CRUD APPLICATION ON RUBY

SUBMITTED BY **SUMA P HIREMATH**

Creating a New Rails Project

The best way to read this guide is to follow it step by step. All steps are essential to run this example application and no additional code or steps are needed.

By following along with this guide, you'll create a Rails project called `blog`, a (very) simple weblog. Before you can start building the application, you need to make sure that you have Rails itself installed.

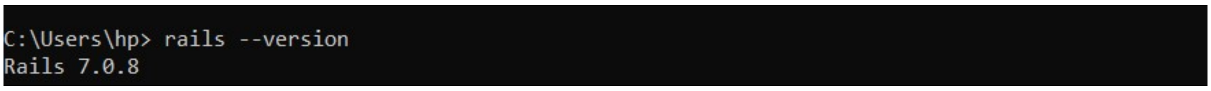
3.1 Installing Rails

Before you install Rails, you should check to make sure that your system has the proper prerequisites installed. These include:

- Ruby
- SQLite3

3.1.1 Installing Ruby

Open up a command line prompt. On macOS open Terminal.app; on Windows choose "Run" from your Start menu and type `cmd.exe`. Any commands prefaced with a dollar sign `$` should be run in the command line. Verify that you have a current version of Ruby installed:



```
C:\Users\hp> rails --version
Rails 7.0.8
```

Rails requires Ruby version 2.7.0 or later. It is preferred to use the latest Ruby version. If the version number returned is less than that number (such as 2.3.7, or 1.8.7), you'll need to install a fresh copy of Ruby.

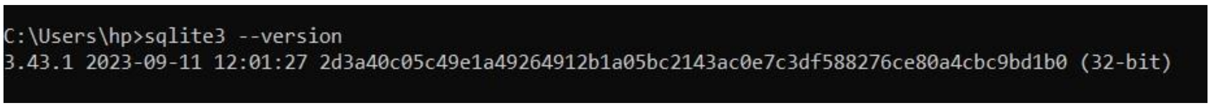
To install Rails on Windows, you'll first need to install Ruby Installer.

For more installation methods for most Operating Systems take a look at ruby-lang.org.

Installing SQLite3

You will also need an installation of the SQLite3 database. Many popular UNIX-like OSes ship with an acceptable version of SQLite3. Others can find installation instructions at the SQLite3 website.

Verify that it is correctly installed and in your load PATH:

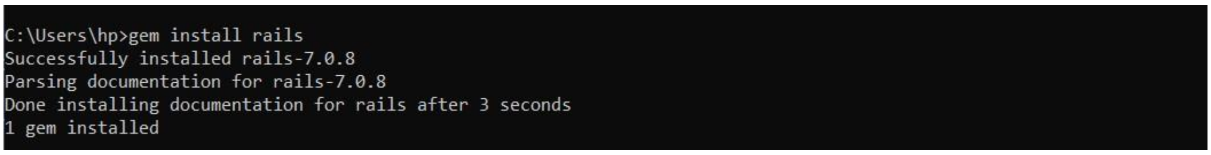


```
C:\Users\hp>sqlite3 --version
3.43.1 2023-09-11 12:01:27 2d3a40c05c49e1a49264912b1a05bc2143ac0e7c3df588276ce80a4cbc9bd1b0 (32-bit)
```

The program should report its version.

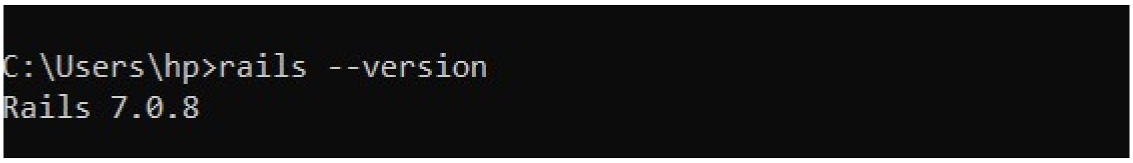
Installing Rails

To install Rails, use the gem install command provided by RubyGems:



```
C:\Users\hp>gem install rails
Successfully installed rails-7.0.8
Parsing documentation for rails-7.0.8
Done installing documentation for rails after 3 seconds
1 gem installed
```

To verify that you have everything installed correctly, you should be able to run the following in a new terminal:



```
C:\Users\hp>rails --version
Rails 7.0.8
```

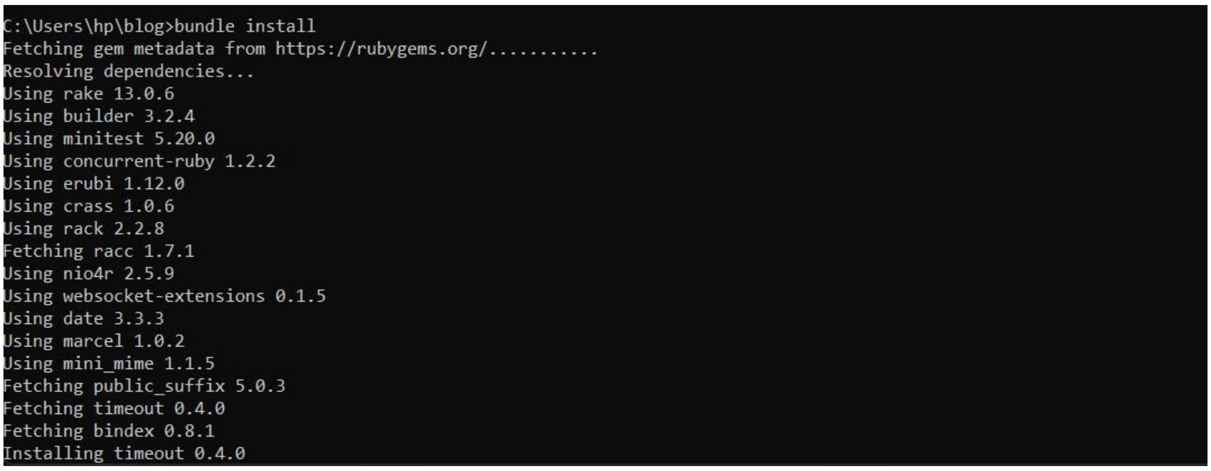
3.2 Creating the Blog Application

Rails comes with a number of scripts called generators that are designed to make your development life easier by creating everything that's necessary to start working on a particular task. One of these is the new application generator, which will provide you with the foundation of a fresh Rails application so that you don't have to write it yourself.

To use this generator, open a terminal, navigate to a directory where you have rights to create files, and run:

```
C:\User\hp>rails new blog
```

This will create a Rails application called Blog in a blog directory and install the gem dependencies that are already mentioned in Gemfile using bundle install.



```
C:\Users\hp\blog>bundle install
Fetching gem metadata from https://rubygems.org/..........
Resolving dependencies...
Using rake 13.0.6
Using builder 3.2.4
Using minitest 5.20.0
Using concurrent-ruby 1.2.2
Using erubi 1.12.0
Using crass 1.0.6
Using rack 2.2.8
Fetching racc 1.7.1
Using nio4r 2.5.9
Using websocket-extensions 0.1.5
Using date 3.3.3
Using marcel 1.0.2
Using mini_mime 1.1.5
Fetching public_suffix 5.0.3
Fetching timeout 0.4.0
Fetching bindex 0.8.1
Installing timeout 0.4.0
```

After you create the blog application, switch to its folder:

```
C:\User\hp> cd blog
```

The blog directory will have a number of generated files and folders that make up the structure of a Rails application. Most of the work in this tutorial will happen in the app folder, but here's a basic rundown on the function of each of the files and folders that Rails creates by default:

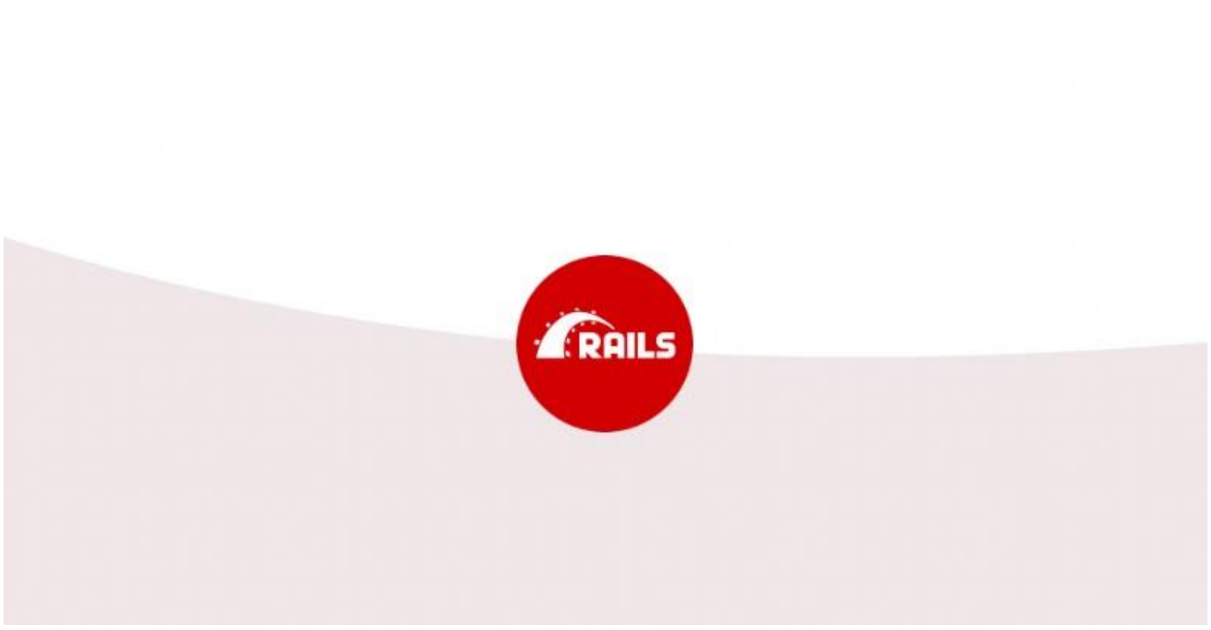
Hello, Rails!

To begin with, let's get some text up on screen quickly. To do this, you need to get your Rails application server running.

Starting Up the Web Server

You actually have a functional Rails application already. To see it, you need to start a web server on your development machine. You can do this by running the following command in the blog directory:

This will start up Puma, a web server distributed with Rails by default. To see your application in action, open a browser window and navigate to http://localhost:3000. You should see the Rails default information page:



When you want to stop the web server, hit Ctrl+C in the terminal window where it's running. In the development environment, Rails does not generally require you to restart the server; changes you make in files will be automatically picked up by the server.

The Rails startup page is the *smoke test* for a new Rails application: it makes sure that you have your software configured correctly enough to serve a page.

Say "Hello", Rails

To get Rails saying "Hello", you need to create at minimum a *route*, a *controller* with an *action*, and a *view*. A route maps a request to a controller action. A controller action performs the necessary work to handle the request, and prepares any data for the view. A view displays data in a desired format.

In terms of implementation: Routes are rules written in a Ruby DSL (Domain-Specific Language). Controllers are Ruby classes, and their public methods are actions. And views are templates, usually written in a mixture of HTML and Ruby.

Let's start by adding a route to our routes file, config/routes.rb, at the top of the Rails.application.routes.draw block:

```
C:\User\hp> Rails.application.routes.draw do

  get "/articles", to: "articles#index"

  # For details on the DSL available within this file, see
https://guides.rubyonrails.org/routing.html

end
```

The route above declares that GET /articles requests are mapped to the index action of ArticlesController.

To create ArticlesController and its index action, we'll run the controller generator (with the --skip-routes option because we already have an appropriate route):

```
C:\User\hp>bin/rails generate controller Articles index --skip-route
```

Rails will create several files for you:

```
create  app/controllers/articles_controller.rb

invoke  erb

create    app/views/articles

create    app/views/articles/index.html.erb

invoke  test_unit

create    test/controllers/articles_controller_test.rb

invoke  helper

create    app/helpers/articles_helper.rb

invoke    test_unit
```

The most important of these is the controller file, app/controllers/articles_controller.rb. Let's take a look at it:

```
class ArticlesController < ApplicationController
  def index
  end
end
```

The index action is empty. When an action does not explicitly render a view (or otherwise trigger an HTTP response), Rails will automatically render a view that matches the name of the controller and action. Convention Over Configuration! Views are located in the app/views directory. So the index action will render app/views/articles/index.html.erb by default.

Let's open app/views/articles/index.html.erb, and replace its contents with:

```
<h1>Hello, Rails!</h1>
```

If you previously stopped the web server to run the controller generator, restart it with bin/rails server. Now visit http://localhost:3000/articles, and see our text displayed!

4.3 Setting the Application Home Page

At the moment, http://localhost:3000 still displays a page with the Ruby on Rails logo. Let's display our "Hello, Rails!" text at http://localhost:3000 as well. To do so, we will add a route that maps the *root path* of our application to the appropriate controller and action.

Let's open config/routes.rb, and add the following root route to the top of the Rails.application.routes.draw block:

```
Rails.application.routes.draw do
  root "articles#index"

  get "/articles", to: "articles#index"
end
```

Now we can see our "Hello, Rails!" text when we visit http://localhost:3000, confirming that the root route is also mapped to the index action of ArticlesController.

Autoloading

Rails applications **do not** use require to load application code.

You may have noticed that ArticlesController inherits from ApplicationController, but app/controllers/articles_controller.rb does not have anything like

You only need require calls for two use cases:

- To load files under the lib directory.
- To load gem dependencies that have require: false in the Gemfile.

6 MVC and You

So far, we've discussed routes, controllers, actions, and views. All of these are typical pieces of a web application that follows the MVC (Model-View-Controller) pattern. MVC is a design pattern that divides the responsibilities of an application to make it easier to reason about. Rails follows this design pattern by convention.

Since we have a controller and a view to work with, let's generate the next piece: a model.

6.1 Generating a Model

A model is a Ruby class that is used to represent data. Additionally, models can interact with the application's database through a feature of Rails called Active Record.

To define a model, we will use the model generator:

```
C:\User\hp>bin/rails generate model Article title:string body:text
```

This will create several files:

```
invoke  active_record

create    db/migrate/<timestamp>_create_articles.rb

create    app/models/article.rb

invoke    test_unit

create      test/models/article_test.rb

create      test/fixtures/articles.yml
```

The two files we'll focus on are the migration file (db/migrate/<timestamp>_create_articles.rb) and the model file (app/models/article.rb).

<u>6.2 Database Migrations</u>

Migrations are used to alter the structure of an application's database. In Rails applications, migrations are written in Ruby so that they can be database-agnostic.

Let's take a look at the contents of our new migration file:

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

  def change

    create_table :articles do |t|

      t.string :title

      t.text :body


      t.timestamps

    end

  end

end
```

The call to create_table specifies how the articles table should be constructed. By default, the create_table method adds an id column as an auto-incrementing primary key. So the first record in the table will have an id of 1, the next record will have an id of 2, and so on.

Inside the block for create_table, two columns are defined: title and body. These were added by the generator because we included them in our generate command (bin/rails generate model Article title:string body:text).

On the last line of the block is a call to t.timestamps. This method defines two additional columns named created_at and updated_at. As we will see, Rails will manage these for us, setting the values when we create or update a model object.

Let's run our migration with the following command:

```
C:\User\hp>bin/rails db:migrate
```

The command will display output indicating that the table was created:

```
==  CreateArticles: migrating ===================================

-- create_table(:articles)

   -> 0.0018s

==  CreateArticles: migrated (0.0018s) ==========================
```

Using a Model to Interact with the Database

To play with our model a bit, we're going to use a feature of Rails called the *console*. The console is an interactive coding environment just like irb, but it also automatically loads Rails and our application code.

Let's launch the console with this command:

```
C:\User\hp>bin/rails console
```

You should see an irb prompt like:

```
Loading development environment (Rails 7.0.0)

irb(main):001:0>
```

At this prompt, we can initialize a new Article object:

```
irb> article = Article.new(title: "Hello Rails", body: "I am on Rails!")
```

It's important to note that we have only *initialized* this object. This object is not saved to the database at all. It's only available in the console at the moment. To save the object to the database, we must call save:

```
irb> article.save

(0.1ms)  begin transaction
```

```
Article Create (0.4ms)  INSERT INTO "articles" ("title", "body",
"created_at", "updated_at") VALUES (?, ?, ?, ?)  [["title", "Hello Rails"],
["body", "I am on Rails!"], ["created_at", "2020-01-18 23:47:30.734416"],
["updated_at", "2020-01-18 23:47:30.734416"]]

(0.9ms)  commit transaction

=> true
```

The above output shows an INSERT INTO "articles" ... database query. This indicates that the article has been inserted into our table. And if we take a look at the article object again, we see something interesting has happened:

```
irb> article
=> #<Article id: 1, title: "Hello Rails", body: "I am on Rails!", created_at: "2020-
01-18 23:47:30", updated_at: "2020-01-18 23:47:30">
```

The id, created_at, and updated_at attributes of the object are now set. Rails did this for us when we saved the object.

When we want to fetch this article from the database, we can call find on the model and pass the id as an argument

```
irb> Article.all

=> #<ActiveRecord::Relation [#<Article id: 1, title: "Hello
Rails", body: "I am on Rails!", created_at: "2020-01-18
23:47:30", updated_at: "2020-01-18 23:47:30">]>
```

This method returns an ActiveRecord::Relation object, which you can think of as a super-powered array.

Models are the final piece of the MVC puzzle. Next, we will connect all of the pieces together.

Showing a List of Articles

Let's go back to our controller in app/controllers/articles_controller.rb, and change the index action to fetch all articles from the database:

```
class ArticlesController < ApplicationController

  def index

    @articles = Article.all

  end

end
```

Controller instance variables can be accessed by the view. That means we can
reference @articles in app/views/articles/index.html.erb. Let's open that file, and replace its contents with:

```
<h1>Articles</h1>

<ul>

  <% @articles.each do |article| %>

    <li>

      <%= article.title %>

    </li>

  <% end %>

</ul>
```

The above code is a mixture of HTML and *ERB*. ERB is a templating system that evaluates Ruby code embedded in a document. Here, we can see two types of ERB tags: <% %> and <%= %>. The <% %> tag means "evaluate the enclosed Ruby code." The <%= %> tag means "evaluate the enclosed Ruby code, and output the value it returns." Anything you could write in a regular Ruby program can go inside these ERB tags, though it's usually best to keep the contents of ERB tags short, for readability.

Since we don't want to output the value returned by @articles.each, we've enclosed that code in <% %>. But, since we *do* want to output the value returned by article.title (for each article), we've enclosed that code in <%= %>.

We can see the final result by visiting http://localhost:3000. (Remember that bin/rails server must be running!) Here's what happens when we do that:

1. The browser makes a request: GET http://localhost:3000.
2. Our Rails application receives this request.
3. The Rails router maps the root route to the index action of ArticlesController.
4. The index action uses the Article model to fetch all articles in the database.
5. Rails automatically renders the app/views/articles/index.html.erb view.

6. The ERB code in the view is evaluated to output HTML.
7. The server sends a response containing the HTML back to the browser.

We've connected all the MVC pieces together, and we have our first controller action! Next, we'll move on to the second action.

CRUDit Where CRUDit Is Due

Almost all web applications involve CRUD (Create, Read, Update, and Delete) operations. You may even find that the majority of the work your application does is CRUD. Rails acknowledges this, and provides many features to help simplify code doing CRUD.

Let's begin exploring these features by adding more functionality to our application.

Showing a Single Article

We currently have a view that lists all articles in our database. Let's add a new view that shows the title and body of a single article.

We start by adding a new route that maps to a new controller action (which we will add next). Open config/routes.rb, and insert the last route shown here:

```
Rails.application.routes.draw do

  root "articles#index"

  get "/articles", to: "articles#index"

  get "/articles/:id", to: "articles#show"

end
```

The new route is another get route, but it has something extra in its path: :id. This designates a route *parameter*. A route parameter captures a segment of the request's path, and puts that value into the params Hash, which is accessible by the controller action. For example, when handling a request like GET http://localhost:3000/articles/1, 1 would be captured as the value for :id, which would then be accessible as params[:id] in the show action of ArticlesController.

Let's add that show action now, below the index action in app/controllers/articles_controller.rb:

```
class ArticlesController < ApplicationController

  def index

    @articles = Article.all

  end

  def show

    @article = Article.find(params[:id])

  end

end
```

The show action calls Article.find (mentioned previously) with the ID captured by the route parameter. The returned article is stored in the @article instance variable, so it is accessible by the view. By default, the show action will render app/views/articles/show.html.erb.

Let's create app/views/articles/show.html.erb, with the following contents:

```
<h1><%= @article.title %></h1>


<p><%= @article.body %></p>
```

Now we can see the article when we visit http://localhost:3000/articles/1!

```
<h1>Articles</h1>

<ul>

  <% @articles.each do |article| %>

    <li>

      <a href="/articles/<%= article.id %>">

        <%= article.title %>

      </a>

    </li>

  <% end %>

</ul>
```

Resourceful Routing

So far, we've covered the "R" (Read) of CRUD. We will eventually cover the "C" (Create), "U" (Update), and "D" (Delete). As you might have guessed, we will do so by adding new routes, controller actions, and views. Whenever we have such a combination of routes, controller actions, and views that work together to perform CRUD operations on an entity, we call that entity a *resource*. For example, in our application, we would say an article is a resource.

Rails provides a routes method named resources that maps all of the conventional routes for a collection of resources, such as articles. So before we proceed to the "C", "U", and "D" sections, let's replace the two get routes in config/routes.rb with resources:

```
Rails.application.routes.draw do
  root "articles#index"


  resources :articles
end
```

We can inspect what routes are mapped by running the bin/rails routes command:

```

bin/rails routes
      Prefix Verb   URI Pattern
Controller#Action
        root GET    /                            articles#index
    articles GET    /articles(.:format)          articles#index
 new_article GET    /articles/new(.:format)      articles#new
     article GET    /articles/:id(.:format)      articles#show
             POST   /articles(.:format)          articles#create
edit_article GET    /articles/:id/edit(.:format) articles#edit
             PATCH  /articles/:id(.:format)      articles#update
             DELETE /articles/:id(.:format)
articles#destroy
```

The resources method also sets up URL and path helper methods that we can use to keep our code from depending on a specific route configuration. The values in the "Prefix" column above plus a suffix of _url or _path form the names of these helpers. For example, the article_path helper returns "/articles/#{article.id}" when given an article. We can use it to tidy up our links in app/views/articles/index.html.erb:

```
<h1>Articles</h1>


<ul>

  <% @articles.each do |article| %>

    <li>

      <a href="<%= article_path(article) %>">

        <%= article.title %>

      </a>

    </li>

  <% end %>

</ul>
```

However, we will take this one step further by using the link_to helper. The link_to helper renders a link with its first argument as the link's text and its second argument as the link's destination. If we pass a model object as the second argument, link_to will call the appropriate path helper to convert the object to a path. For example, if we pass an article, link_to will call article_path.
So app/views/articles/index.html.erb becomes:

```
<h1>Articles</h1>

<ul>

  <% @articles.each do |article| %>

    <li>

      <%= link_to article.title, article %>

    </li>

  <% end %>

</ul>
```

Creating a New Article

Now we move on to the "C" (Create) of CRUD. Typically, in web applications, creating a new resource is a multi-step process. First, the user requests a form to fill out. Then, the user submits the form. If there are no errors, then the resource is created and some kind of confirmation is displayed. Else, the form is redisplayed with error messages, and the process is repeated.

In a Rails application, these steps are conventionally handled by a controller's new and create actions. Let's add a typical implementation of these actions to app/controllers/articles_controller.rb, below the show action:

```ruby
class ArticlesController < ApplicationController
  def index
    @articles = Article.all
  end

  def show
    @article = Article.find(params[:id])
  end

  def new
    @article = Article.new
  end

  def create
    @article = Article.new(title: "...", body: "...")

    if @article.save
      redirect_to @article
    else
      render :new, status: :unprocessable_entity
    end
  end
end
```

The new action instantiates a new article, but does not save it. This article will be used in the view when building the form. By default, the new action will render app/views/articles/new.html.erb, which we will create next.

The create action instantiates a new article with values for the title and body, and attempts to save it. If the article is saved successfully, the action redirects the browser to the article's page at "http://localhost:3000/articles/#{@article.id}". Else, the action redisplays the form by rendering app/views/articles/new.html.erb with status code 422 Unprocessable Entity. The title and body here are dummy values. After we create the form, we will come back and change these.

Using a Form Builder

We will use a feature of Rails called a *form builder* to create our form. Using a form builder, we can write a minimal amount of code to output a form that is fully configured and follows Rails conventions.

Let's create app/views/articles/new.html.erb with the following contents:

<h1>New Article</h1>

```
<%= form_with model: @article do |form| %>

  <div>

    <%= form.label :title %><br>

    <%= form.text_field :title %>

  </div>

  <div>

    <%= form.label :body %><br>

    <%= form.text_area :body %>

  </div>

  <div>

    <%= form.submit %>

  </div>

<% end %>
```

The form_with helper method instantiates a form builder. In the form_with block we call methods like label and text_field on the form builder to output the appropriate form elements.

The resulting output from our form_with call will look like

```
<form action="/articles" accept-charset="UTF-8" method="post">
  <input type="hidden" name="authenticity_token" value="...">

  <div>
    <label for="article_title">Title</label><br>
    <input type="text" name="article[title]" id="article_title">
  </div>

  <div>
    <label for="article_body">Body</label><br>
    <textarea name="article[body]" id="article_body"></textarea>
  </div>

  <div>
    <input type="submit" name="commit" value="Create Article" data-disable-with="Create Article">
  </div>
</form>
```

Using Strong Parameters

Submitted form data is put into the params Hash, alongside captured route parameters. Thus, the create action can access the submitted title via params[:article][:title] and the submitted body via params[:article][:body]. We could pass these values individually to Article.new, but that would be verbose and possibly error-prone. And it would become worse as we add more fields.

Instead, we will pass a single Hash that contains the values. However, we must still specify what values are allowed in that Hash. Otherwise, a malicious user could potentially submit extra form fields and overwrite private data. In fact, if we pass the unfiltered params[:article] Hash directly to Article.new, Rails will raise a ForbiddenAttributesError to alert us about the problem. So we will use a feature of Rails called *Strong Parameters* to filter params. Think of it as strong typing for params.

Let's add a private method to the bottom of app/controllers/articles_controller.rb named article_params that filters params. And let's change create to use it:

```ruby
class ArticlesController < ApplicationController
  def index
    @articles = Article.all
  end

  def show
    @article = Article.find(params[:id])
  end

  def new
    @article = Article.new
  end

  def create
    @article = Article.new(article_params)

    if @article.save
      redirect_to @article
    else
      render :new, status: :unprocessable_entity
    end
  end

  private
    def article_params
      params.require(:article).permit(:title, :body)
    end
end
```

Validations and Displaying Error Messages

As we have seen, creating a resource is a multi-step process. Handling invalid user input is another step of that process. Rails provides a feature called *validations* to help us deal with invalid user input. Validations are rules that are checked before a model object is saved. If any of the checks fail, the save will be aborted, and appropriate error messages will be added to the errors attribute of the model object.

Let's add some validations to our model in app/models/article.rb:

```ruby
class Article < ApplicationRecord

  validates :title, presence: true

  validates :body, presence: true, length: { minimum: 10 }

end
```

The first validation declares that a title value must be present. Because title is a string, this means that the title value must contain at least one non-whitespace character.

The second validation declares that a body value must also be present. Additionally, it declares that the body value must be at least 10 characters long.

With our validations in place, let's modify app/views/articles/new.html.erb to display any error messages for title and body:

```erb
<h1>New Article</h1>


<%= form_with model: @article do |form| %>
  <div>
    <%= form.label :title %><br>
    <%= form.text_field :title %>
    <% @article.errors.full_messages_for(:title).each do |message| %>
      <div><%= message %></div>
    <% end %>
  </div>


  <div>
    <%= form.label :body %><br>
    <%= form.text_area :body %><br>
    <% @article.errors.full_messages_for(:body).each do |message| %>
      <div><%= message %></div>
    <% end %>
  </div>


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

The full_messages_for method returns an array of user-friendly error messages for a specified attribute. If there are no errors for that attribute, the array will be empty.

To understand how all of this works together, let's take another look at the new and create controller actions:

```
def new
    @article = Article.new
  end


  def create
    @article = Article.new(article_params)


    if @article.save
      redirect_to @article
    else
      render :new, status: :unprocessable_entity
    end
  end
```

When we visit http://localhost:3000/articles/new, the GET /articles/new request is mapped to the new action. The new action does not attempt to save @article. Therefore, validations are not checked, and there will be no error messages.

When we submit the form, the POST /articles request is mapped to the create action. The create action *does* attempt to save @article. Therefore, validations *are* checked. If any validation fails, @article will not be saved, and app/views/articles/new.html.erb will be rendered with error messages.

Finishing Up

We can now create an article by visiting http://localhost:3000/articles/new. To finish up, let's link to that page from the bottom of app/views/articles/index.html.erb:

```
<h1>Articles</h1>
<ul>
  <% @articles.each do |article| %>
    <li>
      <%= link_to article.title, article %>
    </li>
  <% end %>
</ul>


<%= link_to "New Article", new_article_path %>
```

Updating an Article

We've covered the "CR" of CRUD. Now let's move on to the "U" (Update). Updating a resource is very similar to creating a resource. They are both multi-step processes. First, the user requests a form to edit the data. Then, the user submits the form. If there are no errors, then the resource is updated. Else, the form is redisplayed with error messages, and the process is repeated.

These steps are conventionally handled by a controller's edit and update actions. Let's add a typical implementation of these actions to app/controllers/articles_controller.rb, below the create action:

```
class ArticlesController < ApplicationController
  def index
    @articles = Article.all
  end

  def show
    @article = Article.find(params[:id])
  end

  def new
    @article = Article.new
  end

  def create
    @article = Article.new(article_params)

    if @article.save
      redirect_to @article
    else
      render :new, status: :unprocessable_entity
    end
  end

  def edit
```

```
    @article = Article.find(params[:id])
  end

  def update
    @article = Article.find(params[:id])

    if @article.update(article_params)
      redirect_to @article
    else
      render :edit, status: :unprocessable_entity
    end
  end

  private
    def article_params
      params.require(:article).permit(:title, :body)
    end
end
```

Notice how the edit and update actions resemble the new and create actions.

The edit action fetches the article from the database, and stores it in @article so that it can be used when building the form. By default, the edit action will render app/views/articles/edit.html.erb.

The update action (re-)fetches the article from the database, and attempts to update it with the submitted form data filtered by article_params. If no validations fail and the update is successful, the action redirects the browser to the article's page. Else, the action redisplays the form — with error messages — by rendering app/views/articles/edit.html.erb.

Using Partials to Share View Code

Our edit form will look the same as our new form. Even the code will be the same, thanks to the Rails form builder and resourceful routing. The form builder automatically configures the form to make the appropriate kind of request, based on whether the model object has been previously saved.

Because the code will be the same, we're going to factor it out into a shared view called a *partial*. Let's create app/views/articles/_form.html.erb with the following contents:

```erb
<%= form_with model: article do |form| %>
  <div>
    <%= form.label :title %><br>
    <%= form.text_field :title %>
    <% article.errors.full_messages_for(:title).each do |message| %>
      <div><%= message %></div>
    <% end %>
  </div>

  <div>
    <%= form.label :body %><br>
    <%= form.text_area :body %><br>
    <% article.errors.full_messages_for(:body).each do |message| %>
      <div><%= message %></div>
    <% end %>
  </div>

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

The above code is the same as our form in app/views/articles/new.html.erb, except that all occurrences of @article have been replaced with article. Because partials are shared code, it is best practice that they do not depend on specific instance variables set by a controller action. Instead, we will pass the article to the partial as a local variable.

Let's update app/views/articles/new.html.erb to use the partial via render:

```erb
<h1>New Article</h1>

<%= render "form", article: @article %>
```

Finishing Up

We can now update an article by visiting its edit page, http://localhost:3000/articles/1/edit. To finish up, let's link to the edit page from the bottom of app/views/articles/show.html.erb:

```
<h1><%= @article.title %></h1>

<p><%= @article.body %></p>

<ul>

  <li><%= link_to "Edit", edit_article_path(@article) %></li>

</ul>
```

Deleting an Article

Finally, we arrive at the "D" (Delete) of CRUD. Deleting a resource is a simpler process than creating or updating. It only requires a route and a controller action. And our resourceful routing (resources :articles) already provides the route, which maps DELETE /articles/:id requests to the destroy action of ArticlesController.

So, let's add a typical destroy action to app/controllers/articles_controller.rb, below the update action:

```
class ArticlesController < ApplicationController

  def index

    @articles = Article.all

  end


  def show

    @article = Article.find(params[:id])

  end


  def new

    @article = Article.new

  end


  def create

    @article = Article.new(article_params)


    if @article.save

      redirect_to @article
```

```ruby
    else
      render :new, status: :unprocessable_entity
    end
  end

  def edit
    @article = Article.find(params[:id])
  end

  def update
    @article = Article.find(params[:id])

    if @article.update(article_params)
      redirect_to @article
    else
      render :edit, status: :unprocessable_entity
    end
  end

  def destroy
    @article = Article.find(params[:id])
    @article.destroy

    redirect_to root_path, status: :see_other
  end

  private
    def article_params
      params.require(:article).permit(:title, :body)
    end
end
```

The destroy action fetches the article from the database, and calls destroy on it. Then, it redirects the browser to the root path with status code 303 See Other.

We have chosen to redirect to the root path because that is our main access point for articles. But, in other circumstances, you might choose to redirect to articles_path.

Now let's add a link at the bottom of app/views/articles/show.html.erb so that we can delete an article from its own page:

```
<h1><%= @article.title %></h1>

<p><%= @article.body %></p>


<ul>

  <li><%= link_to "Edit", edit_article_path(@article) %></li>

  <li><%= link_to "Destroy", article_path(@article), data: {

                    turbo_method: :delete,

                    turbo_confirm: "Are you sure?"

                  } %></li>

</ul>
```

In the above code, we use the data option to set the data-turbo-method and data-turbo-confirm HTML attributes of the "Destroy" link. Both of these attributes hook into Turbo, which is included by default in fresh Rails applications. data-turbo-method="delete" will cause the link to make a DELETE request instead of a GET request. data-turbo-confirm="Are you sure?" will cause a confirmation dialog to appear when the link is clicked. If the user cancels the dialog, the request will be aborted.

And that's it! We can now list, show, create, update, and delete articles! InCRUDable!

Adding a Second Model

It's time to add a second model to the application. The second model will handle comments on articles.

Generating a Model

We're going to see the same generator that we used before when creating the Article model. This time we'll create a Comment model to hold a reference to an article. Run this command in your terminal:

```
bin/rails generate model Comment commenter:string body:text
article:references
```

This command will generate four files:

File	Purpose
db/migrate/20140120201010_create_comments.rb	Migration to create the comments table in your database (your name will include a different timestamp)
app/models/comment.rb	The Comment model
test/models/comment_test.rb	Testing harness for the comment model
test/fixtures/comments.yml	Sample comments for use in testing

First, take a look at app/models/comment.rb:

```
class Comment < ApplicationRecord
  belongs_to :article
end
```

This is very similar to the Article model that you saw earlier. The difference is the line belongs_to :article, which sets up an Active Record *association*. You'll learn a little about associations in the next section of this guide.

The (:references) keyword used in the shell command is a special data type for models. It creates a new column on your database table with the provided model name appended with an _id that can hold integer values. To get a better understanding, analyze the db/schema.rb file after running the migration.

In addition to the model, Rails has also made a migration to create the corresponding database table:

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

  def change

    create_table :comments do |t|

      t.string :commenter

      t.text :body

      t.references :article, null: false, foreign_key: true

      t.timestamps

    end

  end

end
```

The t.references line creates an integer column called article_id, an index for it, and a foreign key constraint that points to the id column of the articles table. Go ahead and run the migration:

```
bin/rails db:migrate
```

Rails is smart enough to only execute the migrations that have not already been run against the current database, so in this case you will just see:

```
==  CreateComments: migrating =================================================

-- create_table(:comments)

   -> 0.015s

==  CreateComments: migrated (0.0119s) ========================================
```

__Associating Models__

Active Record associations let you easily declare the relationship between two models. In the case of comments and articles, you could write out the relationships this way:

- Each comment belongs to one article.
- One article can have many comments.

In fact, this is very close to the syntax that Rails uses to declare this association. You've already seen the line of code inside the Comment model (app/models/comment.rb) that makes each comment belong to an:

```
class Comment < ApplicationRecord

  belongs_to :article

end
```

You'll need to edit app/models/article.rb to add the other side of the association:

```ruby
class Article < ApplicationRecord
  has_many :comments


  validates :title, presence: true
  validates :body, presence: true, length: { minimum: 10 }
end
```

Adding a Route for Comments

As with the articles controller, we will need to add a route so that Rails knows where we would like to navigate to see comments. Open up the config/routes.rb file again, and edit it as follows:

```ruby
Rails.application.routes.draw do
  root "articles#index"
  resources :articles do
    resources :comments
  end
end
```

This creates comments as a *nested resource* within articles. This is another part of capturing the hierarchical relationship that exists between articles and comments.

Generating a Controller

With the model in hand, you can turn your attention to creating a matching controller. Again, we'll use the same generator we used before

```
C:\Users\hp\blog>bin/rails generate controller Comments
```

This creates three files and one empty directory:

File/Directory	Purpose
app/controllers/comments_controller.rb	The Comments controller
app/views/comments/	Views of the controller are stored here
test/controllers/comments_controller_test.rb	The test for the controller

File/Directory	Purpose
app/helpers/comments_helper.rb	A view helper file

Like with any blog, our readers will create their comments directly after reading the article, and once they have added their comment, will be sent back to the article show page to see their comment now listed. Due to this, our CommentsController is there to provide a method to create comments and delete spam comments when they arrive.

So first, we'll wire up the Article show template (app/views/articles/show.html.erb) to let us make a new comment:

```
<h1><%= @article.title %></h1>

<p><%= @article.body %></p>

<ul>
  <li><%= link_to "Edit", edit_article_path(@article) %></li>
  <li><%= link_to "Destroy", article_path(@article), data: {
                    turbo_method: :delete,
                    turbo_confirm: "Are you sure?"
                  } %></li>
</ul>

<h2>Add a comment:</h2>
<%= form_with model: [ @article, @article.comments.build ] do |form| %>
  <p>
    <%= form.label :commenter %><br>
    <%= form.text_field :commenter %>
  </p>
  <p>
    <%= form.label :body %><br>
    <%= form.text_area :body %>
  </p>
  <p>
    <%= form.submit %>
  </p>
<% end %>
```

This adds a form on the Article show page that creates a new comment by calling the CommentsController create action. The form_with call here uses an array, which will build a nested route, such as /articles/1/comments.

Let's wire up the create in app/controllers/comments_controller.rb:

```ruby
class CommentsController < ApplicationController
  def create
    @article = Article.find(params[:article_id])
    @comment = @article.comments.create(comment_params)
    redirect_to article_path(@article)
  end

  private
    def comment_params
      params.require(:comment).permit(:commenter, :body)
    end
end
```

You'll see a bit more complexity here than you did in the controller for articles. That's a side-effect of the nesting that you've set up. Each request for a comment has to keep track of the article to which the comment is attached, thus the initial call to the find method of the Article model to get the article in question.

In addition, the code takes advantage of some of the methods available for an association. We use the create method on @article.comments to create and save the comment. This will automatically link the comment so that it belongs to that particular article.

Once we have made the new comment, we send the user back to the original article using the article_path(@article) helper. As we have already seen, this calls the show action of the ArticlesController which in turn renders the show.html.erb template. This is where we want the comment to show, so let's add that to the app/views/articles/show.html.erb.

```erb
<h1><%= @article.title %></h1>

<p><%= @article.body %></p>
```

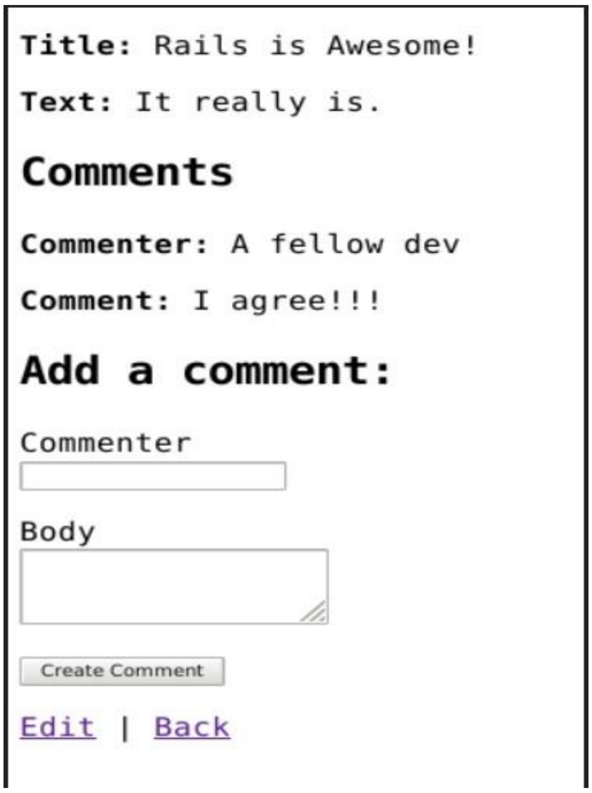
```
<ul>
  <li><%= link_to "Edit", edit_article_path(@article) %></li>
  <li><%= link_to "Destroy", article_path(@article), data: {
                    turbo_method: :delete,
                    turbo_confirm: "Are you sure?"
                  } %></li>
</ul>

<h2>Comments</h2>
<% @article.comments.each do |comment| %>
  <p>
    <strong>Commenter:</strong>
    <%= comment.commenter %>
  </p>

  <p>
    <strong>Comment:</strong>
    <%= comment.body %>
  </p>
<% end %>

<h2>Add a comment:</h2>
<%= form_with model: [ @article, @article.comments.build ] do |form| %>
  <p>
    <%= form.label :commenter %><br>
    <%= form.text_field :commenter %>
  </p>
  <p>
    <%= form.label :body %><br>
    <%= form.text_area :body %>
  </p>
  <p>
```

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


Title: Rails is Awesome!

Text: It really is.

Comments

Commenter: A fellow dev

Comment: I agree!!!

Add a comment:

Refactoring

Now that we have articles and comments working, take a look at the app/views/articles/show.html.erb template. It is getting long and awkward. We can use partials to clean it up.

Rendering Partial Collections

First, we will make a comment partial to extract showing all the comments for the article. Create the file app/views/comments/_comment.html.erb and put the following into it:

```
<p>
  <strong>Commenter:</strong>
  <%= comment.commenter %>
</p>
<p>
  <strong>Comment:</strong>
  <%= comment.body %>
</p>
```

Then you can change app/views/articles/show.html.erb to look like the following:

```erb
<h1><%= @article.title %></h1>

<p><%= @article.body %></p>

<ul>
  <li><%= link_to "Edit", edit_article_path(@article) %></li>
  <li><%= link_to "Destroy", article_path(@article), data: {
                    turbo_method: :delete,
                    turbo_confirm: "Are you sure?"
                  } %></li>
</ul>

<h2>Comments</h2>
<%= render @article.comments %>

<h2>Add a comment:</h2>
<%= form_with model: [ @article, @article.comments.build ] do |form| %>
  <p>
    <%= form.label :commenter %><br>
    <%= form.text_field :commenter %>
  </p>
  <p>
    <%= form.label :body %><br>
    <%= form.text_area :body %>
  </p>
  <p>
    <%= form.submit %>
  </p>
<% end %>
```

This will now render the partial in app/views/comments/_comment.html.erb once for each comment that is in the @article.comments collection. As the render method iterates over the @article.comments collection, it assigns each comment to a local variable named the same as the partial, in this case comment, which is then available in the partial for us to show.

Rendering a Partial Form

Let us also move that new comment section out to its own partial. Again, you create a file app/views/comments/_form.html.erb containing:

```
<%= form_with model: [ @article, @article.comments.build ] do |form| %>
  <p>
    <%= form.label :commenter %><br>
    <%= form.text_field :commenter %>
  </p>
  <p>
    <%= form.label :body %><br>
    <%= form.text_area :body %>
  </p>
  <p>
    <%= form.submit %>
  </p>
<% end %>
```

Then you make the app/views/articles/show.html.erb look like the following:

```
<h1><%= @article.title %></h1>

<p><%= @article.body %></p>

<ul>
  <li><%= link_to "Edit", edit_article_path(@article) %></li>
  <li><%= link_to "Destroy", article_path(@article), data: {
                    turbo_method: :delete,
                    turbo_confirm: "Are you sure?"
                  } %></li>
```

```
</ul>


<h2>Comments</h2>
<%= render @article.comments %>


<h2>Add a comment:</h2>
<%= render 'comments/form' %>
```

The second render just defines the partial template we want to render, comments/form. Rails is smart enough to spot the forward slash in that string and realize that you want to render the _form.html.erb file in the app/views/comments directory.

The @article object is available to any partials rendered in the view because we defined it as an instance variable.

Using Concerns

Concerns are a way to make large controllers or models easier to understand and manage. This also has the advantage of reusability when multiple models (or controllers) share the same concerns. Concerns are implemented using modules that contain methods representing a well-defined slice of the functionality that a model or controller is responsible for. In other languages, modules are often known as mixins.

You can use concerns in your controller or model the same way you would use any module. When you first created your app with rails new blog, two folders were created within app/ along with the rest:

```
app/controllers/concerns
app/models/concerns
```

In the example below, we will implement a new feature for our blog that would benefit from using a concern. Then, we will create a concern, and refactor the code to use it, making the code more DRY and maintainable.

A blog article might have various statuses - for instance, it might be visible to everyone (i.e. public), or only visible to the author (i.e. private). It may also be hidden to all but still retrievable (i.e. archived). Comments may similarly be hidden or visible. This could be represented using a status column in each model.

First, let's run the following migrations to add status to Articles and Comments:

```
bin/rails generate migration AddStatusToArticles status:string bin/rails
generate migration AddStatusToComments status:string
```

And next, let's update the database with the generated migrations:

```
bin/rails db:migrate
```

We also have to permit the :status key as part of the strong parameter, in app/controllers/articles_controller.rb:

```
private

    def article_params

      params.require(:article).permit(:title, :body, :status)

    end
```

and in app/controllers/comments_controller.rb:

```
private

    def comment_params

      params.require(:comment).permit(:commenter, :body, :status)

    end
```

Within the article model, after running a migration to add a status column using bin/rails db:migrate command, you would add

```
class Article < ApplicationRecord

  has_many :comments


  validates :title, presence: true

  validates :body, presence: true, length: { minimum: 10 }


  VALID_STATUSES = ['public', 'private', 'archived']


  validates :status, inclusion: { in: VALID_STATUSES }


  def archived?

    status == 'archived'

  end

end
```

and in the Comment model:

```
class Comment < ApplicationRecord
  belongs_to :article


  VALID_STATUSES = ['public', 'private', 'archived']


  validates :status, inclusion: { in: VALID_STATUSES }


  def archived?
    status == 'archived'
  end
end
```

Then, in our index action template (app/views/articles/index.html.erb) we would use the archived? method to avoid displaying any article that is archived:

```
<h1>Articles</h1>


<ul>
  <% @articles.each do |article| %>
    <% unless article.archived? %>
      <li>
        <%= link_to article.title, article %>
      </li>
    <% end %>
  <% end %>
</ul>


<%= link_to "New Article", new_article_path %>
```

Similarly, in our comment partial view (app/views/comments/_comment.html.erb) we would use the archived? method to avoid displaying any comment that is archived:

```
<% unless comment.archived? %>
  <p>
    <strong>Commenter:</strong>
    <%= comment.commenter %>
  </p>

  <p>
    <strong>Comment:</strong>
    <%= comment.body %>
  </p>
<% end %>
```

However, if you look again at our models now, you can see that the logic is duplicated. If in the future we increase the functionality of our blog - to include private messages, for instance - we might find ourselves duplicating the logic yet again. This is where concerns come in handy.

A concern is only responsible for a focused subset of the model's responsibility; the methods in our concern will all be related to the visibility of a model. Let's call our new concern (module) Visible. We can create a new file inside app/models/concerns called visible.rb , and store all of the status methods that were duplicated in the models.

app/models/concerns/visible.rb

```
module Visible
  def archived?
    status == 'archived'
  end
end
```

We can add our status validation to the concern, but this is slightly more complex as validations are methods called at the class level.
The ActiveSupport::Concern (API Guide) gives us a simpler way to include them:

```
module Visible
  extend ActiveSupport::Concern

  VALID_STATUSES = ['public', 'private', 'archived']

  included do
    validates :status, inclusion: { in: VALID_STATUSES }
  end

  def archived?
    status == 'archived'
  end
end
```

Now, we can remove the duplicated logic from each model and instead include our new Visible module:

In app/models/article.rb:

```
class Article < ApplicationRecord
  include Visible

  has_many :comments

  validates :title, presence: true
  validates :body, presence: true, length: { minimum: 10 }
end
```

and in app/models/comment.rb

```
class Comment < ApplicationRecord
  include Visible
  belongs_to :article
end
```

Class methods can also be added to concerns. If we want to display a count of public articles or comments on our main page, we might add a class method to Visible as follows

```
module Visible
  extend ActiveSupport::Concern
  VALID_STATUSES = ['public', 'private', 'archived']


  included do
    validates :status, inclusion: { in: VALID_STATUSES }
  end
  class_methods do
    def public_count
      where(status: 'public').count
    end
  end
  def archived?
    status == 'archived'
  end
end
```

Then in the view, you can call it like any class method:

```
<h1>Articles</h1>


Our blog has <%= Article.public_count %> articles and counting!
<ul>
  <% @articles.each do |article| %>
    <% unless article.archived? %>
      <li>
        <%= link_to article.title, article %>
      </li>
    <% end %>
  <% end %>
</ul>
<%= link_to "New Article", new_article_path %>
```

To finish up, we will add a select box to the forms, and let the user select the status when they create a new article or post a new comment. We can also specify the default status as public.
In app/views/articles/_form.html.erb, we can add:

```
<div>

  <%= form.label :status %><br>

  <%= form.select :status, ['public', 'private', 'archived'], selected: 'public' %>

</div>
```

and in app/views/comments/_form.html.erb:

```
<p>

  <%= form.label :status %><br>

  <%= form.select :status, ['public', 'private', 'archived'], selected: 'public' %>

</p>
```

Deleting Comments

Another important feature of a blog is being able to delete spam comments. To do this, we need to implement a link of some sort in the view and a destroy action in the CommentsController.

So first, let's add the delete link in the app/views/comments/_comment.html.erb partial:

```
<% unless comment.archived? %>

  <p>

    <strong>Commenter:</strong>

    <%= comment.commenter %>

  </p>

  <p>

    <strong>Comment:</strong>

    <%= comment.body %>

  </p>

  <p>

    <%= link_to "Destroy Comment", [comment.article, comment], data: {

                  turbo_method: :delete,

                  turbo_confirm: "Are you sure?"

                } %>
```

```
 </p>
<% end %>
```

Clicking this new "Destroy Comment" link will fire off a DELETE /articles/:article_id/comments/:id to our CommentsController, which can then use this to find the comment we want to delete, so let's add a destroy action to our controller (app/controllers/comments_controller.rb):

```
class CommentsController < ApplicationController
  def create
    @article = Article.find(params[:article_id])
    @comment = @article.comments.create(comment_params)
    redirect_to article_path(@article)
  end

  def destroy
    @article = Article.find(params[:article_id])
    @comment = @article.comments.find(params[:id])
    @comment.destroy
    redirect_to article_path(@article), status: :see_other
  end

  private
    def comment_params
      params.require(:comment).permit(:commenter, :body, :status)
    end
end
```

The destroy action will find the article we are looking at, locate the comment within the @article.comments collection, and then remove it from the database and send us back to the show action for the article.

Deleting Associated Objects

If you delete an article, its associated comments will also need to be deleted, otherwise they would simply occupy space in the database. Rails allows you to use the dependent option of an association to achieve this. Modify the Article model, app/models/article.rb, as follows:

```
class Article < ApplicationRecord
  include Visible


  has_many :comments, dependent: :destroy


  validates :title, presence: true
  validates :body, presence: true, length: { minimum: 10 }
end
```

Security

Basic Authentication

If you were to publish your blog online, anyone would be able to add, edit and delete articles or delete comments.

Rails provides an HTTP authentication system that will work nicely in this situation.

In the ArticlesController we need to have a way to block access to the various actions if the person is not authenticated. Here we can use the Rails http_basic_authenticate_with method, which allows access to the requested action if that method allows it.

To use the authentication system, we specify it at the top of our ArticlesController in app/controllers/articles_controller.rb. In our case, we want the user to be authenticated on every action except index and show, so we write that:

```
class ArticlesController < ApplicationController


  http_basic_authenticate_with name: "dhh", password: "secret", except:
[:index, :show]


  def index
    @articles = Article.all
  end
```

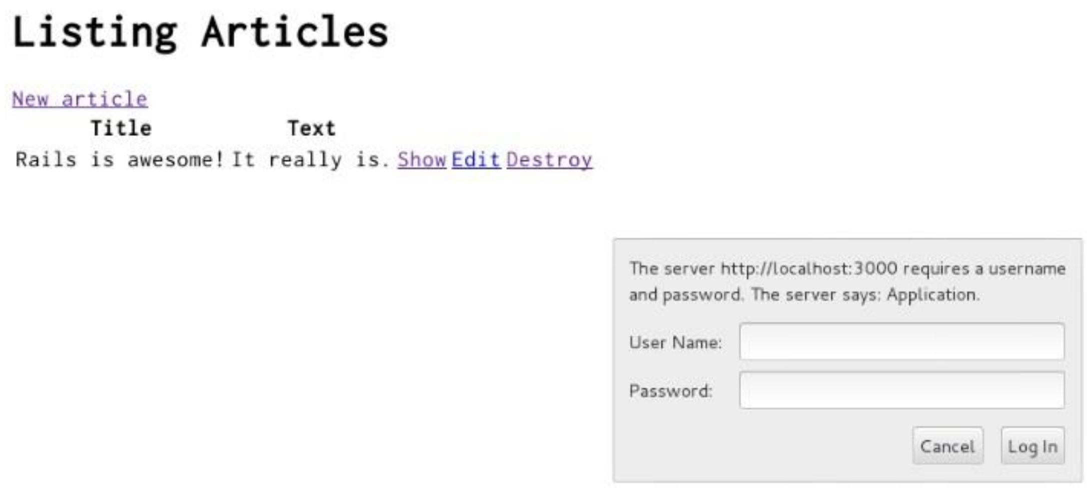
We also want to allow only authenticated users to delete comments, so in the CommentsController (app/controllers/comments_controller.rb) we write:

```
class CommentsController < ApplicationController

  http_basic_authenticate_with name: "dhh", password: "secret", only:
:destroy

  def create
    @article = Article.find(params[:article_id])
    # ...
  end
```

Now if you try to create a new article, you will be greeted with a basic HTTP Authentication challenge:



Listing Articles

New article

Title	Text			
Rails is awesome!	It really is.	Show	Edit	Destroy

The server http://localhost:3000 requires a username and password. The server says: Application.

User Name:

Password:

After entering the correct username and password, you will remain authenticated until a different username and password is required or the browser is closed.

Other authentication methods are available for Rails applications. Two popular authentication add-ons for Rails are the Devise rails engine and the Authlogic gem, along with a number of others.