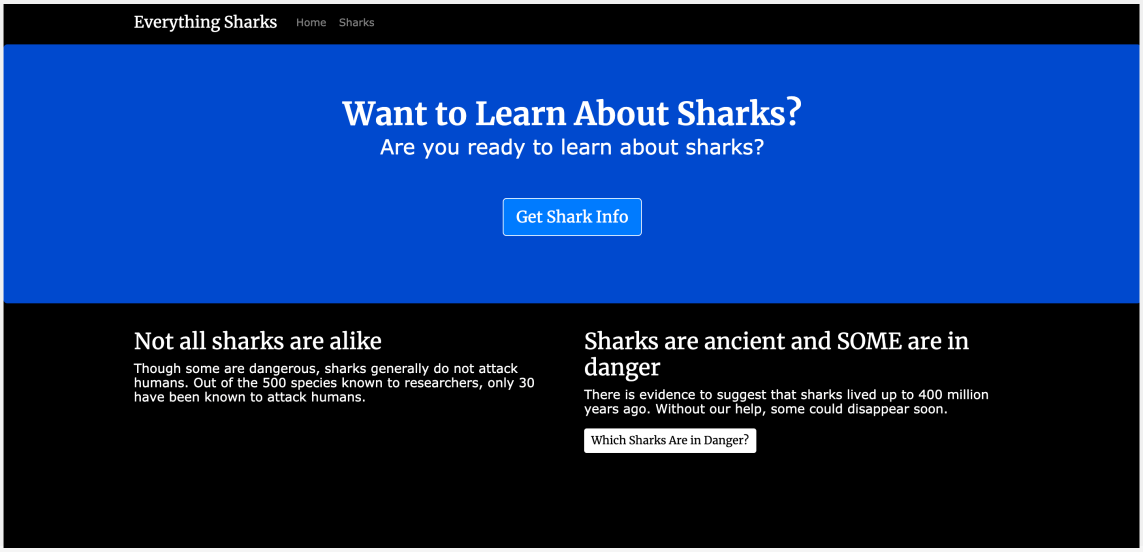


You will also see the beginning of the import. Refresh your page to see the entire table:



Thanks to Sidekiq, our large batch upload of endangered sharks has succeeded without locking up the browser or interfering with other application functionality.

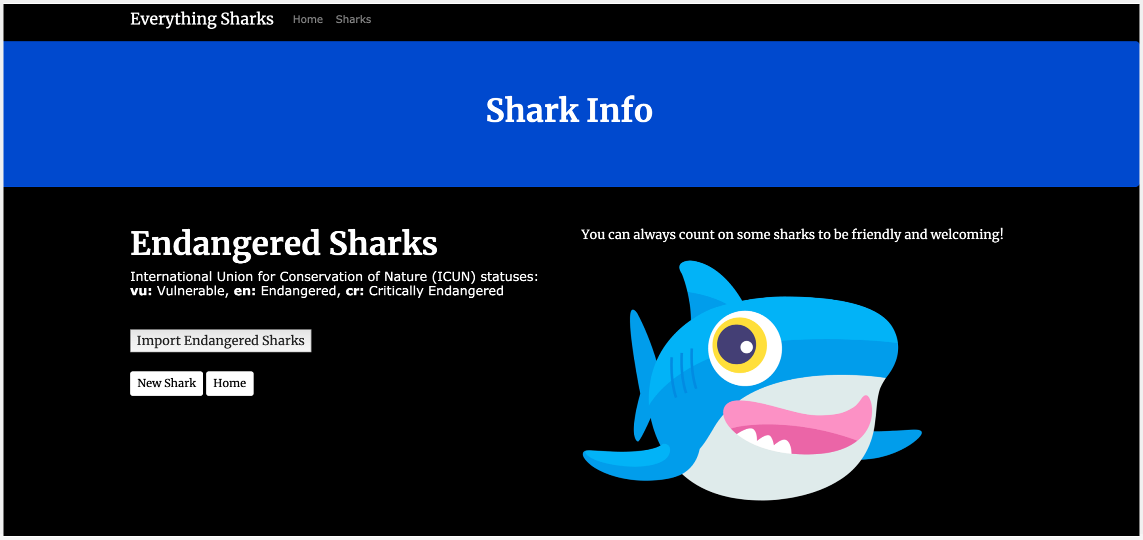
Click on the **Home** button at the bottom of the page, which will bring you back to the application main page:



Sidekiq App Home

From here, click on **Which Sharks Are in Danger?** again. This will now take you directly to the `data` view, since you already uploaded the sharks.

To test the delete functionality, click on the **Delete Endangered Sharks** button below the table. You should be redirected to the home application page once again. Clicking on **Which Sharks Are in Danger?** one last time will take you back to the `index` view, where you will have the option to upload sharks again:



Endangered Index View

Your application is now running with Sidekiq workers in place, which are ready to process jobs and ensure that users have a good experience working with your application.

Conclusion

You now have a working Rails application with Sidekiq enabled, which will allow you to offload costly operations to a job queue managed by Sidekiq and backed by Redis. This will allow you to improve your site's speed and functionality as you develop.

If you would like to learn more about Sidekiq, the docs are a good place to start.

To learn more about Redis, check out our library of Redis resources. You can also learn more about running a managed Redis cluster on DigitalOcean by looking at the product documentation.

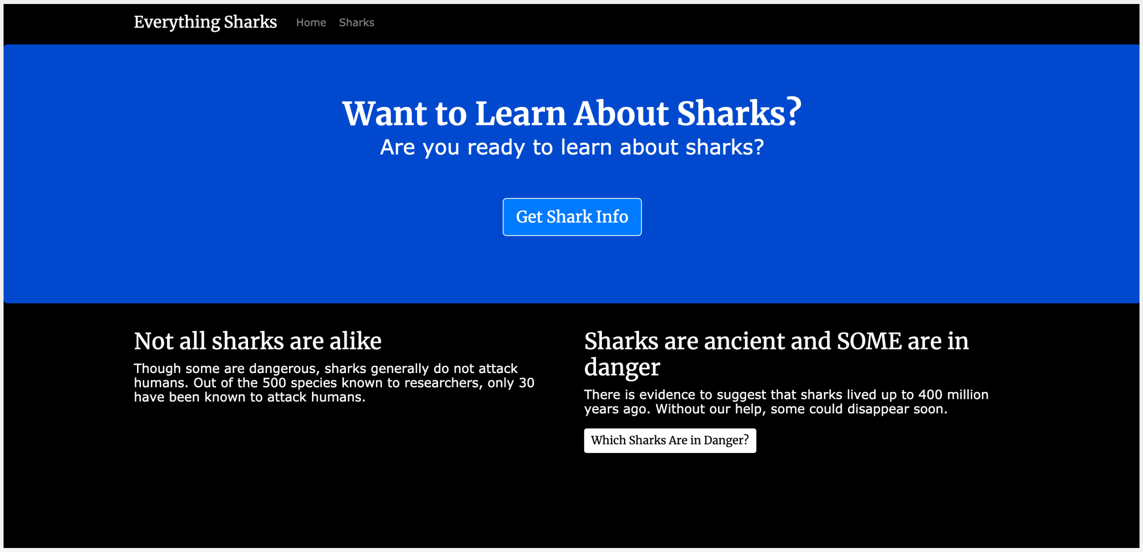
Containerizing a Ruby on Rails Application for Development with Docker Compose

Written by Kathleen Juell

If you are actively developing an application, using [Docker](https://www.docker.com/) can simplify your workflow and the process of deploying your application to production. Working with containers in development offers the following benefits: - Environments are consistent, meaning that you can choose the languages and dependencies you want for your project without worrying about system conflicts. - Environments are isolated, making it easier to troubleshoot issues and onboard new team members. - Environments are portable, allowing you to package and share your code with others.

This tutorial will show you how to set up a development environment for a [Ruby on Rails](https://rubyonrails.org/) application using Docker. You will create multiple containers – for the application itself, the [PostgreSQL](https://www.postgresql.org/) database, [Redis](https://redis.io/), and a [Sidekiq](https://sidekiq.org/) service – with [Docker Compose](https://docs.docker.com/compose/). The setup will do the following: - Synchronize the application code on the host with the code in the container to facilitate changes during development. - Persist application data between container restarts. - Configure Sidekiq workers to process jobs as expected.

At the end of this tutorial, you will have a working shark information application running on Docker containers:



Sidekiq App Home

Prerequisites

To follow this tutorial, you will need: - A local development machine or server running Ubuntu 18.04, along with a non-root user with `sudo` privileges and an active firewall. For guidance on how to set these up, please see this [Initial Server Setup guide](). - Docker installed on your local machine or server, following Steps 1 and 2 of [How To Install and Use Docker on Ubuntu 18.04](). - Docker Compose installed on your local machine or server, following Step 1 of [How To Install Docker Compose on Ubuntu 18.04]().

Step 1 — Cloning the Project and Adding Dependencies

Our first step will be to clone the [rails-sidekiq]() repository from the [DigitalOcean Community GitHub account](). This repository includes the code from the setup described in [How To Add Sidekiq and Redis to a Ruby]()

on Rails Application, which explains how to add Sidekiq to an existing Rails 5 project.

Clone the repository into a directory called `rails-docker`:

```
git clone https://github.com/do-community/rails-sidekiq.git rails-docker
```

Navigate to the `rails-docker` directory:

```
cd rails-docker
```

In this tutorial we will use PostgreSQL as a database. In order to work with PostgreSQL instead of SQLite 3, you will need to add the pg gem to the project's dependencies, which are listed in its Gemfile. Open that file for editing using `nano` or your favorite editor:

```
nano Gemfile
```

Add the gem anywhere in the main project dependencies (above development dependencies):

```
~/rails-docker/Gemfile
. . .
# Reduces boot times through caching; required in config/boot.
rb
gem 'bootsnap', '>= 1.1.0', require: false
gem 'sidekiq', '~>6.0.0'
gem 'pg', '~>1.1.3'

group :development, :test do
. . .
```

We can also comment out the [sqlite gem](), since we won't be using it anymore:

```
~/rails-docker/Gemfile
. . .
# Use sqlite3 as the database for Active Record
# gem 'sqlite3'
. . .
```

Finally, comment out the [spring-watcher-listen gem]() under `development`:

```
~/rails-docker/Gemfile
. . .
gem 'spring'
# gem 'spring-watcher-listen', '~> 2.0.0'
. . .
```

If we do not disable this gem, we will see persistent error messages when accessing the Rails console. These error messages derive from the fact that this gem has Rails use `listen` to watch for changes in development, rather than polling the filesystem for changes. Because this gem watches the root of the project, including the `node_modules` directory, it will throw error messages about which directories are being watched, cluttering the console. If you are concerned about conserving CPU resources, however, disabling this gem may not work for you. In this case, it may be a good idea to upgrade your Rails application to Rails 6.

Save and close the file when you are finished editing.

With your project repository in place, the `pg` gem added to your Gemfile, and the `spring-watcher-listen` gem commented out, you are ready to configure your application to work with PostgreSQL.

Step 2 — Configuring the Application to Work with PostgreSQL and Redis

To work with PostgreSQL and Redis in development, we will want to do the following: - Configure the application to work with PostgreSQL as the default adapter. - Add an `.env` file to the project with our database username and password and Redis host. - Create an `init.sql` script to create a `sammy` user for the database. - Add an [initializer](#) for Sidekiq so that it can work with our containerized `redis` service. - Add the `.env` file and other relevant files to the project's `gitignore` and `dockerignore` files. - Create database seeds so that our application has some records for us to work with when we start it up.

First, open your database configuration file, located at `config/database.yml`:

```
nano config/database.yml
```

Currently, the file includes the following `default` settings, which are applied in the absence of other settings:

```
~/rails-docker/config/database.yml
default: &default
  adapter: sqlite3
  pool: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>
  timeout: 5000
```

We need to change these to reflect the fact that we will use the `postgresql` adapter, since we will be creating a PostgreSQL service with Docker Compose to persist our application data.

Delete the code that sets SQLite as the adapter and replace it with the following settings, which will set the adapter appropriately and the other variables necessary to connect:

~/rails-docker/config/database.yml

```
default: &default
  adapter: postgresql
  encoding: unicode
  database: <%= ENV['DATABASE_NAME'] %>
  username: <%= ENV['DATABASE_USER'] %>
  password: <%= ENV['DATABASE_PASSWORD'] %>
  port: <%= ENV['DATABASE_PORT'] || '5432' %>
  host: <%= ENV['DATABASE_HOST'] %>
  pool: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>
  timeout: 5000
. . .
```

Next, we'll modify the setting for the `development` environment, since this is the environment we're using in this setup.

Delete the existing SQLite database configuration so that section looks like this:

~/rails-docker/config/database.yml

```
. . .
development:
  <<: *default
. . .
```

Finally, delete the `database` settings for the `production` and `test` environments as well:

~/rails-docker/config/database.yml

```
. . .
test:
  <<: *default

production:
  <<: *default
. . .
```

These modifications to our default database settings will allow us to set our database information dynamically using environment variables defined in `.env` files, which will not be committed to version control.

Save and close the file when you are finished editing.

Note that if you are creating a Rails project from scratch, you can set the adapter with the `rails new` command, as described in [Step 3](#) of [How To](#)

[Use PostgreSQL with Your Ruby on Rails Application on Ubuntu 18.04](). This will set your adapter in `config/database.yml` and automatically add the `pg` gem to the project.

Now that we have referenced our environment variables, we can create a file for them with our preferred settings. Extracting configuration settings in this way is part of the [12 Factor approach]() to application development, which defines best practices for application resiliency in distributed environments. Now, when we are setting up our production and test environments in the future, configuring our database settings will involve creating additional `.env` files and referencing the appropriate file in our Docker Compose files.

Open an `.env` file:

```
nano .env
```

Add the following values to the file:

```
~/rails-docker/.env
DATABASE_NAME=rails_development
DATABASE_USER=sammy
DATABASE_PASSWORD=shark
DATABASE_HOST=database
REDIS_HOST=redis
```

In addition to setting our database name, user, and password, we've also set a value for the `DATABASE_HOST`. The value, `database`, refers to the `database` PostgreSQL service we will create using Docker Compose. We've also set a `REDIS_HOST` to specify our `redis` service.

Save and close the file when you are finished editing.

To create the `sammy` database user, we can write an `init.sql` script that we can then mount to the database container when it starts.

Open the script file:

```
nano init.sql
```

Add the following code to create a `sammy` user with administrative privileges:

~/rails-docker/init.sql

```
CREATE USER sammy;
ALTER USER sammy WITH SUPERUSER;
```

This script will create the appropriate user on the database and grant this user administrative privileges.

Set appropriate permissions on the script:

```
chmod +x init.sql
```

Next, we'll configure Sidekiq to work with our containerized `redis` service. We can add an initializer to the `config/initializers` directory, where Rails looks for configuration settings once frameworks and plugins are loaded, that sets a value for a Redis host.

Open a `sidekiq.rb` file to specify these settings:

```
nano config/initializers/sidekiq.rb
```

Add the following code to the file to specify values for a `REDIS_HOST` and `REDIS_PORT`:

```
~/rails-docker/config/initializers/sidekiq.rb
Sidekiq.configure_server do |config|
  config.redis = {
    host: ENV['REDIS_HOST'],
    port: ENV['REDIS_PORT'] || '6379'
  }
end

Sidekiq.configure_client do |config|
  config.redis = {
    host: ENV['REDIS_HOST'],
    port: ENV['REDIS_PORT'] || '6379'
  }
end
```

Much like our database configuration settings, these settings give us the ability to set our host and port parameters dynamically, allowing us to substitute the appropriate values at runtime without having to modify the application code itself. In addition to a `REDIS_HOST`, we have a default value set for `REDIS_PORT` in case it is not set elsewhere.

Save and close the file when you are finished editing.

Next, to ensure that our application's sensitive data is not copied to version control, we can add `.env` to our project's `.gitignore` file, which tells Git which files to ignore in our project. Open the file for editing:

```
nano .gitignore
```

At the bottom of the file, add an entry for `.env`:

```
~/rails-docker/.gitignore
yarn-debug.log*
.yarn-integrity
.env
```

Save and close the file when you are finished editing.

Next, we'll create a `.dockerignore` file to set what should not be copied to our containers. Open the file for editing:

```
.dockerignore
```

Add the following code to the file, which tells Docker to ignore some of the things we don't need copied to our containers:

~/rails-docker/.dockerignore

```
.DS_Store
.bin
.git
.gitignore
.bundleignore
.bundle
.byebug_history
.rspec
tmp
log
test
config/deploy
public/packs
public/packs-test
node_modules
yarn-error.log
coverage/
```

Add `.env` to the bottom of this file as well:

```
~/rails-docker/.dockerignore
. . .
yarn-error.log
coverage/
.env
```

Save and close the file when you are finished editing.

As a final step, we will create some seed data so that our application has a few records when we start it up.

Open a file for the seed data in the `db` directory:

```
nano db/seeds.rb
```

Add the following code to the file to create four demo sharks and one sample post:

```
~/rails-docker/db/seeds.rb
# Adding demo sharks
sharks = Shark.create([{ name: 'Great White', facts: 'Scary'
 }, { name: 'Megalodon', facts: 'Ancient' }, { name: 'Hammerhe
ad', facts: 'Hammer-like' }, { name: 'Speartooth', facts: 'End
angered' }])
Post.create(body: 'These sharks are misunderstood', shark: sha
rks.first)
```

This seed data will create four sharks and one post that is associated with the first shark.

Save and close the file when you are finished editing.

With your application configured to work with PostgreSQL and your environment variables created, you are ready to write your application Dockerfile.

Step 3 — Writing the Dockerfile and Entrypoint Scripts

Your Dockerfile specifies what will be included in your application container when it is created. Using a Dockerfile allows you to define your container environment and avoid discrepancies with dependencies or runtime versions.

Following these [guidelines on building optimized containers](), we will make our image as efficient as possible by using an [Alpine base]() and attempting to minimize our image layers generally.

Open a Dockerfile in your current directory:

```
nano Dockerfile
```

Docker images are created using a succession of layered images that build on one another. Our first step will be to add the base image for our application, which will form the starting point of the application build.

Add the following code to the file to add the [Ruby alpine image]() as a base:

```
~/rails-docker/Dockerfile
FROM ruby:2.5.1-alpine
```

The `alpine` image is derived from the Alpine Linux project, and will help us keep our image size down. For more information about whether or not the `alpine` image is the right choice for your project, please see the full discussion under the **Image Variants** section of the [Docker Hub Ruby image page].

Some factors to consider when using `alpine` in development: - Keeping image size down will decrease page and resource load times, particularly if you also keep volumes to a minimum. This helps keep your user experience in development quick and closer to what it would be if you were working locally in a non-containerized environment. - Having parity between development and production images facilitates successful deployments. Since teams often opt to use Alpine images in production for speed benefits, developing with an Alpine base helps offset issues when moving to production.

Next, set an environment variable to specify the [Bundler] version:

```
~/rails-docker/Dockerfile
. . .
ENV BUNDLER_VERSION=2.0.2
```

This is one of the steps we will take to avoid version conflicts between the default `bundler` version available in our environment and our application code, which requires Bundler 2.0.2.

Next, add the packages that you need to work with the application to the Dockerfile:

~/rails-docker/Dockerfile

```
. . .
RUN apk add --update --no-cache \
      binutils-gold \
      build-base \
      curl \
      file \
      g++ \
      gcc \
      git \
      less \
      libstdc++ \
      libffi-dev \
      libc-dev \
      linux-headers \
      libxml2-dev \
      libxslt-dev \
      libgcrypt-dev \
      make \
      netcat-openbsd \
      nodejs \
      openssl \
      pkgconfig \
      postgresql-dev \
      python \
```

```
    tzdata \
    yarn
```

These packages include `nodejs` and `yarn`, among others. Since our application serves assets with webpack, we need to include Node.js and Yarn for the application to work as expected.

Keep in mind that the `alpine` image is extremely minimal: the packages listed here are not exhaustive of what you might want or need in development when you are containerizing your own application.

Next, install the appropriate `bundler` version:

~/rails-docker/Dockerfile

```
. . .
RUN gem install bundler -v 2.0.2
```

This step will guarantee parity between our containerized environment and the specifications in this project's `Gemfile.lock` file.

Now set the working directory for the application on the container:

~/rails-docker/Dockerfile

```
. . .
WORKDIR /app
```

Copy over your `Gemfile` and `Gemfile.lock`:

```
~/rails-docker/Dockerfile
. . .
COPY Gemfile Gemfile.lock ./
```

Copying these files as an independent step, followed by `bundle install`, means that the project gems do not need to be rebuilt every time you make changes to your application code. This will work in conjunction with the gem volume that we will include in our Compose file, which will mount gems to your application container in cases where the service is recreated but project gems remain the same.

Next, set the configuration options for the `nokogiri` gem build:

```
~/rails-docker/Dockerfile
. . .
RUN bundle config build.nokogiri --use-system-libraries
. . .
```

This step builds `nokigiri` [with the `libxml2` and `libxslt` library versions](#) that we added to the application container in the `RUN apk add…` step above.

Next, install the project gems:

```
~/rails-docker/Dockerfile
. . .
RUN bundle check || bundle install
```

This instruction checks that the gems are not already installed before installing them.

Next, we'll repeat the same procedure that we used with gems with our JavaScript packages and dependencies. First we'll copy package metadata, then we'll install dependencies, and finally we'll copy the application code into the container image.

To get started with the Javascript section of our Dockerfile, copy `package.json` and `yarn.lock` from your current project directory on the host to the container:

```
~/rails-docker/Dockerfile
. . .
COPY package.json yarn.lock ./
```

Then install the required packages with `yarn install`:

```
~/rails-docker/Dockerfile
. . .
RUN yarn install --check-files
```

This instruction includes a `--check-files` flag with the `yarn` command, a feature that makes sure any previously installed files have not been removed. As in the case of our gems, we will manage the persistence of the packages in the `node_modules` directory with a volume when we write our Compose file.

Finally, copy over the rest of the application code and start the application with an entrypoint script:

~/rails-docker/Dockerfile

```
. . .
COPY . ./


ENTRYPOINT ["./entrypoints/docker-entrypoint.sh"]
```

Using an entrypoint script allows us to [run the container as an executable](https://docs.docker.com/engine/reference/run/#entrypoint-default-command-to-execute-at-runtime).

The final Dockerfile will look like this:

~/rails-docker/Dockerfile

```
FROM ruby:2.5.1-alpine


ENV BUNDLER_VERSION=2.0.2


RUN apk add --update --no-cache \
      binutils-gold \
      build-base \
      curl \
      file \
      g++ \
      gcc \
      git \
      less \
      libstdc++ \
      libffi-dev \
      libc-dev \
      linux-headers \
      libxml2-dev \
      libxslt-dev \
      libgcrypt-dev \
      make \
      netcat-openbsd \
      nodejs \
      openssl \
      pkgconfig \
```

```
      postgresql-dev \
      python \
      tzdata \
      yarn

RUN gem install bundler -v 2.0.2

WORKDIR /app

COPY Gemfile Gemfile.lock ./

RUN bundle config build.nokogiri --use-system-libraries

RUN bundle check || bundle install

COPY package.json yarn.lock ./

RUN yarn install --check-files

COPY . ./

ENTRYPOINT ["./entrypoints/docker-entrypoint.sh"]
```

Save and close the file when you are finished editing.

Next, create a directory called `entrypoints` for the entrypoint scripts:

```
mkdir entrypoints
```

This directory will include our main entrypoint script and a script for our Sidekiq service.

Open the file for the application entrypoint script:

```
nano entrypoints/docker-entrypoint.sh
```

Add the following code to the file:

```
rails-docker/entrypoints/docker-entrypoint.sh
#!/bin/sh


set -e


if [ -f tmp/pids/server.pid ]; then
  rm tmp/pids/server.pid
fi


bundle exec rails s -b 0.0.0.0
```

The first important line is `set -e`, which tells the `/bin/sh` shell that runs the script to fail fast if there are any problems later in the script. Next, the script checks that `tmp/pids/server.pid` is not present to ensure that there won't be server conflicts when we start the application. Finally, the script starts the Rails server with the `bundle exec rails s` command. We use the

`-b` option with this command to bind the server to all IP addresses rather than to the default, `localhost`. This invocation makes the Rails server route incoming requests to the container IP rather than to the default `localhost`.

Save and close the file when you are finished editing.

Make the script executable:

```
chmod +x entrypoints/docker-entrypoint.sh
```

Next, we will create a script to start our `sidekiq` service, which will process our Sidekiq jobs. For more information about how this application uses Sidekiq, please see How To Add Sidekiq and Redis to a Ruby on Rails Application.

Open a file for the Sidekiq entrypoint script:

```
nano entrypoints/sidekiq-entrypoint.sh
```

Add the following code to the file to start Sidekiq:

```
~/rails-docker/entrypoints/sidekiq-entrypoint.sh
#!/bin/sh


set -e


if [ -f tmp/pids/server.pid ]; then
  rm tmp/pids/server.pid
fi


bundle exec sidekiq
```

This script starts Sidekiq in the context of our application bundle.

Save and close the file when you are finished editing. Make it executable:

```
chmod +x entrypoints/sidekiq-entrypoint.sh
```

With your entrypoint scripts and Dockerfile in place, you are ready to define your services in your Compose file.

Step 4 — Defining Services with Docker Compose

Using Docker Compose, we will be able to run the multiple containers required for our setup. We will define our Compose services in our main `docker-compose.yml` file. A service in Compose is a running container, and service definitions — which you will include in your `docker-compose.yml` file — contain information about how each container image will run. The

Compose tool allows you to define multiple services to build multi-container applications.

Our application setup will include the following services: - The application itself - The PostgreSQL database - Redis - Sidekiq

We will also include a bind mount as part of our setup, so that any code changes we make during development will be immediately synchronized with the containers that need access to this code.

Note that we are not defining a `test` service, since testing is outside of the scope of this tutorial and series, but you could do so by following the precedent we are using here for the `sidekiq` service.

Open the `docker-compose.yml` file:

```
nano docker-compose.yml
```

First, add the application service definition:

```
~/rails-docker/docker-compose.yml
version: '3.4'


services:
  app:
    build:
      context: .
      dockerfile: Dockerfile
    depends_on:
      - database
      - redis
    ports:
      - "3000:3000"
    volumes:
      - .:/app
      - gem_cache:/usr/local/bundle/gems
      - node_modules:/app/node_modules
    env_file: .env
    environment:
      RAILS_ENV: development
```

The `app` service definition includes the following options: - `build`: This defines the configuration options, including the `context` and `dockerfile`, that will be applied when Compose builds the application image. If you wanted to use an existing image from a registry like [Docker Hub](), you could use the [`image` instruction]() instead, with information about your username, repository, and image tag. - `context`: This defines the build context for the image build — in this case, the current project directory. - `dockerfile`: This specifies the `Dockerfile` in your current project directory as the file Compose will use to build the application image. - `depends_on`: This sets up the `database` and `redis` containers first so that they are up and running before `app`. - `ports`: This maps port `3000` on the host to port `3000` on the container. - `volumes`: We are including two types of mounts here: - The first is a [bind mount]() that mounts our application code on the host to the `/app` directory on the container. This will facilitate rapid development, since any changes you make to your host code will be populated immediately in the container. - The second is a named [volume](), `gem_cache`. When the `bundle install` instruction runs in the container, it will install the project gems. Adding this volume means that if you recreate the container, the gems will be mounted to the new container. This mount presumes that there haven't been any changes to the project, so if you do make changes to your project gems in development, you will need to remember to delete this volume before recreating your application service. - The third volume is a named volume for the `node_modules` directory. Rather than having `node_modules` mounted to the host, which can lead to package discrepancies and permissions conflicts in development, this volume will ensure that the packages in this directory are persisted and reflect the current state of the project. Again, if you modify the project's Node dependencies, you will need to remove and recreate this volume. - `env_file`: This tells Compose that we would like to add environment variables from a file called `.env` located in the build context. - `environment`: Using this option allows us to set a non-sensitive environment variable, passing information about the Rails environment to the container.

Next, below the `app` service definition, add the following code to define your `database` service:

~/rails-docker/docker-compose.yml

```
. . .
  database:
    image: postgres:12.1
    volumes:
      - db_data:/var/lib/postgresql/data
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql
```

Unlike the `app` service, the `database` service pulls a `postgres` image directly from [Docker Hub](). Note that we're also pinning the version here, rather than setting it to `latest` or not specifying it (which defaults to `latest`). This way, we can ensure that this setup works with the versions specified here and avoid unexpected surprises with breaking code changes to the image.

We are also including a `db_data` volume here, which will persist our application data in between container starts. Additionally, we've mounted our `init.sql` startup script to the appropriate directory, `docker-entrypoint-initdb.d/` on the container, in order to create our `sammy` database user. After the image entrypoint creates the default `postgres` user and database, it will run any scripts found in the `docker-entrypoint-initdb.d/` directory, which you can use for necessary initialization tasks. For more details, look at the **Initialization scripts** section of the [PostgreSQL image documentation]

Next, add the `redis` service definition:

```
~/rails-docker/docker-compose.yml
. . .
  redis:
    image: redis:5.0.7
```

Like the `database` service, the `redis` service uses an image from Docker Hub. In this case, we are not persisting the Sidekiq job cache.

Finally, add the `sidekiq` service definition:

~/rails-docker/docker-compose.yml

```
. . .
  sidekiq:
    build:
      context: .
      dockerfile: Dockerfile
    depends_on:
      - app
      - database
      - redis
    volumes:
      - .:/app
      - gem_cache:/usr/local/bundle/gems
      - node_modules:/app/node_modules
    env_file: .env
    environment:
      RAILS_ENV: development
    entrypoint: ./entrypoints/sidekiq-entrypoint.sh
```

Our `sidekiq` service resembles our `app` service in a few respects: it uses the same build context and image, environment variables, and volumes. However, it is dependent on the `app`, `redis`, and `database` services, and so will be the last to start. Additionally, it uses an `entrypoint` that will override the entrypoint set in the Dockerfile. This `entrypoint` setting points to `entrypoints/sidekiq-entrypoint.sh`, which includes the appropriate command to start the `sidekiq` service.

As a final step, add the volume definitions below the `sidekiq` service definition:

```
~/rails-docker/docker-compose.yml
. . .
volumes:
  gem_cache:
  db_data:
  node_modules:
```

Our top-level volumes key defines the volumes `gem_cache`, `db_data`, and `node_modules`. When Docker creates volumes, the contents of the volume are stored in a part of the host filesystem, `/var/lib/docker/volumes/`, that's managed by Docker. The contents of each volume are stored in a directory under `/var/lib/docker/volumes/` and get mounted to any container that uses the volume. In this way, the shark information data that our users will create will persist in the `db_data` volume even if we remove and recreate the `database` service.

The finished file will look like this:

~/rails-docker/docker-compose.yml

```yaml
version: '3.4'

services:
  app:
    build:
      context: .
      dockerfile: Dockerfile
    depends_on:
      - database
      - redis
    ports:
      - "3000:3000"
    volumes:
      - .:/app
      - gem_cache:/usr/local/bundle/gems
      - node_modules:/app/node_modules
    env_file: .env
    environment:
      RAILS_ENV: development

  database:
    image: postgres:12.1
    volumes:
      - db_data:/var/lib/postgresql/data
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql
```

```
  redis:
    image: redis:5.0.7

  sidekiq:
    build:
      context: .
      dockerfile: Dockerfile
    depends_on:
      - app
      - database
      - redis
    volumes:
      - .:/app
      - gem_cache:/usr/local/bundle/gems
      - node_modules:/app/node_modules
    env_file: .env
    environment:
      RAILS_ENV: development
    entrypoint: ./entrypoints/sidekiq-entrypoint.sh

volumes:
  gem_cache:
  db_data:
  node_modules:
```

Save and close the file when you are finished editing.

With your service definitions written, you are ready to start the application.

Step 5 — Testing the Application

With your `docker-compose.yml` file in place, you can create your services with the [`docker-compose up`](https://docs.docker.com/compose/reference/up/) command and seed your database. You can also test that your data will persist by stopping and removing your containers with [`docker-compose down`](https://docs.docker.com/compose/reference/down/) and recreating them.

First, build the container images and create the services by running `docker-compose up` with the `-d` flag, which will run the containers in the background:

```
docker-compose up -d
```

You will see output that your services have been created:

```
Output
Creating rails-docker_database_1 ... done
Creating rails-docker_redis_1    ... done
Creating rails-docker_app_1      ... done
Creating rails-docker_sidekiq_1  ... done
```

You can also get more detailed information about the startup processes by displaying the log output from the services:

```
docker-compose logs
```

You will see something like this if everything has started correctly:

```
Output
sidekiq_1   | 2019-12-19T15:05:26.365Z pid=6 tid=grk7r6xly INF
O: Booting Sidekiq 6.0.3 with redis options {:host=>"redis", :
port=>"6379", :id=>"Sidekiq-server-PID-6", :url=>nil}
sidekiq_1   | 2019-12-19T15:05:31.097Z pid=6 tid=grk7r6xly INF
O: Running in ruby 2.5.1p57 (2018-03-29 revision 63029) [x86_6
4-linux-musl]
sidekiq_1   | 2019-12-19T15:05:31.097Z pid=6 tid=grk7r6xly INF
O: See LICENSE and the LGPL-3.0 for licensing details.
sidekiq_1   | 2019-12-19T15:05:31.097Z pid=6 tid=grk7r6xly INF
O: Upgrade to Sidekiq Pro for more features and support: htt
p://sidekiq.org
app_1       | => Booting Puma
app_1       | => Rails 5.2.3 application starting in developme
nt
app_1       | => Run `rails server -h` for more startup option
s
app_1       | Puma starting in single mode...
app_1       | * Version 3.12.1 (ruby 2.5.1-p57), codename: Lla
mas in Pajamas
app_1       | * Min threads: 5, max threads: 5
app_1       | * Environment: development
app_1       | * Listening on tcp://0.0.0.0:3000
app_1       | Use Ctrl-C to stop
. . .
database_1  | PostgreSQL init process complete; ready for star
```

```
t up.
database_1  |
database_1  | 2019-12-19 15:05:20.160 UTC [1] LOG:  starting P
ostgreSQL 12.1 (Debian 12.1-1.pgdg100+1) on x86_64-pc-linux-gn
u, compiled by gcc (Debian 8.3.0-6) 8.3.0, 64-bit
database_1  | 2019-12-19 15:05:20.160 UTC [1] LOG:  listening
 on IPv4 address "0.0.0.0", port 5432
database_1  | 2019-12-19 15:05:20.160 UTC [1] LOG:  listening
 on IPv6 address "::", port 5432
database_1  | 2019-12-19 15:05:20.163 UTC [1] LOG:  listening
 on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
database_1  | 2019-12-19 15:05:20.182 UTC [63] LOG:  database
 system was shut down at 2019-12-19 15:05:20 UTC
database_1  | 2019-12-19 15:05:20.187 UTC [1] LOG:  database s
ystem is ready to accept connections
. . .
redis_1     | 1:M 19 Dec 2019 15:05:18.822 * Ready to accept c
onnections
```

You can also check the status of your containers with `docker-compose ps`:

```
docker-compose ps
```

You will see output indicating that your containers are running:

```
Output
         Name                            Command               Sta
te           Ports
-------------------------------------------------------------------
---------------------------
rails-docker_app_1          ./entrypoints/docker-resta ...   Up
0.0.0.0:3000->3000/tcp
rails-docker_database_1   docker-entrypoint.sh postgres    Up
5432/tcp
rails-docker_redis_1        docker-entrypoint.sh redis ...   Up
6379/tcp
rails-docker_sidekiq_1     ./entrypoints/sidekiq-entr ...   Up
```

Next, create and seed your database and run migrations on it with the following [`docker-compose exec` command](docker-compose exec command):

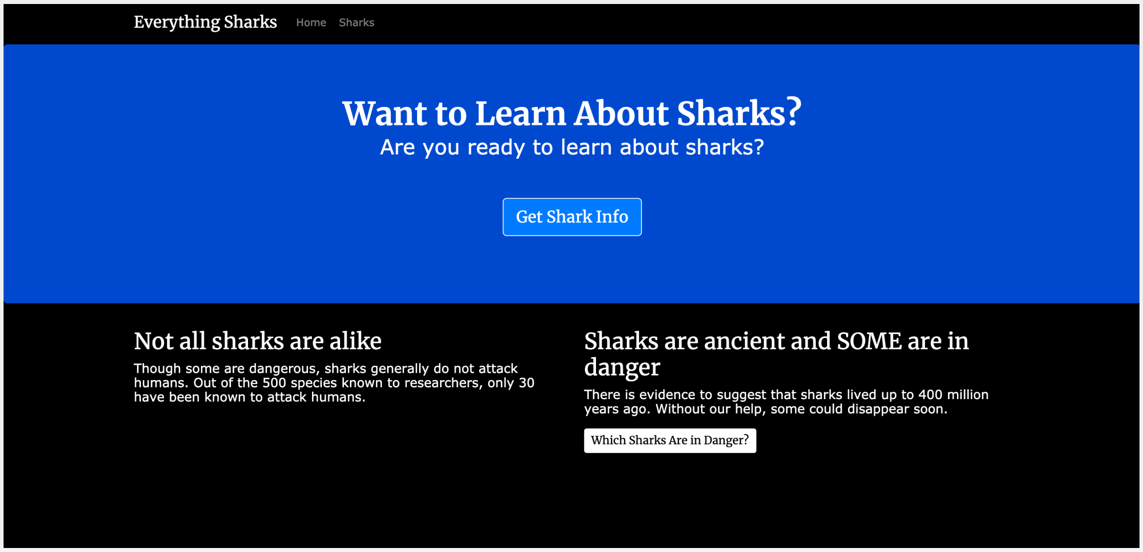
```
docker-compose exec app bundle exec rake db:setup db:migrate
```

The `docker-compose exec` command allows you to run commands in your services; we are using it here to run `rake db:setup` and `db:migrate` in the context of our application bundle to create and seed the database and run migrations. As you work in development, `docker-compose exec` will prove useful to you when you want to run migrations against your development database.

You will see the following output after running this command:

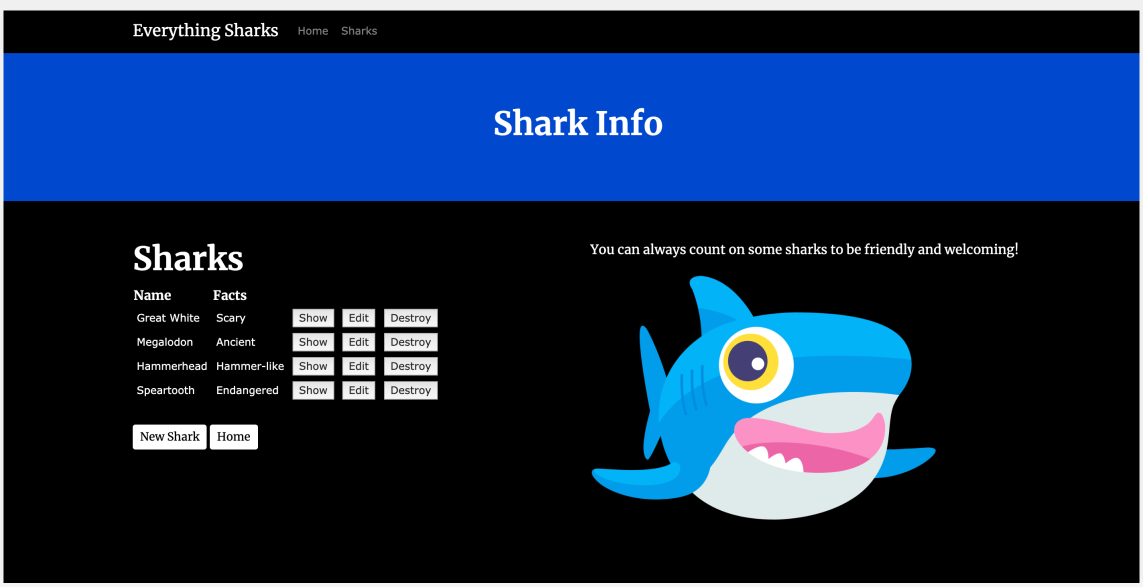
```
Output
Created database 'rails_development'
Database 'rails_development' already exists
-- enable_extension("plpgsql")
   -> 0.0140s
-- create_table("endangereds", {:force=>:cascade})
   -> 0.0097s
-- create_table("posts", {:force=>:cascade})
   -> 0.0108s
-- create_table("sharks", {:force=>:cascade})
   -> 0.0050s
-- enable_extension("plpgsql")
   -> 0.0173s
-- create_table("endangereds", {:force=>:cascade})
   -> 0.0088s
-- create_table("posts", {:force=>:cascade})
   -> 0.0128s
-- create_table("sharks", {:force=>:cascade})
   -> 0.0072s
```

With your services running, you can visit `localhost:3000` or `http://your_server_ip:3000` in the browser. You will see a landing page that looks like this:



Sidekiq App Home

We can now test data persistence. Create a new shark by clicking on **Get Shark Info** button, which will take you to the `sharks/index` route:



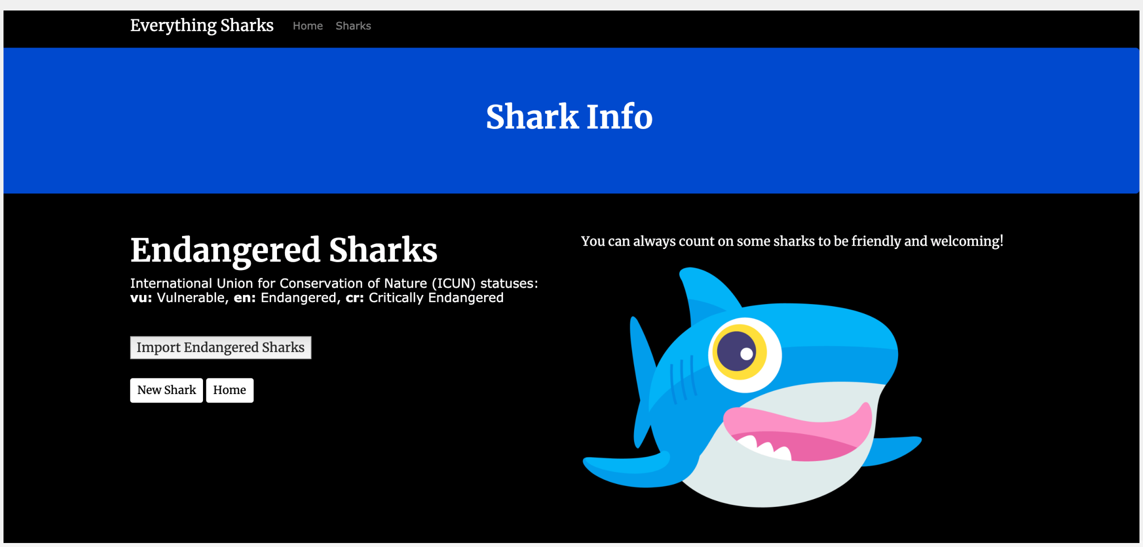
Sharks Index Page with Seeded Data

To verify that the application is working, we can add some demo information to it. Click on **New Shark**. You will be prompted for a username (**sammy**) and password (**shark**), thanks to the project's authentication settings.

On the **New Shark** page, input “Mako” into the **Name** field and “Fast” into the **Facts** field.

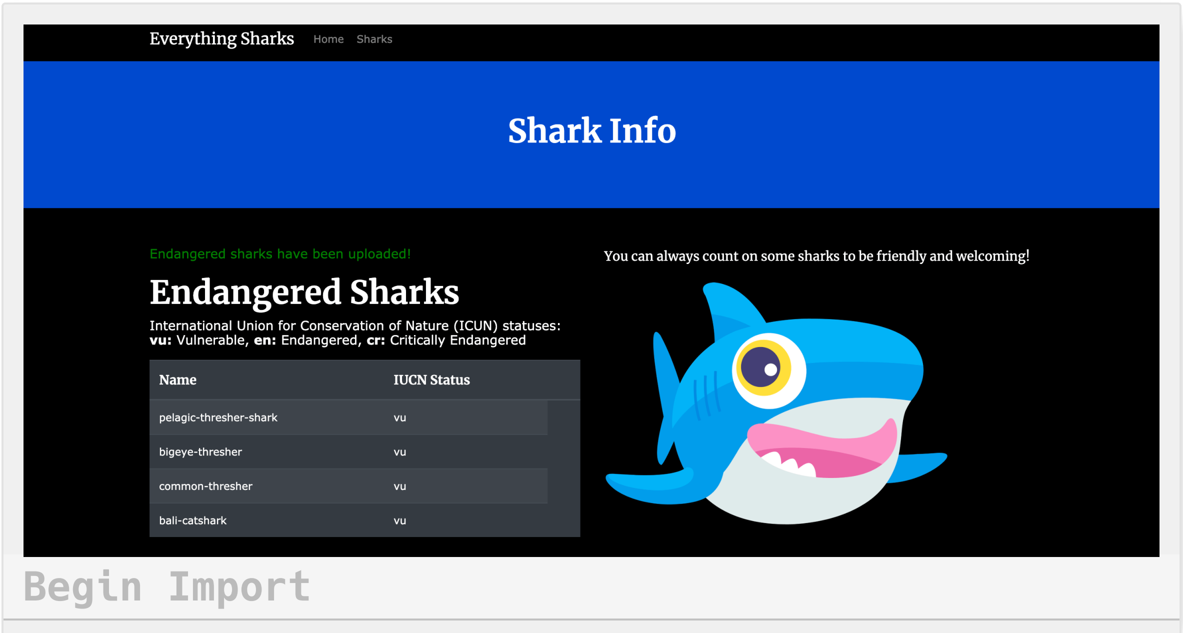
Click on the **Create Shark** button to create the shark. Once you have created the shark, click **Home** on the site's navbar to get back to the main application landing page. We can now test that Sidekiq is working.

Click on the **Which Sharks Are in Danger?** button. Since you have not uploaded any endangered sharks, this will take you to the `endangered` `index` view:

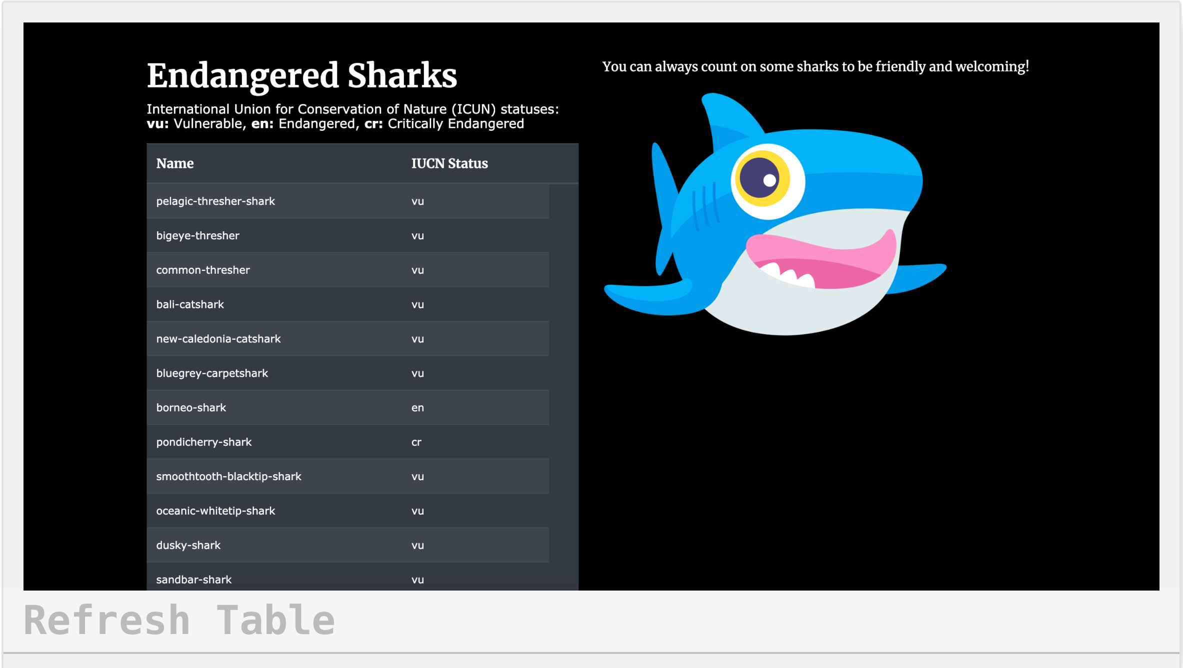


Endangered Index View

Click on **Import Endangered Sharks** to import the sharks. You will see a status message telling you that the sharks have been imported:

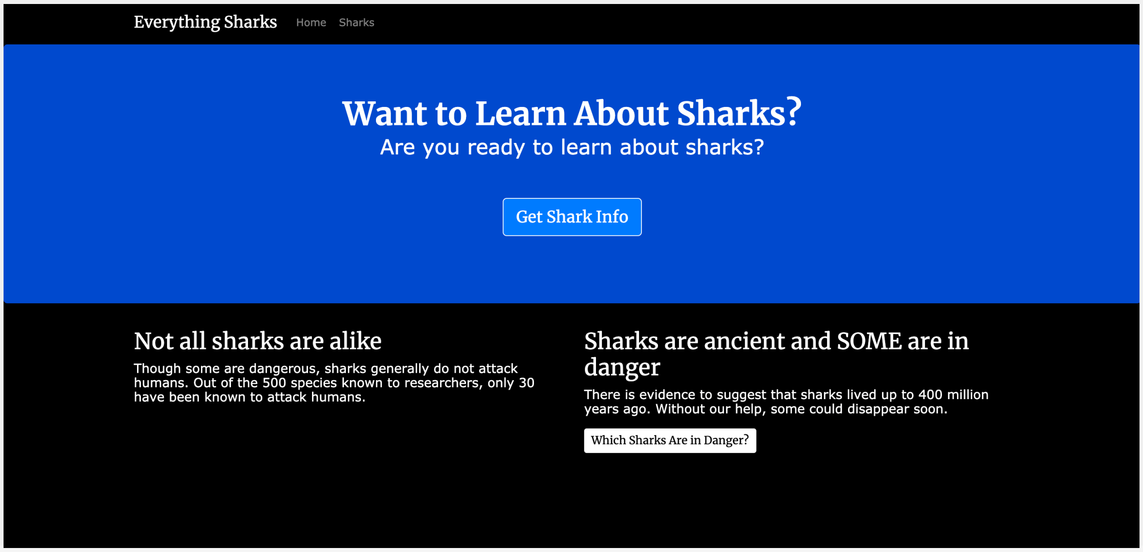


You will also see the beginning of the import. Refresh your page to see the entire table:



Thanks to Sidekiq, our large batch upload of endangered sharks has succeeded without locking up the browser or interfering with other application functionality.

Click on the **Home** button at the bottom of the page, which will bring you back to the application main page:



Sidekiq App Home

From here, click on **Which Sharks Are in Danger?** again. You will see the uploaded sharks once again.

Now that we know our application is working properly, we can test our data persistence.

Back at your terminal, type the following command to stop and remove your containers:

```
docker-compose down
```

Note that we are not including the `--volumes` option; hence, our `db_data` volume is not removed.

The following output confirms that your containers and network have been removed:

```
Output
Stopping rails-docker_sidekiq_1  ... done
Stopping rails-docker_app_1      ... done
Stopping rails-docker_database_1 ... done
Stopping rails-docker_redis_1    ... done
Removing rails-docker_sidekiq_1  ... done
Removing rails-docker_app_1      ... done
Removing rails-docker_database_1 ... done
Removing rails-docker_redis_1    ... done
Removing network rails-docker_default
```

Recreate the containers:

```
docker-compose up -d
```

Open the Rails console on the `app` container with `docker-compose exec` and `bundle exec rails console`:

```
docker-compose exec app bundle exec rails console
```

At the prompt, inspect the `last` Shark record in the database:

```
Shark.last.inspect
```

You will see the record you just created:

IRB session

```
  Shark Load (1.0ms)  SELECT  "sharks".* FROM "sharks" ORDER B
Y "sharks"."id" DESC LIMIT $1  [["LIMIT", 1]]
=> "#<Shark id: 5, name: \"Mako\", facts: \"Fast\", created_a
t: \"2019-12-20 14:03:28\", updated_at: \"2019-12-20 14:03:28
\">"
```

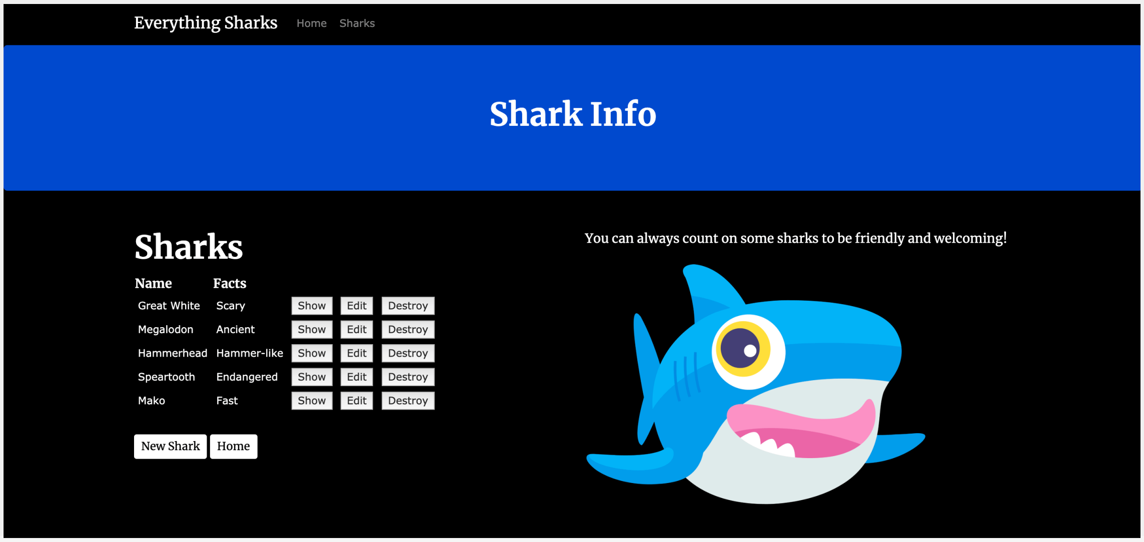
You can then check to see that your `Endangered` sharks have been persisted with the following command:

```
Endangered.all.count
```

IRB session

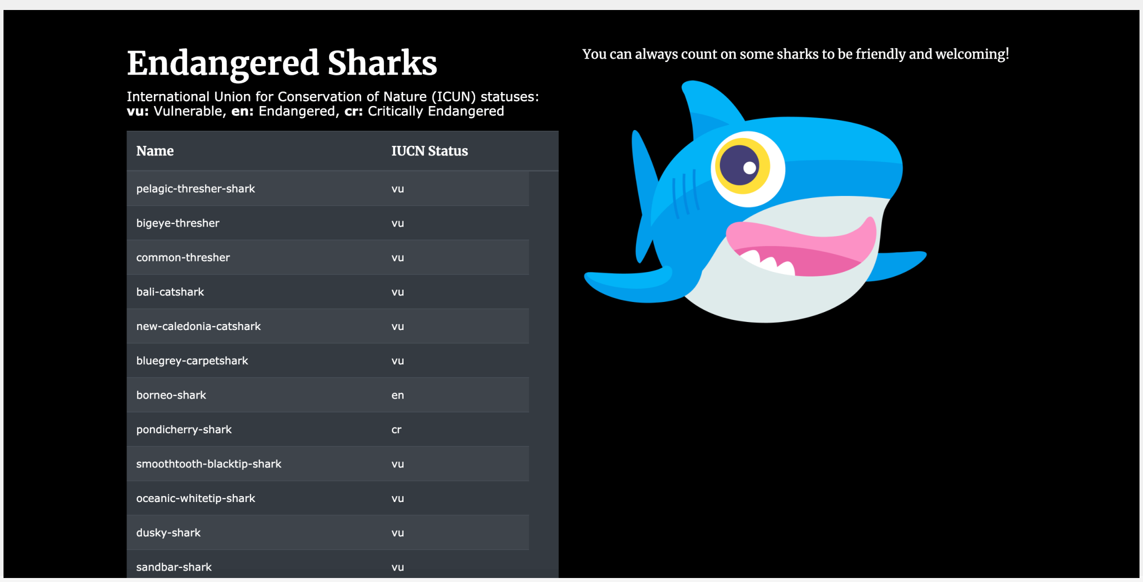
```
   (0.8ms)  SELECT COUNT(*) FROM "endangereds"
=> 73
```

Your `db_data` volume was successfully mounted to the recreated `database` service, making it possible for your `app` service to access the saved data. If you navigate directly to the `index` `shark` page by visiting `localhost:3000/sharks` or `http://your_server_ip:3000/sharks` you will also see that record displayed:



Sharks Index Page with Mako

Your endangered sharks will also be at the `localhost:3000/endangered/data` or `http://your_server_ip:3000/endangered/data` view:



Refresh Table

Your application is now running on Docker containers with data persistence and code synchronization enabled. You can go ahead and test out local code changes on your host, which will be synchronized to your container thanks to the bind mount we defined as part of the `app` service.

Conclusion

By following this tutorial, you have created a development setup for your Rails application using Docker containers. You've made your project more modular and portable by extracting sensitive information and decoupling your application's state from your code. You have also configured a boilerplate `docker-compose.yml` file that you can revise as your development needs and requirements change.

As you develop, you may be interested in learning more about designing applications for containerized and Cloud Native workflows. Please see Architecting Applications for Kubernetes and Modernizing Applications for Kubernetes for more information on these topics. Or, if you would like to invest in a Kubernetes learning sequence, please have a look at out Kubernetes for Full-Stack Developers curriculum.

To learn more about the application code itself, please see the other tutorials in this series: - How To Build a Ruby on Rails Application - How To Create Nested Resources for a Ruby on Rails Application - How To Add Stimulus to a Ruby on Rails Application - How To Add Bootstrap to a Ruby on Rails Application - How To Add Sidekiq and Redis to a Ruby on Rails Application

How To Migrate a Docker Compose Workflow for Rails Development to Kubernetes

Written by Kathleen Juell and Jamon Camisso

When building modern, stateless applications, containerizing your application's components is the first step in deploying and scaling on distributed platforms. If you have used Docker Compose in development, you will have modernized and containerized your application by:

- Extracting necessary configuration information from your code
- Offloading your application's state
- Packaging your application for repeated use.

You will also have written service definitions that specify how your container images should run.

To run your services on a distributed platform like Kubernetes, you will need to translate your Compose service definitions to Kubernetes objects. This will allow you to scale your application with resiliency. One tool that can speed up the translation process to Kubernetes is kompose, a conversion tool that helps developers move Compose workflows to container orchestrators like Kubernetes or OpenShift.

In this tutorial, you will translate Compose services to Kubernetes objects using kompose. You will use the object definitions that kompose provides as a starting point and make adjustments to ensure that your setup will use

Secrets, Services, and PersistentVolumeClaims in the way that Kubernetes expects. By the end of the tutorial, you will have a single-instance Rails application with a PostgreSQL database running on a Kubernetes cluster. This setup will mirror the functionality of the code described in Containerizing a Ruby on Rails Application for Development with Docker Compose and will be a good starting point to build out a production-ready solution that will scale with your needs.

Prerequisites

- A Kubernetes 1.19+ cluster with role-based access control (RBAC) enabled. This setup will use a DigitalOcean Kubernetes cluster, but you are free to create a cluster using another method.
- The `kubectl` command-line tool installed on your local machine or development server and configured to connect to your cluster. You can read more about installing `kubectl` in the official documentation.
- Docker installed on your local machine or development server. If you are working with Ubuntu 20.04, follow Steps 1 and 2 of How To Install and Use Docker on Ubuntu 20.04; otherwise, follow the official documentation for information about installing on other operating systems. Be sure to add your non-root user to the `docker` group, as described in Step 2 of the linked tutorial.
- A Docker Hub account. For an overview of how to set this up, refer to this introduction to Docker Hub.

Step 1 — Installing kompose

To begin using kompose, navigate to the project's GitHub **Releases** page, and copy the link to the current release (version **1.22.0** as of this writing). Paste this link into the following `curl` command to download the latest version of kompose:

```
curl -L https://github.com/kubernetes/kompose/releases/download/v1.22.0/kompose-linux-amd64 -o kompose
```

For details about installing on non-Linux systems, please refer to the installation instructions.

Make the binary executable:

```
chmod +x kompose
```

Move it to your `PATH`:

```
sudo mv ./kompose /usr/local/bin/kompose
```

To verify that it has been installed properly, you can do a version check:

```
kompose version
```

If the installation was successful, you will see output like the following:

```
Output
1.22.0 (955b78124)
```

With `kompose` installed and ready to use, you can now clone the Node.js project code that you will be translating to Kubernetes.

Step 2 — Cloning and Packaging the Application

To use our application with Kubernetes, we will need to clone the project code and package the application so that the `kubelet` service can pull the image.

Our first step will be to clone the [rails-sidekiq repository](#) from the [DigitalOcean Community GitHub account](#). This repository includes the code from the setup described in [Containerizing a Ruby on Rails Application for Development with Docker Compose](#), which uses a demo Rails application to demonstrate how to set up a development environment using Docker Compose. You can find more information about the application itself in the series [Rails on Containers](#).

Clone the repository into a directory called `rails_project`:

```
git clone https://github.com/do-community/rails-sidekiq.git rails_project
```

Navigate to the `rails_project` directory:

```
cd rails_project
```

Now checkout the code for this tutorial from the `compose-workflow` branch:

```
git checkout compose-workflow
```

```
Output
Branch 'compose-workflow' set up to track remote branch 'compo
se-workflow' from 'origin'.
Switched to a new branch 'compose-workflow'
```

The `rails_project` directory contains files and directories for a shark information application that works with user input. It has been modernized to work with containers: sensitive and specific configuration information has been removed from the application code and refactored to be injected at runtime, and the application's state has been offloaded to a PostgreSQL database.

For more information about designing modern, stateless applications, please see [Architecting Applications for Kubernetes]() and [Modernizing Applications for Kubernetes]().

The project directory includes a `Dockerfile` with instructions for building the application image. Let's build the image now so that you can push it to your Docker Hub account and use it in your Kubernetes setup.

Using the [`docker build`]() command, build the image with the `-t` flag, which allows you to tag it with a memorable name. In this case, tag the image with your Docker Hub username and name it `rails-kubernetes` or a name of your own choosing:

```
docker build -t your_dockerhub_user/rails-kubernetes .
```

The `.` in the command specifies that the build context is the current directory.

It will take a minute or two to build the image. Once it is complete, check your images:

```
docker images
```

You will see the following output:

```
Output
REPOSITORY                                  TAG
 IMAGE ID            CREATED             SIZE
your_dockerhub_user/rails-kubernetes     latest                   2
4f7e88b6ef2        2 days ago          606MB
alpine                                       latest
 d6e46aa2470d        6 weeks ago         5.57MB
```

Next, log in to the Docker Hub account you created in the prerequisites:

```
docker login -u your_dockerhub_user
```

When prompted, enter your Docker Hub account password. Logging in this way will create a `~/.docker/config.json` file in your user's home directory with your Docker Hub credentials.

Push the application image to Docker Hub with the [`docker push` command](). Remember to replace `your_dockerhub_user` with your own Docker Hub username:

```
docker push your_dockerhub_user/rails-kubernetes
```

You now have an application image that you can pull to run your application with Kubernetes. The next step will be to translate your application service definitions to Kubernetes objects.

Step 3 — Translating Compose Services to Kubernetes Objects with kompose

Our Docker Compose file, here called `docker-compose.yml`, lays out the definitions that will run our services with Compose. A service in Compose is a running container, and service definitions contain information about how each container image will run. In this step, we will translate these definitions to Kubernetes objects by using `kompose` to create `yaml` files. These files will contain specs for the Kubernetes objects that describe their desired state.

We will use these files to create different types of objects: Services, which will ensure that the Pods running our containers remain accessible; Deployments, which will contain information about the desired state of our Pods; a PersistentVolumeClaim to provision storage for our database data; a ConfigMap for environment variables injected at runtime; and a Secret for our application's database user and password. Some of these definitions will be in the files `kompose` will create for us, and others we will need to create ourselves.

First, we will need to modify some of the definitions in our `docker-compose.yml` file to work with Kubernetes. We will include a reference to our newly-built application image in our `app` service definition and remove the bind mounts, volumes, and additional commands that we used to run the application container in development with Compose. Additionally, we'll redefine both containers' restart policies to be in line with the behavior Kubernetes expects.

If you have followed the steps in this tutorial and checked out the `compose-workflow` branch with git, then you should have a `docker-compose.yml` file in your working directory.

If you don't have a `docker-compose.yml` then be sure to visit the previous tutorial in this series, Containerizing a Ruby on Rails Application for Development with Docker Compose, and paste the contents from the linked section into a new `docker-compose.yml` file.

Open the file with `nano` or your favorite editor:

```
nano docker-compose.yml
```

The current definition for the `app` application service looks like this:

~/rails_project/docker-compose.yml

```
. . .
services:
  app:
    build:
      context: .
      dockerfile: Dockerfile
    depends_on:
      - database
      - redis
    ports:
      - "3000:3000"
    volumes:
      - .:/app
      - gem_cache:/usr/local/bundle/gems
      - node_modules:/app/node_modules
    env_file: .env
    environment:
      RAILS_ENV: development
. . .
```

Make the following edits to your service definition:

- Replace the `build:` line with `image: your_dockerhub_user/rails-kubernetes`

- Remove the following `context: .`, and `dockerfile: Dockerfile` lines.
- Remove the `volumes` list.

The finished service definition will now look like this:

~/rails_project/docker-compose.yml

```
. . .
services:
  app:
    image: your_dockerhub_user/rails-kubernetes
    depends_on:
      - database
      - redis
    ports:
      - "3000:3000"
    env_file: .env
    environment:
      RAILS_ENV: development
. . .
```

Next, scroll down to the `database` service definition and make the following edits:

- Remove the `- ./init.sql:/docker-entrypoint-initdb.d/init.sql` volume line. Instead of using values from the local SQL file, we will pass the values for our `POSTGRES_USER` and `POSTGRES_PASSWORD` to the database container using the Secret we will create in Step 4.

- Add a `ports:` section that will make PostgreSQL available inside your Kubernetes cluster on port 5432.
- Add an `environment:` section with a `PGDATA` variable that points to a directory inside `/var/lib/postgresql/data`. This setting is required when PostgreSQL is configured to use block storage, since the database engine expects to find its data files in a sub-directory.

The `database` service definition should look like this when you are finished editing it:

~/rails_project/docker-compose.yml

```
. . .
  database:
    image: postgres:12.1
    volumes:
      - db_data:/var/lib/postgresql/data
    ports:
      - "5432:5432"
    environment:
      PGDATA: /var/lib/postgresql/data/pgdata
. . .
```

Next, edit the `redis` service definition to expose its default TCP port by adding a `ports:` section with the default 6379 port. Adding the `ports:`

section will make Redis available inside your Kubernetes cluster. Your edited `redis` service should resemble the following:

```
~/rails_project/docker-compose.yml
. . .
  redis:
    image: redis:5.0.7
    ports:
      - "6379:6379"
```

After editing the `redis` section of the file, continue to the `sidekiq` service definition. Just as with the `app` service, you'll need to switch from building a local docker image to pulling from Docker Hub. Make the following edits to your `sidekiq` service definition:

- Replace the `build:` line with `image: your_dockerhub_user/rails-kubernetes`
- Remove the following `context: .`, and `dockerfile: Dockerfile` lines.
- Remove the `volumes` list.

Your edited `sidekiq` definition should look like this:

```
~/rails_project/docker-compose.yml
. . .
  sidekiq:
    image: your_dockerhub_user/rails-kubernetes
    depends_on:
      - app
      - database
      - redis
    env_file: .env
    environment:
      RAILS_ENV: development
    entrypoint: ./entrypoints/sidekiq-entrypoint.sh
. . .
```

Finally, at the bottom of the file, remove the `gem_cache` and `node_modules` volumes from the top-level `volumes` key. The key will now look like this:

```
~/rails_project/docker-compose.yml
...
volumes:
  db_data:
```

Save and close the file when you are finished editing.

For reference, your completed `docker-compose.yml` file should contain the following:

~/rails_project/docker-compose.yml

```yaml
version: '3'

services:
  app:
    image: your_dockerhub_user/rails-kubernetes
    depends_on:
      - database
      - redis
    ports:
      - "3000:3000"
    env_file: .env
    environment:
      RAILS_ENV: development

  database:
    image: postgres:12.1
    volumes:
      - db_data:/var/lib/postgresql/data
    ports:
      - "5432:5432"
    environment:
      PGDATA: /var/lib/postgresql/data/pgdata

  redis:
```

```
    image: redis:5.0.7
    ports:
      - "6379:6379"

  sidekiq:
    image: your_dockerhub_user/rails-kubernetes
    depends_on:
      - app
      - database
      - redis
    env_file: .env
    environment:
      RAILS_ENV: development
    entrypoint: ./entrypoints/sidekiq-entrypoint.sh

volumes:
  db_data:
```

Before translating our service definitions, we will need to write the `.env` file that `kompose` will use to create the ConfigMap with our non-sensitive information. Please see Step 2 of Containerizing a Ruby on Rails Application for Development with Docker Compose for a longer explanation of this file.

In that tutorial, we added `.env` to our `.gitignore` file to ensure that it would not copy to version control. This means that it did not copy over when we cloned the rails-sidekiq repository in Step 2 of this tutorial. We will therefore need to recreate it now.

Create the file:

```
nano .env
```

`kompose` will use this file to create a ConfigMap for our application. However, instead of assigning all of the variables from the `app` service definition in our Compose file, we will only add settings for the PostgreSQL and Redis. We will assign the database name, username, and password separately when we manually create a Secret object in Step 4.

Add the following port and database name information to the `.env` file. Feel free to rename your database if you would like:

```
~/rails_project/.env
DATABASE_HOST=database
DATABASE_PORT=5432
REDIS_HOST=redis
REDIS_PORT=6379
```

Save and close the file when you are finished editing.

You are now ready to create the files with your object specs. `kompose` offers multiple options for translating your resources. You can: - Create `yaml` files based on the service definitions in your `docker-compose.yml` file with `komp` `ose convert`. - Create Kubernetes objects directly with `kompose up`. - Create a Helm chart with `kompose convert -c`.

For now, we will convert our service definitions to `yaml` files and then add to and revise the files that `kompose` creates.

Convert your service definitions to `yaml` files with the following command:

```
kompose convert
```

After you run this command, `kompose` will output information about the files it has created:

```
Output
INFO Kubernetes file "app-service.yaml" created
INFO Kubernetes file "database-service.yaml" created
INFO Kubernetes file "redis-service.yaml" created
INFO Kubernetes file "app-deployment.yaml" created
INFO Kubernetes file "env-configmap.yaml" created
INFO Kubernetes file "database-deployment.yaml" created
INFO Kubernetes file "db-data-persistentvolumeclaim.yaml" crea
ted
INFO Kubernetes file "redis-deployment.yaml" created
INFO Kubernetes file "sidekiq-deployment.yaml" created
```

These include `yaml` files with specs for the Rails application Service, Deployment, and ConfigMap, as well as for the `db-data`

PersistentVolumeClaim and PostgreSQL database Deployment. Also included are files for Redis and Sidekiq respectively.

To keep these manifests out of the main directory for your Rails project, create a new directory called `k8s-manifests` and then use the `mv` command to move the generated files into it:

```
mkdir k8s-manifests
mv *.yaml k8s-manifests
```

Finally, `cd` into the `k8s-manifests` directory. We'll work from inside this directory from now on to keep things tidy:

```
cd k8s-manifests
```

These files are a good starting point, but in order for our application's functionality to match the setup described in [Containerizing a Ruby on Rails Application for Development with Docker Compose]() we will need to make a few additions and changes to the files that `kompose` has generated.

Step 4 — Creating Kubernetes Secrets

In order for our application to function in the way we expect, we will need to make a few modifications to the files that `kompose` has created. The first of these changes will be generating a Secret for our database user and password and adding it to our application and database Deployments. Kubernetes offers two ways of working with environment variables: ConfigMaps and Secrets. `kompose` has already created a ConfigMap with the non-confidential information we included in our `.env` file, so we will now create a Secret with our confidential information: our database name, username and password.

The first step in manually creating a Secret will be to convert the data to [base64](), an encoding scheme that allows you to uniformly transmit data, including binary data.

First convert the database name to base64 encoded data:

```
echo -n 'your_database_name' | base64
```

Note down the encoded value.

Next convert your database username:

```
echo -n 'your_database_username' | base64
```

Again record the value you see in the output.

Finally, convert your password:

```
echo -n 'your_database_password' | base64
```

Take note of the value in the output here as well.

Open a file for the Secret:

```
nano secret.yaml
```

Note: Kubernetes objects are typically defined using YAML, which strictly forbids tabs and requires two spaces for indentation. If you would like to check the formatting of any of your `yaml` files, you can use a linter or test the validity of your syntax using `kubectl create` with the `--dry-run` and `--validate` flags:

```
kubectl create -f your_yaml_file.yaml --dry-run --validate=true
```

In general, it is a good idea to validate your syntax before creating resources with `kubectl`.

Add the following code to the file to create a Secret that will define your `DATABASE_NAME`, `DATABASE_USER` and `DATABASE_PASSWORD` using the encoded values you just created. Be sure to replace the highlighted placeholder values here with your **encoded** database name, username and password:

~/rails_project/k8s-manifests/secret.yaml

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: database-secret
data:
  DATABASE_NAME: your_database_name
  DATABASE_PASSWORD: your_encoded_password
  DATABASE_USER: your_encoded_username
```

We have named the Secret object `database-secret`, but you are free to name it anything you would like.

These secrets are used with the Rails application so that it can connect to PostgreSQL. However, the database itself needs to be initialized with these same values. So next, copy the three lines and paste them at the end of the file. Edit the last three lines and change the `DATABASE` prefix for each variable to `POSTGRES`. Finally change the `POSTGRES_NAME` variable to read `POSTGRES_DB`.

Your final `secret.yaml` file should contain the following:

~/rails_project/k8s-manifests/secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: database-secret
data:
  DATABASE_NAME: your_database_name
  DATABASE_PASSWORD: your_encoded_password
  DATABASE_USER: your_encoded_username
  POSTGRES_DB: your_database_name
  POSTGRES_PASSWORD: your_encoded_password
  POSTGRES_USER: your_encoded_username
```

Save and close this file when you are finished editing. As you did with your `.env` file, be sure to add `secret.yaml` to your `.gitignore` file to keep it out of version control.

With `secret.yaml` written, our next step will be to ensure that our application and database Deployments both use the values that we added to the file. Let's start by adding references to the Secret to our application Deployment.

Open the file called `app-deployment.yaml`:

```
nano app-deployment.yaml
```

The file's container specifications include the following environment variables defined under the `env` key:

~/rails_project/k8s-manifests/app-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
. . .
    spec:
      containers:
      - env:
        - name: DATABASE_HOST
          valueFrom:
            configMapKeyRef:
              key: DATABASE_HOST
              name: env
        - name: DATABASE_PORT
          valueFrom:
            configMapKeyRef:
              key: DATABASE_PORT
              name: env
        - name: RAILS_ENV
              value: development
        - name: REDIS_HOST
            valueFrom:
             configMapKeyRef:
              key: REDIS_HOST
              name: env
         - name: REDIS_PORT
```

```
            valueFrom:
            configMapKeyRef:
              key: REDIS_PORT
              name: env
. . .
```

We will need to add references to our Secret so that our application will have access to those values. Instead of including a `configMapKeyRef` key to point to our `env` ConfigMap, as is the case with the existing values, we'll include a `secretKeyRef` key to point to the values in our **`database-secret`** secret.

Add the following Secret references after the `- name: REDIS_PORT` variable section:

~/rails_project/k8s-manifests/app-deployment.yaml

```yaml
. . .
    spec:
      containers:
      - env:
      . . .
        - name: REDIS_PORT
          valueFrom:
          configMapKeyRef:
            key: REDIS_PORT
            name: env
        - name: DATABASE_NAME
          valueFrom:
            secretKeyRef:
              name: database-secret
              key: DATABASE_NAME
        - name: DATABASE_PASSWORD
          valueFrom:
            secretKeyRef:
              name: database-secret
              key: DATABASE_PASSWORD
        - name: DATABASE_USER
          valueFrom:
            secretKeyRef:
```

```
            name: database-secret
            key: DATABASE_USER
```

Save and close the file when you are finished editing. As with your `secrets.yaml` file, be sure to validate your edits using `kubectl` to ensure there are no issues with spaces, tabs, and indentation:

```
kubectl create -f app-deployment.yaml --dry-run --validate=true
```

```
Output
deployment.apps/app created (dry run)
```

Next, we'll add the same values to the `database-deployment.yaml` file.

Open the file for editing:

```
nano database-deployment.yaml
```

In this file, we will add references to our Secret for following variable keys: `POSTGRES_DB`, `POSTGRES_USER` and `POSTGRES_PASSWORD`. The `postgres` image makes these variables available so that you can modify the initialization of your database instance. The `POSTGRES_DB` creates a default database that is available when the container starts. The `POSTGRES_USER` and `POSTGRES_PASSWORD` together create a privileged user that can access the created database.

Using the these values means that the user we create has access to all of the administrative and operational privileges of that role in PostgreSQL. When working in production, you will want to create a dedicated application user with appropriately scoped privileges.

Under the `POSTGRES_DB`, `POSTGRES_USER` and `POSTGRES_PASSWORD` variables, add references to the Secret values:

~/rails_project/k8s-manifests/database-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
. . .
    spec:
      containers:
        - env:
            - name: PGDATA
              value: /var/lib/postgresql/data/pgdata
            - name: POSTGRES_DB
              valueFrom:
                secretKeyRef:
                  name: database-secret
                  key: POSTGRES_DB
            - name: POSTGRES_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: database-secret
                  key: POSTGRES_PASSWORD
            - name: POSTGRES_USER
              valueFrom:
                secretKeyRef:
                  name: database-secret
                  key: POSTGRES_USER
```

```
            image: postgres:12.1
. . .
```

Save and close the file when you are finished editing. Again be sure to lint your edited file using `kubectl` with the `--dry-run --validate=true` arguments.

With your Secret in place, you can move on to creating the database Service and ensuring that your application container only attempts to connect to the database once it is fully set up and initialized.

Step 5 — Modifying the PersistentVolumeClaim and Exposing the Application Frontend

Before running our application, we will make two final changes to ensure that our database storage will be provisioned properly and that we can expose our application frontend using a LoadBalancer.

First, let's modify the `storage` [`resource`](https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/) defined in the PersistentVolumeClaim that kompose created for us. This Claim allows us to [dynamically provision](https://kubernetes.io/docs/concepts/storage/dynamic-provisioning/) storage to manage our application's state.

To work with PersistentVolumeClaims, you must have a [StorageClass](https://kubernetes.io/docs/concepts/storage/storage-classes/) created and configured to provision storage resources. In our case, because we are working with [DigitalOcean Kubernetes](https://www.digitalocean.com/products/kubernetes/), our default StorageClass `provisioner` is set to `dobs.csi.digitalocean.com` — [DigitalOcean Block Storage](https://www.digitalocean.com/products/block-storage/).

We can check this by typing:

```
kubectl get storageclass
```

If you are working with a DigitalOcean cluster, you will see the following output:

```
Output
NAME                            PROVISIONER                     RECLA
IMPOLICY    VOLUMEBINDINGMODE    ALLOWVOLUMEEXPANSION    AGE
do-block-storage (default)    dobs.csi.digitalocean.com    Delet
e           Immediate            true                    76m
```

If you are not working with a DigitalOcean cluster, you will need to create a StorageClass and configure a `provisioner` of your choice. For details about how to do this, please see the official documentation.

When `kompose` created `db-data-persistentvolumeclaim.yaml`, it set the `storage` `resource` to a size that does not meet the minimum size requirements of our `provisioner`. We will therefore need to modify our PersistentVolumeClaim to use the minimum viable DigitalOcean Block Storage unit: 1GB. Please feel free to modify this to meet your storage requirements.

Open `db-data-persistentvolumeclaim.yaml`:

```
nano db-data-persistentvolumeclaim.yaml
```

Replace the `storage` value with `1Gi`:

```
~/rails_project/k8s-manifests/db-data-persistentvolumeclaim.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  creationTimestamp: null
  labels:
    io.kompose.service: db-data
  name: db-data
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
status: {}
```

Also note the `accessMode: ReadWriteOnce` means that the volume provisioned as a result of this Claim will be read-write only by a single node. Please see the documentation for more information about different access modes.

Save and close the file when you are finished.

Next, open `app-service.yaml`:

```
nano app-service.yaml
```

We are going to expose this Service externally using a DigitalOcean Load Balancer. If you are not using a DigitalOcean cluster, please consult the relevant documentation from your cloud provider for information about their load balancers. Alternatively, you can follow the official Kubernetes documentation on setting up a highly available cluster with `kubeadm`, but in this case you will not be able to use PersistentVolumeClaims to provision storage.

Within the Service spec, specify `LoadBalancer` as the Service `type`:

```
~/rails_project/k8s-manifests/app-service.yaml
apiVersion: v1
kind: Service
. . .
spec:
  type: LoadBalancer
  ports:
  . . .
```

When we create the app Service, a load balancer will be automatically created, providing us with an external IP where we can access our application. Save and close the file when you are finished editing. With all of our files in place, we are ready to start and test the application. ## Step 6 — Starting and Accessing the Application It's time to create our Kubernetes objects and test that our application is working as expected. To create the objects we've defined, we'll use `kubectl create` with the `-f` flag, which will allow us to specify the files that `kompose` created for us, along with the files we wrote. Run the following command to create the Rails application and PostgreSQL database, Redis cache, and Sidekiq Services and Deployments, along with your Secret, ConfigMap, and PersistentVolumeClaim:

```
kubectl create -f app-deployment.yaml,app-service.yaml,databas
e-deployment.yaml,database-service.yaml,db-data-persistentvolu
meclaim.yaml,env-configmap.yaml,redis-deployment.yaml,redis-se
rvice.yaml,secret.yaml,sidekiq-deployment.yaml
```

You will receive the following output, indicating that the objects have been created:

```
Output
deployment.apps/app created
service/app created
deployment.apps/database created
service/database created
persistentvolumeclaim/db-data created
configmap/env created
deployment.apps/redis created
service/redis created
secret/database-secret created
deployment.apps/sidekiq created
```

To check that your Pods are running, type:

```
kubectl get pods
```

You don't need to specify a Namespace here, since we have created our objects in the `default` Namespace. If you are working with multiple Namespaces, be sure to include the `-n` flag when running this `kubectl create` command, along with the name of your Namespace.

You will see output similar to the following while your `database` container is starting (the status will be either `Pending` or `ContainerCreating`):

```
Output
NAME                          READY   STATUS    RESTARTS   AGE
app-854d645fb9-9hv7w          1/1     Running   0          23s
database-c77d55fbb-bmfm8      0/1     Pending   0          23s
redis-7d65467b4d-9hcxk        1/1     Running   0          23s
sidekiq-867f6c9c57-mcwks      1/1     Running   0          23s
```

Once the database container is started, you will have output like this:

```
Output
NAME                          READY   STATUS    RESTARTS   AGE
app-854d645fb9-9hv7w          1/1     Running   0          30s
database-c77d55fbb-bmfm8      1/1     Running   0          30s
redis-7d65467b4d-9hcxk        1/1     Running   0          30s
sidekiq-867f6c9c57-mcwks      1/1     Running   0          30s
```

The `Running` `STATUS` indicates that your Pods are bound to nodes and that the containers associated with those Pods are running. `READY` indicates how many containers in a Pod are running. For more information, please consult the documentation on Pod lifecycles.

Note: If you see unexpected phases in the `STATUS` column, remember that you can troubleshoot your Pods with the following commands:

```
kubectl describe pods your_pod
kubectl logs your_pod
```

Now that your application is up and running, the last step that is required is to run Rails' database migrations. This step will load a schema into the PostgreSQL database for the demo application.

To run pending migrations you'll `exec` into the running application pod and then call the `rake db:migrate` command.

First, find the name of the application pod with the following command:

```
kubectl get pods
```

Find the pod that corresponds to your application like the highlighted pod name in the following output:

```
Output
NAME                        READY   STATUS    RESTARTS   AGE
app-854d645fb9-9hv7w        1/1     Running   0          30s
database-c77d55fbb-bmfm8    1/1     Running   0          30s
redis-7d65467b4d-9hcxk      1/1     Running   0          30s
sidekiq-867f6c9c57-mcwks    1/1     Running   0          30s
```

With that pod name noted down, you can now run the `kubectl exec` command to complete the database migration step.

Run the migrations with this command:

```
kubectl exec your_app_pod_name rake db:migrate
```

You should receive output similar to the following, which indicates that the database schema has been loaded:

```
Output
== 20190927142853 CreateSharks: migrating =====================================
-- create_table(:sharks)
   -> 0.0190s
== 20190927142853 CreateSharks: migrated (0.0208s) ============================

== 20190927143639 CreatePosts: migrating ======================================
-- create_table(:posts)
   -> 0.0398s
== 20190927143639 CreatePosts: migrated (0.0421s) =============================

== 20191120132043 CreateEndangereds: migrating ================================
-- create_table(:endangereds)
   -> 0.8359s
== 20191120132043 CreateEndangereds: migrated (0.8367s) =======================
```

With your containers running and data loaded, you can now access the application. To get the IP for the `app` LoadBalancer, type:

```
kubectl get svc
```

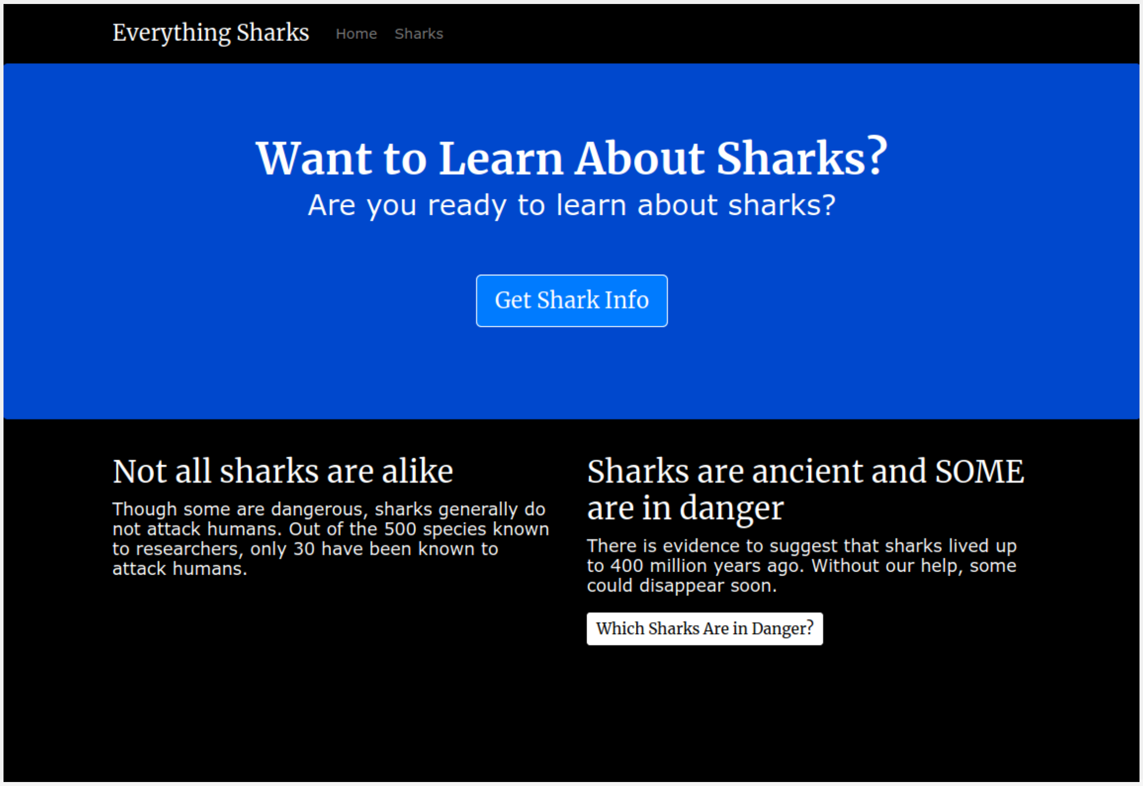
You will receive output like the following:

```
Output
NAME          TYPE           CLUSTER-IP      EXTERNAL-IP      P
ORT(S)           AGE
app           LoadBalancer   10.245.73.142   your_lb_ip    3000:
31186/TCP   21m
database      ClusterIP      10.245.155.87   <none>           5
432/TCP          21m
kubernetes    ClusterIP      10.245.0.1      <none>           4
43/TCP           21m
redis         ClusterIP      10.245.119.67   <none>           6
379/TCP          21m
```

The `EXTERNAL_IP` associated with the `app` service is the IP address where you can access the application. If you see a `<pending>` status in the `EXTERNAL_IP` column, this means that your load balancer is still being created.

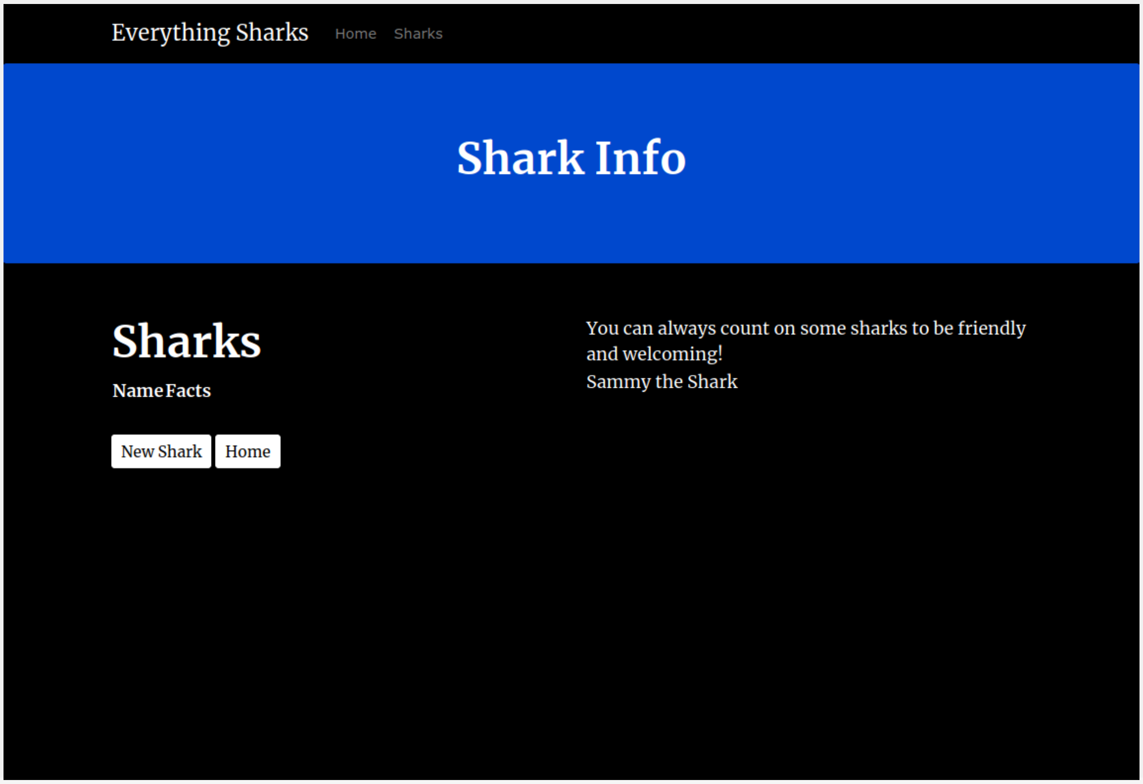
Once you see an IP in that column, navigate to it in your browser: `http://your_lb_ip:3000`.

You should see the following landing page:



Application Landing Page

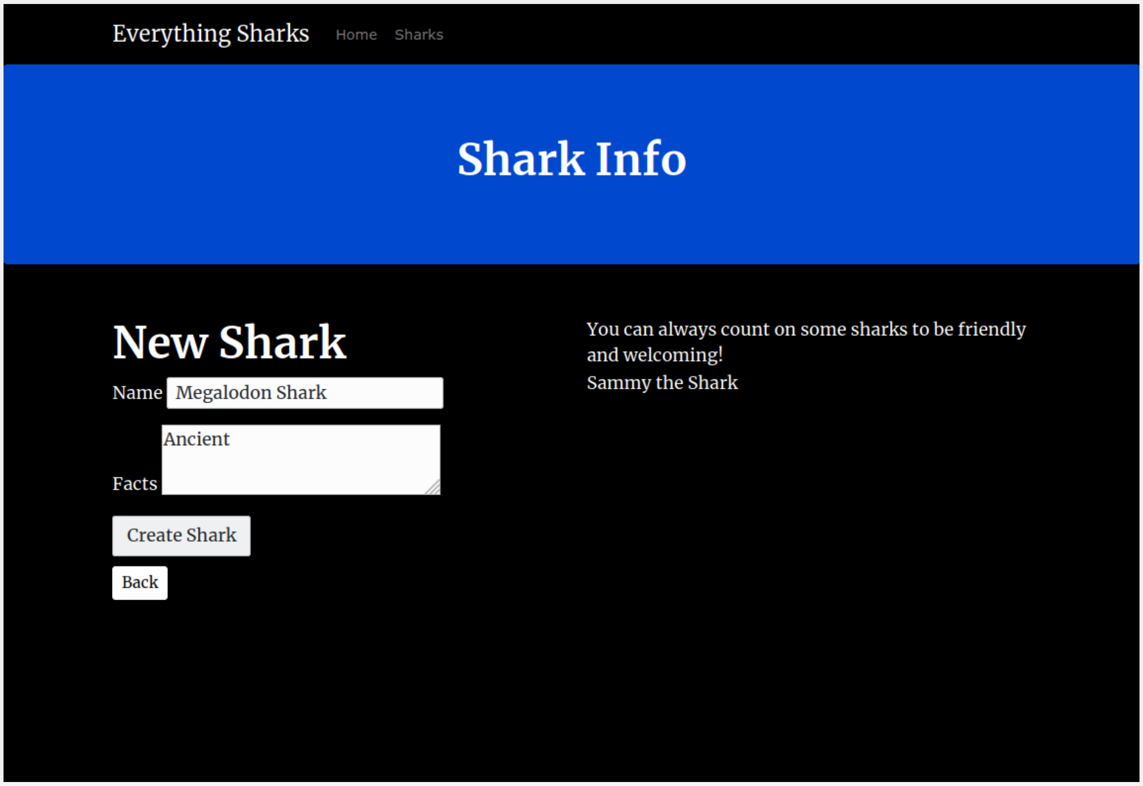
Click on the **Get Shark Info** button. You will have a page with a button to create a new shark:



Shark Info Form

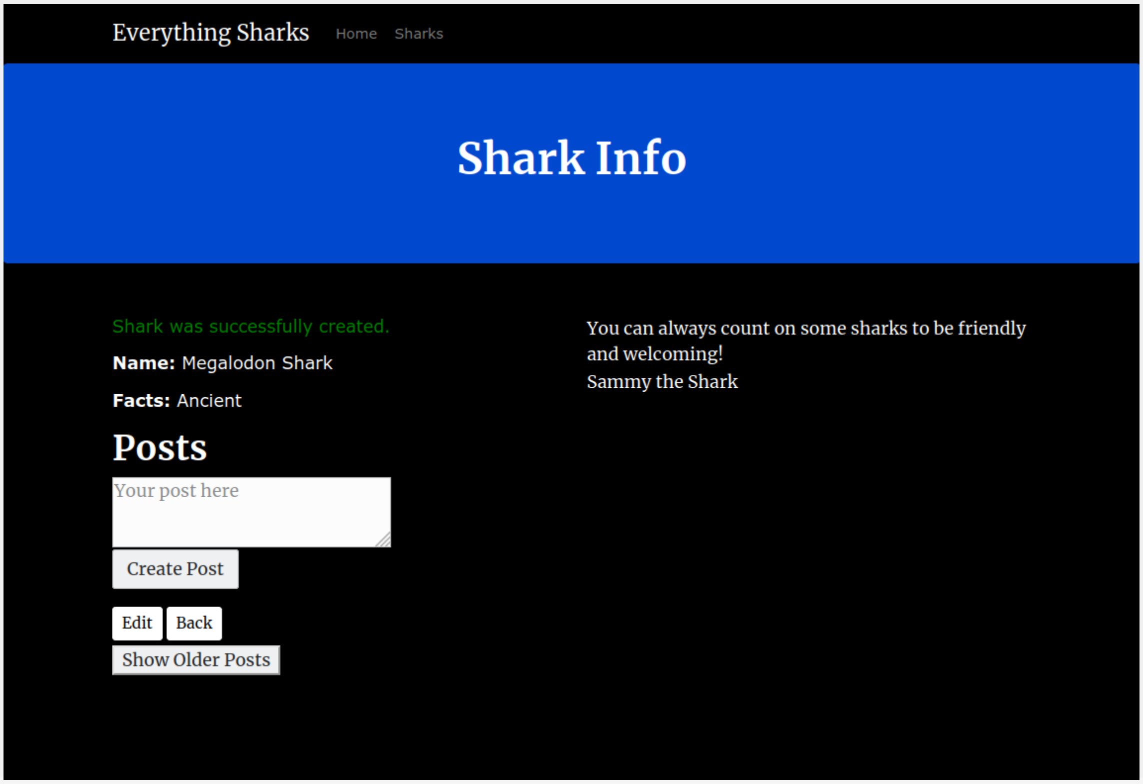
Click it and when prompted, enter the username and password from earlier in the tutorial series. If you did not change these values then the defaults are `sammy` and `shark` respectively.

In the form, add a shark of your choosing. To demonstrate, we will add `Megalodon Shark` to the **Shark Name** field, and `Ancient` to the **Shark Character** field:



Filled Shark Form

Click on the **Submit** button. You will see a page with this shark information displayed back to you:



Shark Output

You now have a single instance setup of a Rails application with a PostgreSQL database running on a Kubernetes cluster. You also have a Redis cache and a Sidekiq worker to process data that users submit.

Conclusion

The files you have created in this tutorial are a good starting point to build from as you move toward production. As you develop your application, you can work on implementing the following: - **Centralized logging and monitoring**. Please see the relevant discussion in Modernizing Applications for Kubernetes for a general overview. You can also look at How To Set Up an Elasticsearch, Fluentd and Kibana (EFK) Logging Stack on Kubernetes to learn how to set up a logging stack with Elasticsearch, Fluentd, and Kibana. Also check out An Introduction to Service Meshes for information about how service meshes like Istio implement this functionality. - **Ingress Resources to route traffic to your cluster**. This is a good alternative to a LoadBalancer in cases where you are running multiple Services, which each require their own LoadBalancer, or where you would like to implement application-level routing strategies (A/B & canary tests, for example). For more information, check out How to Set Up an Nginx Ingress with Cert-Manager on DigitalOcean Kubernetes and the related discussion of routing in the service mesh context in An Introduction to Service Meshes. - **Backup strategies for your Kubernetes objects**. For guidance on implementing backups with Velero with DigitalOcean's Kubernetes product, please see How To Back Up and Restore a Kubernetes Cluster on DigitalOcean Using Velero.