Blogger 2

In this project you'll create a simple blog system and learn the basics of Ruby on Rails including:

- Models, Views, and Controllers (MVC)
- Data Structures & Relationships
- Routing
- Migrations
- Views with forms, partials, and helpers
- RESTful design
- Adding gems for extra features

> NOTE
>
> This tutorial is open source. If you notice errors, typos, or have questions/suggestions, please submit them to the project on GitHub [https://github.com/JumpstartLab/curriculum/blob/master/source/projects/blogger.markdown] .

I0: Up and Running

Part of the reason Ruby on Rails became popular quickly is that it takes a lot of the hard work off your hands, and that's especially true in starting up a project. Rails practices the idea of "sensible defaults" and will, with one command, create a working application ready for your customization.

Setting the Stage

First we need to make sure everything is set up and installed. See the Environment Setup [/topics/environment/environment.html] page for instructions on setting up and verifying your Ruby and Rails environment.

This tutorial targets Rails 4.0.0, and may need slight adaptations for other versions. Let us know if you run into something strange!

From the command line, switch to the folder that will store your projects. For instance, I use `/Users/jcasimir/projects/`. Within that folder, run the following command:

Terminal

```
$ rails new blogger
```

Use `cd blogger` to change into the directory, then open it in your text editor. If you're using Sublime Text you can do that with `subl .`.

Project Tour

The generator has created a Rails application for you. Let's figure out what's in there. Looking at the project root, we have these folders:

- `app` - This is where 98% of your effort will go. It contains subfolders which will hold most of the code you write including Models, Controllers, Views, Helpers, JavaScript, etc.
- `bin` - This is where your app's executables are stored: `bundle`, `rails`, `rake`, and `spring`.
- `config` - Control the environment settings for your application. It also includes the `initializers` subfolder which holds items to be run on startup.
- `db` - Will eventually have a `migrations` subfolder where your migrations, used to structure the database, will be stored. When using SQLite3, as is the Rails default, the database file will also be stored in this folder.
- `lib` - This folder is to store code you control that is reusable outside the project.
- `log` - Log files, one for each environment (development, test, production)
- `public` - Static files can be stored and accessed from here, but all the interesting things (JavaScript, Images, CSS) have been moved up to `app` since Rails 3.1
- `test` - If your project is using the default `Test::Unit` testing library, the tests will live here
- `tmp` - Temporary cached files
- `vendor` - Infrequently used, this folder is to store code you *do not* control. With Bundler and Rubygems, we generally don't need anything in here during development.

Configuring the Database

Look in the `config` directory and open the file `database.yml`. This file controls how Rails' database connection system will access your database. You can configure many different databases, including SQLite3, MySQL, PostgreSQL, SQL Server, and Oracle.

If you were connecting to an existing database you would enter the database configuration parameters here. Since we're using SQLite3 and starting from scratch, we can leave the defaults to create a new database, which will happen automatically. The database will be stored in `db/development.sqlite3`

Starting the Server

Let's start up the server. From your project directory:

Terminal

```
$ bin/rails server
  => Booting WEBrick
  => Rails 4.0.0 application starting in development on htt
  p://0.0.0.0:3000
  => Call with -d to detach
  => Ctrl-C to shutdown server
  [2012-01-07 11:16:52] INFO  WEBrick 1.3.1
  [2012-01-07 11:16:52] INFO  ruby 1.9.3 (2011-10-30) [x86_
  64-darwin11.2.0]
  [2012-01-07 11:16:52] INFO  WEBrick::HTTPServer#start: pi
  d=36790 port=3000
```

You're ready to go!

Viewing the App

Open any web browser and enter the address `http://0.0.0.0:3000`. You can also use `http://localhost:3000` or `http://127.0.0.1:3000` – they are all "loopback" addresses that point to your machine.

You'll see the Rails' "Welcome Aboard" page. Click the "About your application's environment" link and you should see the versions of various gems. As long as there's no big ugly error message, you're good to go.

Getting an Error?

If you see an error here, it's most likely related to the database. You are probably running Windows and don't have either the SQLite3 application installed or the gem isn't installed properly. Go back to Environment Setup [/topics/environment/environment.html] and use the Rails Installer package. Make sure you check the box during setup to configure the environment variables. Restart your machine after the installation and give it another try.

Creating the Article Model

Our blog will be centered around "articles," so we'll need a table in the database to store all the articles and a model to allow our Rails app to work with that data. We'll use one of Rails' generators to create the required files. Switch to your terminal and enter the following:

Terminal

```
$ bin/rails generate model Article
```

Note that we use `bin/rails` here but we used `rails` previously. The `rails` command is used for generating new projects, and the `bin/rails` command is used for controlling Rails.

We're running the `generate` script, telling it to create a `model`, and naming that model `Article`. From that information, Rails creates the following files:

- `db/migrate/(some_time_stamp)_create_articles.rb` : A database migration to create the `articles` table
- `app/models/article.rb` : The file that will hold the model code
- `test/models/article_test.rb` : A file to hold unit tests for `Article`
- `test/fixtures/articles.yml` : A fixtures file to assist with unit testing

With those files in place we can start developing!

Working with the Database

Rails uses migration files to perform modifications to the database. Almost any modification you can make to a DB can be done through a migration. The killer feature about Rails migrations is that they're generally database agnostic. When developing applications developers might use SQLite3 as we are in this tutorial, but in production we'll use PostgreSQL. Many others choose MySQL. It doesn't matter – the same migrations will work on all of them! This is an example of how Rails takes some of the painful work off your hands. You write your migrations once, then run them against almost any database.

Migration?

What is a migration? Let's open `db/migrate/(some_time_stamp)_create_articles.rb` and take a look. First you'll notice that the filename begins with a mish-mash of numbers which is a timestamp of when the migration was created. Migrations need to be ordered, so the timestamp serves to keep them in chronological order. Inside the file, you'll see just the method `change`.

Migrations used to have two methods, `up` and `down`. The `up` was used to make your change, and the `down` was there as a safety valve to undo the change. But this usually meant a bunch of extra typing, so with Rails 3.1 those two were replaced with `change`.

We write `change` migrations just like we used to write `up`, but Rails will figure out the undo operations for us automatically.

Modifying `change`

Inside the `change` method you'll see the generator has placed a call to the `create_table` method, passed the symbol `:articles` as a parameter, and created a block with the variable `t` referencing the table that's created.

We call methods on `t` to create columns in the `articles` table. What kind of fields does our Article need to have? Since migrations make it easy to add or change columns later, we don't need to think of everything right now, we just need a few to get us rolling. Let's start with:

- `title` (a string)
- `body` (a "text")

That's it! You might be wondering, what is a "text" type? This is an example of relying on the Rails database adapters to make the right call. For some DBs, large text fields are stored as `varchar`, while others like Postgres use a `text` type. The database adapter will figure out the best choice for us depending on the configured database – we don't have to worry about it.

Add those into your `change` like this:

```
1  def change
2    create_table :articles do |t|
3      t.string :title
4      t.text :body
5
6      t.timestamps
7    end
8  end
```

Timestamps

What is that `t.timestamps` doing there? It will create two columns inside our table named `created_at` and `updated_at`. Rails will manage these columns for us. When an article is created its `created_at` and `updated_at` are automatically set. Each time we make a change to the article, the `updated_at` will automatically be updated.

Running the Migration

Save that migration file, switch over to your terminal, and run this command:

Terminal

```
$ bin/rake db:migrate
```

This command starts the `rake` program which is a ruby utility for running

maintenance-like functions on your application (working with the DB, executing unit tests, deploying to a server, etc).

We tell `rake` to `db:migrate` which means "look in your set of functions for the database (`db`) and run the `migrate` function." The `migrate` action finds all migrations in the `db/migrate/` folder, looks at a special table in the DB to determine which migrations have and have not been run yet, then runs any migration that hasn't been run.

In this case we had just one migration to run and it should print some output like this to your terminal:

Terminal

```
$ bin/rake db:migrate
==  CreateArticles: migrating ==============================
======================
-- create_table(:articles)
-> 0.0012s
==  CreateArticles: migrated (0.0013s) ====================
======================
```

It tells you that it is running the migration named `CreateArticles`. And the "migrated" line means that it completed without errors. When the migrations are run, data is added to the database to keep track of which migrations have *already* been run. Try running `rake db:migrate` again now, and see what happens.

We've now created the `articles` table in the database and can start working on our `Article` model.

Working with a Model in the Console

Another awesome feature of working with Rails is the `console`. The `console` is a command-line interface to your application. It allows you to access and work with just about any part of your application directly instead of going through the web interface. This will accelerate your development process. Once an app is in production the console makes it very easy to do bulk modifications, searches, and other data operations. So let's open the console now by going to your terminal and entering this:

Terminal

```
$ bin/rails console
```

You'll then just get back a prompt of `>>`. You're now inside an `irb` interpreter with full access to your application. Let's try some experiments. Enter each of these commands one at a time and observe the results:

IRB

```
2.1.1 :001>  Time.now
2.1.1 :002>  Article.all
2.1.1 :003>  Article.new
```

The first line was just to demonstrate that you can run normal Ruby, just like `irb`, within your `console`. The second line referenced the `Article` model and called the `all` class method which returns an array of all articles in the database – so far an empty array. The third line created a new article object. You can see that this new object had attributes `id`, `title`, `body`, `created_at`, and `updated_at`.

Looking at the Model

All the code for the `Article` model is in the file `app/models/article.rb`, so let's open that now.

Not very impressive, right? There are no attributes defined inside the model, so how does Rails know that an Article should have a `title`, a `body`, etc? The answer is a technique called reflection. Rails queries the database, looks at the articles table, and assumes that whatever columns that table has should be the attributes for the model.

You'll recognize most of them from your migration file, but what about `id`? Every table you create with a migration will automatically have an `id` column which serves as the table's primary key. When you want to find a specific article, you'll look it up in the articles table by its unique ID number. Rails and the database work together to make sure that these IDs are unique, usually using a special column type in the DB called "serial".

In your console, try entering `Article.all` again. Do you see the blank article that we created with the `Article.new` command? No? The console doesn't change values in the database until we explicitly call the `.save` method on an object. Let's create a sample article and you'll see how it works. Enter each of the following lines one at a time:

IRB

```
2.1.1 :001>  a = Article.new
2.1.1 :002>  a.title = "Sample Article Title"
2.1.1 :003>  a.body = "This is the text for my article, woo ho
```

```
2.1.1 :004> a.save
2.1.1 :005> Article.all
```

Now you'll see that the `Article.all` command gave you back an array holding the one article we created and saved. Go ahead and **create 3 more sample articles**.

Setting up the Router

We've created a few articles through the console, but we really don't have a web application until we have a web interface. Let's get that started. We said that Rails uses an "MVC" architecture and we've worked with the Model, now we need a Controller and View.

When a Rails server gets a request from a web browser it first goes to the *router*. The router decides what the request is trying to do, what resources it is trying to interact with. The router dissects a request based on the address it is requesting and other HTTP parameters (like the request type of GET or PUT). Let's open the router's configuration file, `config/routes.rb`.

Inside this file you'll see a LOT of comments that show you different options for routing requests. Let's remove everything *except* the first line (`Blogger::Application.routes.draw do`) and the final `end`. Then, in between those two lines, add `resources :articles` so your file looks like this:

```
1  Blogger::Application.routes.draw do
2    resources :articles
3  end
```

This line tells Rails that we have a resource named `articles` and the router should expect requests to follow the *RESTful* model of web interaction (REpresentational State Transfer). The details don't matter right now, but when you make a request like `http://localhost:3000/articles/`, the router will understand you're looking for a listing of the articles, and `http://localhost:3000/articles/new` means you're trying to create a new article.

Looking at the Routing Table

Dealing with routes is commonly very challenging for new Rails programmers. There's a great tool that can make it easier on you. To get a list of the routes in your application, go to a command prompt and run `rake routes`. You'll get a listing like this:

Terminal

```
$ bin/rake routes
   Prefix Verb   URI Pattern                  Controller#Act
   ion
   articles GET    /articles(.:format)          articles#ind
   ex
   POST   /articles(.:format)          articles#create
   new_article GET    /articles/new(.:format)      articles#
   new
   edit_article GET    /articles/:id/edit(.:format) articles
   #edit
   article GET    /articles/:id(.:format)      articles#show
   PATCH  /articles/:id(.:format)      articles#update
   PUT    /articles/:id(.:format)      articles#update
   DELETE /articles/:id(.:format)      articles#destroy
```

Experiment with commenting out the `resources :articles` in `routes.rb` and running the command again. Un-comment the line after you see the results.

These are the seven core actions of Rails' REST implementation. To understand the table, let's look at the first row as an example:

Terminal

```
   Prefix Verb   URI Pattern                  Controller#Act
   ion
   articles GET    /articles(.:format)          articles#ind
   ex
```

The left most column says `articles`. This is the *prefix* of the path. The router will provide two methods to us using that name, `articles_path` and `articles_url`. The `_path` version uses a relative path while the `_url` version uses the full URL with protocol, server, and path. The `_path` version is always preferred.

The second column, here `GET`, is the HTTP verb for the route. Web browsers typically submit requests with the verbs `GET` or `POST`. In this column, you'll see other HTTP verbs including `PUT` and `DELETE` which browsers don't actually use. We'll talk more about those later.

The third column is similar to a regular expression which is matched against the requested URL. Elements in parentheses are optional. Markers starting with a `:` will be made available to the controller with that name. In our example line, `/articles(.:format)` will match the URLs `/articles/`, `/articles.json`, `/articles` and other similar forms.

The fourth column is where the route maps to in the applications. Our example has `articles#index`, so requests will be sent to the `index` method of the `ArticlesController` class.

Now that the router knows how to handle requests about articles, it needs a place to actually send those requests, the *Controller*.

Creating the Articles Controller

We're going to use another Rails generator but your terminal has the console currently running. Let's open one more terminal or command prompt and move to your project directory which we'll use for command-line scripts. In that new terminal, enter this command:

Terminal

```
$ bin/rails generate controller articles
```

The output shows that the generator created several files/folders for you:

- `app/controllers/articles_controller.rb` : The controller file itself
- `app/views/articles` : The directory to contain the controller's view templates
- `test/controllers/articles_controller_test.rb` : The controller's unit tests file
- `app/helpers/articles_helper.rb` : A helper file to assist with the views (discussed later)
- `test/helpers/articles_helper_test.rb` : The helper's unit test file
- `app/assets/javascripts/articles.js.coffee` : A CoffeeScript file for this controller
- `app/assets/stylesheets/articles.css.scss` : An SCSS stylesheet for this controller

Let's open up the controller file, `app/controllers/articles_controller.rb`. You'll see that this is basically a blank class, beginning with the `class` keyword and ending with the `end` keyword. Any code we add to the controller must go *between* these two lines.

Defining the Index Action

The first feature we want to add is an "index" page. This is what the app will send back when a user requests `http://localhost:3000/articles/` – following the RESTful conventions, this should be a list of the articles. So when the router sees this request come in, it tries to call the `index` action inside `articles_controller`.

Let's first try it out by entering `http://localhost:3000/articles/` into your web

browser. You should get an error message that looks like this:

```
Unknown action

The action 'index' could not be found for ArticlesController
```

The router tried to call the `index` action, but the articles controller doesn't have a method with that name. It then lists available actions, but there aren't any. This is because our controller is still blank. Let's add the following method inside the controller:

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

Instance Variables

What is that "at" sign doing on the front of `@articles`? That marks this variable as an "instance level variable". We want the list of articles to be accessible from both the controller and the view that we're about to create. In order for it to be visible in both places it has to be an instance variable. If we had just named it `articles`, that local variable would only be available within the `index` method of the controller.

A normal Ruby instance variable is available to all methods within an instance.

In Rails' controllers, there's a *hack* which allows instance variables to be automatically transferred from the controller to the object which renders the view template. So any data we want available in the view template should be promoted to an instance variable by adding a `@` to the beginning.

There are ways to accomplish the same goals without instance variables, but they're not widely used. Check out the Decent Exposure [https://github.com/voxdolo/decent_exposure] gem to learn more.

Creating the Template

Now refresh your browser. The error message changed, but you've still got an error, right?

```
Template is missing

Missing template articles/index, application/index with {:locale=>
[:en], :formats=>[:html], :handlers=>[:erb, :builder, :raw, :ruby,
 :jbuilder, :coffee]}. Searched in: * "/Users/you/projects/blogger
/app/views"
```

The error message is pretty helpful here. It tells us that the app is looking for a (view) template in `app/views/articles/` but it can't find one named `index.erb`. Rails has *assumed* that our `index` action in the controller should have a corresponding `index.erb` view template in the views folder. We didn't have to put any code in the controller to tell it what view we wanted, Rails just figures it out.

In your editor, find the folder `app/views/articles` and, in that folder, create a file named `index.html.erb`.

Naming Templates

Why did we choose `index.html.erb` instead of the `index.erb` that the error message said it was looking for? Putting the HTML in the name makes it clear that this view is for generating HTML. In later versions of our blog we might create an RSS feed which would just mean creating an XML view template like `index.xml.erb`. Rails is smart enough to pick the right one based on the browser's request, so when we just ask for `http://localhost:3000/articles/` it will find the `index.html.erb` and render that file.

Index Template Content

Now you're looking at a blank file. Enter in this view template code which is a mix of HTML and what are called ERB tags:

```
1  <h1>All Articles</h1>
2
3  <ul id="articles">
4    <% @articles.each do |article| %>
5      <li>
6        <%= article.title %>
7      </li>
8    <% end %>
9  </ul>
```

ERB is a templating language that allows us to mix Ruby into our HTML. There are only a few things to know about ERB:

- An ERB clause starts with `<%` or `<%=` and ends with `%>`
- If the clause started with `<%`, the result of the ruby code will be hidden
- If the clause started with `<%=`, the result of the ruby code will be output in place of the clause

Save the file and refresh your web browser. You should see a listing of the articles you created in the console. We've got the start of a web application!

Adding Navigation to the Index

Right now our article list is very plain, let's add some links.

Looking at the Routing Table

Remember when we looked at the Routing Table using `bin/rake routes` from the command line? Look at the left-most column and you'll see the route names. These are useful when creating links.

When we create a link, we'll typically use a "route helper" to specify where the link should point. We want our link to display the single article which happens in the `show` action. Looking at the table, the name for that route is `article` and it requires a parameter `id` in the URL. The route helper we'll use looks like this:

```
1  article_path(id)
```

For example, `article_path(1)` would generate the string `"/articles/1"`. Give the method a different parameter and you'll change the ID on the end.

Completing the Article Links

Back in `app/views/articles/index.html.erb`, find where we have this line:

```
1  <%= article.title %>
```

Instead, let's use a `link_to` helper:

```
1  <%= link_to article.title, article_path(article) %>
```

The first part of this helper after the `link_to`, in this case `article.title`, is the text you want the link to say. The next part is our route helper.

When the template is rendered, it will output HTML like this:

```
1  <a href="/articles/1">First Sample Article</a>
```

New Article Link

At the very bottom of the template, let's add a link to the "Create a New Article" page.

We'll use the `link_to` helper, we want it to display the text `"Create a New Article"`

, and where should it point? Look in the routing table for the `new` action, that's where the form to create a new article will live. You'll find the name `new_article`, so the helper is `new_article_path`. Assemble those three parts and write the link in your template.

But wait, there's one more thing. Our stylesheet for this project is going to look for a certain class on the link to make it look fancy. To add HTML attributes to a link, we include them in a Ruby hash style on the end like this:

```
1  <%= link_to article.title, article_path(article), class: 'article_
   title' %>
```

Or, if you wanted to also have a CSS ID attribute:

```
1  <%= link_to article.title, article_path(article),
2      class: 'article_title', id: "article_#{article.id}" %>
```

Use that technique to add the CSS class `new_article` to your "Create a New Article" link.

```
 1  <h1>All Articles</h1>
 2
 3  <ul id="articles">
 4    <% @articles.each do |article| %>
 5      <li>
 6        <%= link_to article.title, article_path(article) %>
 7      </li>
 8    <% end %>
 9  </ul>
10
11  <%= link_to "Create a New Article", new_article_path, class: "new_
    article" %>
```

Review the Results

Refresh your browser and each sample article title should be a link. If you click the link, you'll get an error as we haven't implemented the `show` method yet. Similarly, the new article link will lead you to a dead end. Let's tackle the `show` next.

Creating the SHOW Action

Click the title link for one of your sample articles and you'll get the "Unknown Action" error we saw before. Remember how we moved forward?

An "action" is just a method of the controller. Here we're talking about the

`ArticlesController`, so our next step is to open `app/controllers/articles_controller.rb` and add a `show` method:

```
1  def show
2
3  end
```

Refresh the browser and you'll get the "Template is Missing" error. Let's pause here before creating the view template.

A Bit on Parameters

Look at the URL: `http://localhost:3000/articles/1`. When we added the `link_to` in the index and pointed it to the `article_path` for this `article`, the router created this URL. Following the RESTful convention, this URL goes to a SHOW method which would display the Article with ID number `1`. Your URL might have a different number depending on which article title you clicked in the index.

So what do we want to do when the user clicks an article title? Find the article, then display a page with its title and body. We'll use the number on the end of the URL to find the article in the database.

Within the controller, we have access to a method named `params` which returns us a hash of the request parameters. Often we'll refer to it as "the `params` hash", but technically it's "the `params` method which returns a hash".

Within that hash we can find the `:id` from the URL by accessing the key `params[:id]`. Use this inside the `show` method of `ArticlesController` along with the class method `find` on the `Article` class:

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

Back to the Template

Refresh your browser and we still have the "Template is Missing" error. Create the file `app/views/articles/show.html.erb` and add this code:

```
1  <h1><%= @article.title %></h1>
2  <p><%= @article.body %></p>
3  <%= link_to "<< Back to Articles List", articles_path %>
```

Refresh your browser and your article should show up along with a link back to the index. We can now navigate from the index to a show page and back.

Styling

This is not a CSS project, so to make it a bit more fun we've prepared a CSS file you can drop in. It should match up with all the example HTML in the tutorial.

Download the file from http://tutorials.jumpstartlab.com/assets/blogger/screen.css [http://tutorials.jumpstartlab.com/assets/blogger/screen.css] and place it in your `app/assets/stylesheets/` folder. It will be automatically picked up by your project.

Saving Your Work On GitHub

Now that we have completed our first feature, it's a great time to start thinking about how to save our project.

If you have not already installed git, please follow the instructions on installation here [http://tutorials.jumpstartlab.com/topics/environment/environment.html] .

Git tracks changes in code throughout time, and is a great tool once you have started working collaboratively. First you need to create a GitHub account [https://github.com/signup/free] .

Next, create a repository [https://github.com/new] for the project and on the command line do;

Terminal

```
$ git init
$ git add .
$ git commit –m "first blogger commit"
$ git remote add origin git@github.com:your_github_username
$ /your_repository_name.git
  git push –u origin master
```

Congratulations! You have pushed the code to your GitHub repository. At any time in the future you can backtrack to this commit and refer to your project in this state. We'll cover this in further detail later on.

I1: Form-based Workflow

We've created sample articles from the console, but that isn't a viable long-term solution. The users of our app will expect to add content through a web interface. In this iteration we'll create an HTML form to submit the article, then all the backend processing to get it into the database.

Creating the NEW Action and View

Previously, we set up the `resources :articles` route in `routes.rb`, and that told Rails that we were going to follow the RESTful conventions for this model named Article. Following this convention, the URL for creating a new article would be `http://localhost:3000/articles/new`. From the articles index, click your "Create a New Article" link and it should go to this path.

Then you'll see an "Unknown Action" error. The router went looking for an action named `new` inside the `ArticlesController` and didn't find it.

First let's create that action. Open `app/controllers/articles_controller.rb` and add this method, making sure it's *inside* the `ArticlesController` class, but *outside* the existing `index` and `show` methods:

```
1  def new
2
3  end
```

Starting the Template

With that defined, refresh your browser and you should get the "Template is Missing" error.

Create a new file `app/views/articles/new.html.erb` with these contents:

```
1  <h1>Create a New Article</h1>
```

Refresh your browser and you should just see the heading "Create a New Article".

> NOTE
>
> Why the name `new.html.erb`? The first piece, `new`, matches the name of the controller method. The second, `html`, specifies the output format sent to the client. The third, `erb`, specifies the language the template is written in. Under different circumstances, we might use `new.json.erb` to output JSON or `new.html.haml` to use the HAML templating language.

Writing a Form

It's not very impressive so far – we need to add a form to the `new.html.erb` so the user can enter in the article title and body. Because we're following the RESTful conventions, Rails can take care of many of the details. Inside that `erb` file, enter this code below your header:

```
<%= form_for(@article) do |f| %>
  <ul>
  <% @article.errors.full_messages.each do |error| %>
    <li><%= error %></li>
  <% end %>
  </ul>
  <p>
    <%= f.label :title %><br />
    <%= f.text_field :title %>
  </p>
  <p>
    <%= f.label :body %><br />
    <%= f.text_area :body %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>
```

What is all that? Let's look at it piece by piece:

- `form_for` is a Rails helper method which takes one parameter, in this case `@article` and a block with the form fields. The first line basically says "Create a form for the object named `@article`, refer to the form by the name `f` and add the following elements to the form..."
- The `f.label` helper creates an HTML label for a field. This is good usability practice and will have some other benefits for us later
- The `f.text_field` helper creates a single-line text box named `title`
- The `f.text_area` helper creates a multi-line text box named `body`
- The `f.submit` helper creates a button labeled "Create"

Does it Work?

Refresh your browser and you'll see this:

```
ArgumentError in Articles#new
Showing /Users/you/projects/blogger/app/views/articles/new.html.er
b where line #2 raised:
First argument in form cannot contain nil or be empty
```

Huh? We didn't call a method `model_name`?

We didn't *explicitly*, but the `model_name` method is called by `form_for`. What's happening here is that we're passing `@article` to `form_for`. Since we haven't created an `@article` in this action, the variable just holds `nil`. The `form_for` method calls `model_name` on `nil`, generating the error above.

Setting up for Reflection

Rails uses some of the *reflection* techniques that we talked about earlier in order to set up the form. Remember in the console when we called `Article.new` to see what fields an `Article` has? Rails wants to do the same thing, but we need to create the blank object for it. Go into your `articles_controller.rb`, and *inside* the `new` method, add this line:

```
1   @article = Article.new
```

Then refresh your browser and your form should come up. Enter in a title, some body text, and click CREATE.

The `create` Action

Your old friend pops up again...

```
1   Unknown action
2   The action 'create' could not be found for ArticlesController
```

We accessed the `new` action to load the form, but Rails' interpretation of REST uses a second action named `create` to process the data from that form. Inside your `articles_controller.rb` add this method (again, *inside* the `ArticlesContoller` class, but *outside* the other methods):

```
1   def create
2
3   end
```

Refresh the page and you'll get the "Template is Missing" error.

We Don't Always Need Templates

When you click the "Create" button, what would you expect to happen? Most web applications would process the data submitted then show you the object. In this case, display the article.

We already have an action and template for displaying an article, the `show`, so there's no sense in creating another template to do the same thing.

Processing the Data

Before we can send the client to the `show`, let's process the data. The data from the

form will be accesible through the `params` method.

To check out the structure and content of `params`, I like to use this trick:

```
1  def create
2    fail
3  end
```

The `fail` method will halt the request allowing you to examine the request parameters.

Refresh/resubmit the page in your browser.

Understanding Form Parameters

The page will say "RuntimeError".

Below the error information is the request information. We are interested in the parameters (I've inserted line breaks for readability):

```
1  {"utf8"=>"✓", "authenticity_token"=>"UDbJdVIJjK+qim3m3N9qtZZKgSI00
2  53S7N8OkoCmDjA=",
3   "article"=>{"title"=>"Fourth Sample", "body"=>"This is my fourth
   sample article."},
    "commit"=>"Create", "action"=>"create", "controller"=>"articles"}
```

What are all those? We see the `{` and `}` on the outside, representing a `Hash`. Within the hash we see keys:

- `utf8` : This meaningless checkmark is a hack to force Internet Explorer to submit the form using UTF-8. Read more on StackOverflow [http://stackoverflow.com/questions/3222013/what-is-the-snowman-param-in-rails-3-forms-for]
- `authenticity_token` : Rails has some built-in security mechanisms to resist "cross-site request forgery". Basically, this value proves that the client fetched the form from your site before submitting the data.
- `article` : Points to a nested hash with the data from the form itself
  - `title` : The title from the form
  - `body` : The body from the form
- `commit` : This key holds the text of the button they clicked. From the server side, clicking a "Save" or "Cancel" button look exactly the same except for this parameter.
- `action` : Which controller action is being activated for this request
- `controller` : Which controller class is being activated for this request

Pulling Out Form Data

Now that we've seen the structure, we can access the form data to mimic the way we created sample objects in the console. In the `create` action, remove the `fail` instruction and, instead, try this:

```
1  def create
2    @article = Article.new
3    @article.title = params[:article][:title]
4    @article.save
5  end
```

If you refresh the page in your browser you'll still get the template error. Add one more line to the action, the redirect:

```
1  redirect_to article_path(@article)
```

Refresh the page and you should go to the show for your new article. (*NOTE*: You've now created the same sample article twice)

More Body

The `show` page has the title, but where's the body? Add a line to the `create` action to pull out the `:body` key from the `params` hash and store it into `@article`.

Then try it again in your browser. Both the `title` and `body` should show up properly.

Fragile Controllers

Controllers are middlemen in the MVC framework. They should know as little as necessary about the other components to get the job done. This controller action knows too much about our model.

To clean it up, let me first show you a second way to create an instance of `Article`. You can call `new` and pass it a hash of attributes, like this:

```
1  def create
2    @article = Article.new(
3      title: params[:article][:title],
4      body: params[:article][:body])
5    @article.save
6    redirect_to article_path(@article)
7  end
```

Try that in your app, if you like, and it'll work just fine.

But look at what we're doing. `params` gives us back a hash, `params[:article]` gives us back the nested hash, and `params[:article][:title]` gives us the string from the form. We're hopping into `params[:article]` to pull its data out and stick it right back into a hash with the same keys/structure.

There's no point in that! Instead, just pass the whole hash:

```
1  def create
2    @article = Article.new(params[:article])
3    @article.save
4    redirect_to article_path(@article)
5  end
```

Test and you'll find that it... blows up! What gives?

For security reasons, it's not a good idea to blindly save parameters sent into us via the params hash. Luckily, Rails gives us a feature to deal with this situation: Strong Parameters.

It works like this: You use two new methods, `require` and `permit`. They help you declare which attributes you'd like to accept. Most of the time, they're used in a helper method. Add the below code to `app/helpers/articles_helper.rb`.

```
1  def article_params
2    params.require(:article).permit(:title, :body)
3  end
```

Now on your articles_controller.rb add: 'include ArticlesHelper' directly below your class name.

You then use this method instead of the `params` hash directly:

```
1  @article = Article.new(article_params)
```

Go ahead and add this helper method to your code, and change the arguments to `new`. It should look like this, in your articles_controller.rb file, when you're done:

```
1  class ArticlesController < ApplicationController
2    include ArticlesHelper
3
4    #...
```

```
5
6     def create
7       @article = Article.new(article_params)
8       @article.save
9
10      redirect_to article_path(@article)
11    end
```

Now in your articles_helper.rb file it should look like this:

```
1   module ArticlesHelper
2
3     def article_params
4       params.require(:article).permit(:title, :body)
5     end
6
7   end
```

We can then re-use this method any other time we want to make an `Article`.

Deleting Articles

We can create articles and we can display them, but when we eventually deliver this to less perfect people than us, they're going to make mistakes. There's no way to remove an article, let's add that next.

We could put delete links on the index page, but instead let's add them to the `show.html.erb` template. Let's figure out how to create the link.

We'll start with the `link_to` helper, and we want it to say the word "delete" on the link. So that'd be:

```
1   <%= link_to "delete", some_path %>
```

But what should `some_path` be? Look at the routes table with `rake routes`. The `destroy` action will be the last row, but it has no name in the left column. In this table the names "trickle down," so look up two lines and you'll see the name `article`.

The helper method for the destroy-triggering route is `article_path`. It needs to know which article to delete since there's an `:id` in the path, so our link will look like this:

```
1   <%= link_to "delete", article_path(@article) %>
```

Go to your browser, load the show page, click the link, and observe what happens.

REST is about Path and Verb

Why isn't the article being deleted? If you look at the server window, this is the response to our link clicking:

Terminal

```
Started GET "/articles/3" for 127.0.0.1 at 2012-01-08 13:
05:39 -0500
Processing by ArticlesController#show as HTML
Parameters: {"id"=>"3"}
Article Load (0.1ms)  SELECT "articles".* FROM "articles"
 WHERE "articles"."id" = ? LIMIT 1  [["id", "3"]]
Rendered articles/show.html.erb within layouts/applicatio
n (5.2ms)
Completed 200 OK in 13ms (Views: 11.2ms | ActiveRecord: 0
.3ms)
```

Compare that to what we see in the routes table:

Terminal

```
DELETE /articles/:id(.:format)      articles#destroy
```

The path `"articles/3"` matches the route pattern `articles/:id`, but look at the verb. The server is seeing a `GET` request, but the route needs a `DELETE` verb. How do we make our link trigger a `DELETE`?

You can't, exactly. While most browsers support all four verbs, `GET`, `PUT`, `POST`, and `DELETE`, HTML links are always `GET`, and HTML forms only support `GET` and `POST`. So what are we to do?

Rails' solution to this problem is to *fake* a `DELETE` verb. In your view template, you can add another attribute to the link like this:

```
1  <%= link_to "delete", article_path(@article), method: :delete %>
```

Through some JavaScript tricks, Rails can now pretend that clicking this link triggers a `DELETE`. Try it in your browser.

The `destroy` Action

Now that the router is recognizing our click as a delete, we need the action. The HTTP verb is `DELETE`, but the Rails method is `destroy`, which is a bit confusing.

Let's define the `destroy` method in our `ArticlesController` so it:

1. Uses `params[:id]` to find the article in the database
2. Calls `.destroy` on that object
3. Redirects to the articles index page

Do that now on your own and test it.

Confirming Deletion

There's one more parameter you might want to add to your `link_to` call in your `show.html.erb`:

```
1  data: {confirm: "Really delete the article?"}
```

This will pop up a JavaScript dialog when the link is clicked. The Cancel button will stop the request, while the OK button will submit it for deletion.

Creating an Edit Action & View

Sometimes we don't want to destroy an entire object, we just want to make some changes. We need an edit workflow.

In the same way that we used `new` to display the form and `create` to process that form's data, we'll use `edit` to display the edit form and `update` to save the changes.

Adding the Edit Link

Again in `show.html.erb`, let's add this:

```
1  <%= link_to "edit", edit_article_path(@article) %>
```

Trigger the `edit_article` route and pass in the `@article` object. Try it!

Implementing the `edit` Action

The router is expecting to find an action in `ArticlesController` named `edit`, so let's add this:

```
1  def edit
```

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

All the `edit` action does is find the object and display the form. Refresh and you'll see the template missing error.

An Edit Form

Create a file `app/views/articles/edit.html.erb` but *hold on before you type anything*. Below is what the edit form would look like:

```
1   <h1>Edit an Article</h1>
2
3   <%= form_for(@article) do |f| %>
4     <ul>
5     <% @article.errors.full_messages.each do |error| %>
6       <li><%= error %></li>
7     <% end %>
8     </ul>
9     <p>
10      <%= f.label :title %><br />
11      <%= f.text_field :title %>
12    </p>
13    <p>
14      <%= f.label :body %><br />
15      <%= f.text_area :body %>
16    </p>
17    <p>
18      <%= f.submit %>
19    </p>
20  <% end %>
```

In the Ruby community there is a mantra of "Don't Repeat Yourself" – but that's exactly what I've done here. This view is basically the same as the `new.html.erb` – the only change is the H1. We can abstract this form into a single file called a *partial*, then reference this partial from both `new.html.erb` and `edit.html.erb`.

Creating a Form Partial

Partials are a way of packaging reusable view template code. We'll pull the common parts out from the form into the partial, then render that partial from both the new template and the edit template.

Create a file `app/views/articles/_form.html.erb` and, yes, it has to have the underscore at the beginning of the filename. Partials always start with an underscore.

Open your `app/views/articles/new.html.erb` and CUT all the text from and including the `form_for` line all the way to its `end`. The only thing left will be your

H1 line.

Add the following code to that view:

```
1  <%= render partial: 'form' %>
```

Now go back to the `_form.html.erb` and paste the code from your clipboard.

Writing the Edit Template

Then look at your `edit.html.erb` file. You already have an H1 header, so add the line which renders the partial.

Testing the Partial

Go back to your articles list and try creating a new article – it should work just fine. Try editing an article and you should see the form with the existing article's data – it works OK until you click "Update Article."

Implementing Update

The router is looking for an action named `update`. Just like the `new` action sends its form data to the `create` action, the `edit` action sends its form data to the `update` action. In fact, within our `articles_controller.rb`, the `update` method will look very similar to `create`:

```
1  def update
2    @article = Article.find(params[:id])
3    @article.update(article_params)
4
5    redirect_to article_path(@article)
6  end
```

The only new bit here is the `update` method. It's very similar to `Article.new` where you can pass in the hash of form data. It changes the values in the object to match the values submitted with the form. One difference from `new` is that `update` automatically saves the changes.

We use the same `article_params` method as before so that we only update the attributes we're allowed to.

Now try editing and saving some of your articles.

Adding a Flash

Our operations are working, but it would be nice if we gave the user some kind of status message about what took place. When we create an article the message might say "Article ‘the-article-title’ was created", or "Article ‘the-article-title’ was removed" for the remove action. We can accomplish this with the `flash`.

The controller provides you with accessors to interact with the `flash`. Calling `flash.notice` will fetch a value, and `flash.notice = "Your Message"` will store the string in the `flash`.

Flash for Update

Let’s look first at the `update` method we just worked on. It currently looks like this:

```
1  def update
2    @article = Article.find(params[:id])
3    @article.update(article_params)
4
5    redirect_to article_path(@article)
6  end
```

We can add a flash message by inserting one line:

```
1  def update
2    @article = Article.find(params[:id])
3    @article.update(article_params)
4
5    flash.notice = "Article '#{@article.title}' Updated!"
6
7    redirect_to article_path(@article)
8  end
```

Testing the Flash

Try editing and saving an article through your browser. Does anything show up?

We need to add the flash to our view templates. The `update` method redirects to the `show`, so we *could* just add the display to our show template.

However, we will use the flash in many actions of the application. Most of the time, it’s preferred to add it to our layout.

Flash in the Layout

If you look in `app/views/layouts/application.html.erb` you’ll find what is called the "application layout". A layout is used to wrap multiple view templates in your application. You can create layouts specific to each controller, but most often we’ll

just use one layout that wraps every view template in the application.

Looking at the default layout, you'll see this:

```
<!DOCTYPE html>
<html>
<head>
  <title>Blogger</title>
  <%= stylesheet_link_tag    "application", media: "all", "data-tu
rbolinks-track" => true %>
  <%= javascript_include_tag "application", "data-turbolinks-track
" => true %>
  <%= csrf_meta_tags %>
</head>
<body>

<%= yield %>

</body>
</html>
```

The `yield` is where the view template content will be injected. Just *above* that yield, let's display the flash by adding this:

```
<p class="flash"><%= flash.notice %></p>
```

This outputs the value stored in the `flash` object in the attribute `:notice`.

More Flash Testing

With the layout modified, try changing your article, clicking save, and you should see the flash message appear at the top of the `show` page.

Adding More Messages

Typical controllers will set flash messages in the `update`, `create`, and `destroy` actions. Insert messages into the latter two actions now.

Test out each action/flash, then you're done with I1.

An Aside on the Site Root

It's annoying me that we keep going to `http://localhost:3000/` and seeing the Rails starter page. Let's make the root show our articles index page.

Open `config/routes.rb` and right above the other routes (in this example, right

above `resources :articles`) add in this one:

```
1  root to: 'articles#index'
```

Now visit `http://localhost:3000` and you should see your article list.

Another Save to GitHub.

The form-based workflow is complete, and it is common to commit and push changes after each feature. Go ahead and add/commit/push it up to GitHub:

Terminal

```
$ git add -A
$ git commit -m "form-based workflow feature completed"
$ git push
```

If you are not happy with the code changes you have implemented in this iteration, you don't have to throw the whole project away and restart it. You can use GitHub's reset –hard functionality to roll back to your first commit, and retry this iteration from there. To do so, in your terminal, type in:

Terminal

```
$ git log
  commit 15384dbc144d4cb99dc335ecb1d4608c29c46371
  Author: your_name your_email
  Date:   Thu Apr 11 11:02:57 2013 -0600
  first blogger commit
$ git reset --hard 15384dbc144d4cb99dc335ecb1d4608c29c46371
```

I2: Adding Comments

Most blogs allow the reader to interact with the content by posting comments. Let's add some simple comment functionality.

Designing the Comment Model

First, we need to brainstorm what a comment *is*...what kinds of data does it have...

- It's attached to an article
- It has an author name

- It has a body

With that understanding, let's create a `Comment` model. Switch over to your terminal and enter this line:

Terminal

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

We've already gone through what files this generator creates, we'll be most interested in the migration file and the `comment.rb`.

Setting up the Migration

Open the migration file that the generator created, `db/migrate/some-timestamp_create_comments.rb`. Let's see the fields that were added:

```
1  t.string  :author_name
2  t.text    :body
3  t.references :article
```

Once that's complete, go to your terminal and run the migration:

Terminal

```
$ bin/rake db:migrate
```

Relationships

The power of SQL databases is the ability to express relationships between elements of data. We can join together the information about an order with the information about a customer. Or in our case here, join together an article in the `articles` table with its comments in the `comments` table. We do this by using foreign keys.

Foreign keys are a way of marking one-to-one and one-to-many relationships. An article might have zero, five, or one hundred comments. But a comment only belongs to one article. These objects have a one-to-many relationship – one article connects to many comments.

Part of the big deal with Rails is that it makes working with these relationships very

easy. When we created the migration for comments we started with an `references` field named `article`. The Rails convention for a one-to-many relationship:

- the objects on the "many" end should have a foreign key referencing the "one" object.
- that foreign key should be titled with the name of the "one" object, then an underscore, then "id".

In this case one article has many comments, so each comment has a field named `article_id`.

Following this convention will get us a lot of functionality "for free." Open your `app/models/comment.rb` and check it out:

```
1  class Comment < ActiveRecord::Base
2    belongs_to :article
3  end
```

A comment relates to a single article, it "belongs to" an article. We then want to declare the other side of the relationship inside `app/models/article.rb` like this:

```
1  class Article < ActiveRecord::Base
2    has_many :comments
3  end
```

Now an article "has many" comments, and a comment "belongs to" an article. We have explained to Rails that these objects have a one-to-many relationship.

Testing in the Console

Let's use the console to test how this relationship works in code. If you don't have a console open, go to your terminal and enter `rails console` from your project directory. If you have a console open already, enter the command `reload!` to refresh any code changes.

Run the following commands one at a time and observe the output:

IRB

```
2.1.1 :001>  a = Article.first
2.1.1 :002>  a.comments
2.1.1 :003>  Comment.new
2.1.1 :004>  a.comments.new
2.1.1 :005>  a.comments
```

When you called the `comments` method on object `a`, it gave you back a blank array because that article doesn't have any comments. When you executed `Comment.new` it gave you back a blank Comment object with those fields we defined in the migration.

But, if you look closely, when you did `a.comments.new` the comment object you got back wasn't quite blank – it has the `article_id` field already filled in with the ID number of article `a`. Additionally, the following (last) call to `a.comments` shows that the new comment object has already been added to the in-memory collection for the `a` article object.

Try creating a few comments for that article like this:

IRB

```
2.1.1 :001> c = a.comments.new
2.1.1 :002> c.author_name = "Daffy Duck"
2.1.1 :003> c.body = "I think this article is thhh-thhh-thupi
2.1.1 :004> c.save
2.1.1 :005> d = a.comments.create(author_name: "Chewbacca", b
            WR!")
```

For the first comment, `c`, I used a series of commands like we've done before. For the second comment, `d`, I used the `create` method. `new` doesn't send the data to the database until you call `save`. With `create` you build and save to the database all in one step.

Now that you've created a few comments, try executing `a.comments` again. Did your comments all show up? When I did it, only one comment came back. The console tries to minimize the number of times it talks to the database, so sometimes if you ask it to do something it's already done, it'll get the information from the cache instead of really asking the database – giving you the same answer it gave the first time. That can be annoying. To force it to clear the cache and lookup the accurate information, try this:

IRB

```
2.1.1 :001> a.reload
2.1.1 :002> a.comments
```

You'll see that the article has associated comments. Now we need to integrate them into the article display.

Displaying Comments for an Article

We want to display any comments underneath their parent article. Open `app/views/articles/show.html.erb` and add the following lines right before the link to the articles list:

```
1  <h3>Comments</h3>
2  <%= render partial: 'articles/comment', collection: @article.comme
   nts %>
```

This renders a partial named `"comment"` and that we want to do it once for each element in the collection `@article.comments`. We saw in the console that when we call the `.comments` method on an article we'll get back an array of its associated comment objects. This render line will pass each element of that array one at a time into the partial named `"comment"`. Now we need to create the file `app/views/articles/_comment.html.erb` and add this code:

```
1  <div>
2    <h4>Comment by <%= comment.author_name %></h4>
3    <p class="comment"><%= comment.body %></p>
4  </div>
```

Display one of your articles where you created the comments, and they should all show up.

Web-Based Comment Creation

Good start, but our users can't get into the console to create their comments. We'll need to create a web interface.

Building a Comment Form Partial

The lazy option would be to add a "New Comment" link to the article `show` page. A user would read the article, click the link, go to the new comment form, enter their comment, click save, and return to the article.

But, in reality, we expect to enter the comment directly on the article page. Let's look at how to embed the new comment form onto the article `show`.

Just above the "Back to Articles List" in the articles `show.html.erb`:

```
1  <%= render partial: 'comments/form' %>
```

This is expecting a file `app/views/comments/_form.html.erb`, so create the `app/views/comments/` directory with the `_form.html.erb` file, and add this starter content:

```
1  <h3>Post a Comment</h3>
2  <p>(Comment form will go here)</p>
```

Look at an article in your browser to make sure that partial is showing up. Then we can start figuring out the details of the form.

In the `ArticlesController`

First look in your `articles_controller.rb` for the `new` method.

Remember how we created a blank `Article` object so Rails could figure out which fields an article has? We need to do the same thing before we create a form for the `Comment`.

But when we view the article and display the comment form we're not running the article's `new` method, we're running the `show` method. So we'll need to create a blank `Comment` object inside that `show` method like this:

```
1  @comment = Comment.new
2  @comment.article_id = @article.id
```

Due to the Rails' mass-assignment protection, the `article_id` attribute of the new `Comment` object needs to be manually assigned with the `id` of the `Article`. Why do you think we use `Comment.new` instead of `@article.comments.new`?

Improving the Comment Form

Now we can create a form inside our `comments/_form.html.erb` partial like this:

```
 1  <h3>Post a Comment</h3>
 2
 3  <%= form_for [ @article, @comment ] do |f| %>
 4    <p>
 5      <%= f.label :author_name %><br/>
 6      <%= f.text_field :author_name %>
 7    </p>
 8    <p>
 9      <%= f.label :body %><br/>
10      <%= f.text_area :body %>
11    </p>
12    <p>
13      <%= f.submit 'Submit' %>
```

```
14      </p>
15    <% end %>
```

Trying the Comment Form

Save and refresh your web browser and you'll get an error like this:

```
1   NoMethodError in Articles#show
2   Showing app/views/comments/_form.html.erb where line #3 raised:
3   undefined method `article_comments_path' for #<ActionView::Base:0x
    10446e510>
```

The `form_for` helper is trying to build the form so that it submits to `article_comments_path`. That's a helper which we expect to be created by the router, but we haven't told the router anything about `Comments` yet. Open `config/routes.rb` and update your article to specify comments as a sub-resource.

```
1   resources :articles do
2     resources :comments
3   end
```

Then refresh your browser and your form should show up. Try filling out the comments form and click SUBMIT – you'll get an error about `uninitialized constant CommentsController`.

NOTE

Did you figure out why we aren't using `@article.comments.new`? If you want, edit the `show` action and replace `@comment = Comment.new` with `@comment = @article.comments.new`. Refresh the browser. What do you see?

For me, there is an extra empty comment at the end of the list of comments. That is due to the fact that `@article.comments.new` has added the new `Comment` to the in-memory collection for the `Article`. Don't forget to change this back.

Creating a Comments Controller

Just like we needed an `articles_controller.rb` to manipulate our `Article` objects, we'll need a `comments_controller.rb`.

Switch over to your terminal to generate it:

Terminal

```
$ bin/rails generate controller comments
```

Writing `CommentsController.create`

The comment form is attempting to create a new `Comment` object which triggers the `create` action. How do we write a `create`?

You can cheat by looking at the `create` method in your `articles_controller.rb`. For your `comments_controller.rb`, the instructions should be the same just replace `Article` with `Comment`.

There is one tricky bit, though! We need to assign the article id to our comment like this:

```
1  def create
2    @comment = Comment.new(comment_params)
3    @comment.article_id = params[:article_id]
4
5    @comment.save
6
7    redirect_to article_path(@comment.article)
8  end
9
10 def comment_params
11   params.require(:comment).permit(:author_name, :body)
12 end
```

After Creation

As a user, imagine you write a witty comment, click save, then what would you expect? Probably to see the article page, maybe automatically scrolling down to your comment.

At the end of our `create` action in `CommentsController`, how do we handle the redirect? Instead of showing them the single comment, let's go back to the article page:

```
1  redirect_to article_path(@comment.article)
```

Recall that `article_path` needs to know *which* article we want to see. We might not have an `@article` object in this controller action, but we can find the `Article` associated with this `Comment` by calling `@comment.article`.

Test out your form to create another comment now – and it should work!

Cleaning Up

We've got some decent comment functionality, but there are a few things we should add and tweak.

Comments Count

Let's make it so where the view template has the "Comments" header it displays how many comments there are, like "Comments (3)". Open up your article's `show.html.erb` and change the comments header so it looks like this:

```
1  <h3>Comments (<%= @article.comments.size %>)</h3>
```

Form Labels

The comments form looks a little silly with "Author Name". It should probably say "Your Name", right? To change the text that the label helper prints out, you pass in the desired text as a second parameter, like this:

```
1  <%= f.label :author_name, "Your Name"  %>
```

Change your `comments/_form.html.erb` so it has labels "Your Name" and "Your Comment".

Add a Timestamp to the Comment Display

We should add something about when the comment was posted. Rails has a really neat helper named `distance_of_time_in_words` which takes two dates and creates a text description of their difference like "32 minutes later", "3 months later", and so on.

You can use it in your `_comment.html.erb` partial like this:

```
1  <p>Posted <%= distance_of_time_in_words(comment.article.created_at
   , comment.created_at) %> later</p>
```

With that, you're done with I2!

Time to Save to GitHub Again!

Now that the comments feature has been added push it up to GitHub:

Terminal

```
$ git add .
$ git commit -m "finished blog comments feature"
```

```
$ git push
```

I3: Tagging

In this iteration we'll add the ability to tag articles for organization and navigation.

First we need to think about what a tag is and how it'll relate to the Article model. If you're not familiar with tags, they're commonly used in blogs to assign the article to one or more categories.

For instance, if I write an article about a feature in Ruby on Rails, I might want it tagged with all of these categories: "ruby", "rails" and "programming". That way if one of my readers is looking for more articles about one of those topics they can click on the tag and see a list of my articles with that tag.

Understanding the Relationship

What is a tag? We need to figure that out before we can create the model. First, a tag must have a relationship to an article so they can be connected. A single tag, like "ruby" for instance, should be able to relate to *many* articles. On the other side of the relationship, the article might have multiple tags (like "ruby", "rails", and "programming" as above) - so it's also a *many* relationship. Articles and tags have a *many-to-many* relationship.

Many-to-many relationships are tricky because we're using an SQL database. If an Article "has many" tags, then we would put the foreign key `article_id` inside the `tags` table - so then a Tag would "belong to" an Article. But a tag can connect to *many* articles, not just one. We can't model this relationship with just the `articles` and `tags` tables.

When we start thinking about the database modeling, there are a few ways to achieve this setup. One way is to create a "join table" that just tracks which tags are connected to which articles. Traditionally this table would be named `articles_tags` and Rails would express the relationships by saying that the Article model `has_and_belongs_to_many` Tags, while the Tag model `has_and_belongs_to_many` Articles.

Most of the time this isn't the best way to really model the relationship. The connection between the two models usually has value of its own, so we should promote it to a real model. For our purposes, we'll introduce a model named "Tagging" which is the connection between Articles and Tags. The relationships will setup like this:

- An Article `has_many` Taggings
- A Tag `has_many` Taggings
- A Tagging `belongs_to` an Article and `belongs_to` a Tag

Making Models

With those relationships in mind, let's design the new models:

- Tag
  - `name`: A string
- Tagging
  - `tag_id`: Integer holding the foreign key of the referenced Tag
  - `article_id`: Integer holding the foreign key of the referenced Article

Note that there are no changes necessary to Article because the foreign key is stored in the Tagging model. So now lets generate these models in your terminal:

Terminal

```
$ bin/rails generate model Tag name:string
$ bin/rails generate model Tagging tag:references article:r
$ eferences
  bin/rake db:migrate
```

Expressing Relationships

Now that our model files are generated we need to tell Rails about the relationships between them. For each of the files below, add these lines:

In `app/models/article.rb`:

```
1  has_many :taggings
```

In `app/models/tag.rb`:

```
1  has_many :taggings
```

After Rails had been around for awhile, developers were finding this kind of relationship very common. In practical usage, if I had an object named `article` and I wanted to find its Tags, I'd have to run code like this:

```
1  tags = article.taggings.collect{|tagging| tagging.tag}
```

That's a pain for something that we need commonly.

An article has a list of tags through the relationship of taggings. In Rails we can express this "has many" relationship through an existing "has many" relationship. We will update our article model and tag model to express that relationship.

In `app/models/article.rb`:

```
1  has_many :taggings
2  has_many :tags, through: :taggings
```

In `app/models/tag.rb`:

```
1  has_many :taggings
2  has_many :articles, through: :taggings
```

Now if we have an object like `article` we can just ask for `article.tags` or, conversely, if we have an object named `tag` we can ask for `tag.articles`.

To see this in action, start the `bin/rails console` and try the following:

IRB

```
2.1.1 :001> a = Article.first
2.1.1 :002> a.tags.create name: "tag1"
2.1.1 :003> a.tags.create name: "tag2"
2.1.1 :004> a.tags
            => [#<Tag id: 1, name: "tag1", created_at: "2012-
            :17:55", updated_at: "2012-11-28 20:17:55">, #<Ta
             name: "tag2", created_at: "2012-11-28 20:31:49",
            _at: "2012-11-28 20:31:49">]
```

An Interface for Tagging Articles

The first interface we're interested in is within the article itself. When I write an article, I want to have a text box where I can enter a list of zero or more tags separated by commas. When I save the article, my app should associate my article with the tags with those names, creating them if necessary.

Add the following to our existing form in `app/views/articles/_form.html.erb`:

```
<p>
  <%= f.label :tag_list %><br />
  <%= f.text_field :tag_list %>
</p>
```

With that added, try to create an new article in your browser and your should see this error:

```
NoMethodError in Articles#new
Showing app/views/articles/_form.html.erb where line #14 raised:
undefined method `tag_list' for #<Article:0x10499bab0>
```

An Article doesn't have an attribute or method named `tag_list`. We made it up in order for the form to display related tags, but we need to add a method to the `article.rb` file like this:

```
def tag_list
  tags.join(", ")
end
```

Back in your console, find that article again, and take a look at the results of `tag_list`:

IRB

```
2.1.1 :001>  reload!
2.1.1 :002>  a = Article.first
2.1.1 :003>  a.tag_list
             => "#<Tag:0x007fe4d60c2430>, #<Tag:0x007fe4d617da
```

That is not quite right. What happened?

Our array of tags is an array of Tag instances. When we joined the array Ruby called the default `#to_s` method on every one of these Tag instances. The default `#to_s` method for an object produces some really ugly output.

We could fix the `tag_list` method by:

- Converting all our tag objects to an array of tag names
- Joining the array of tag names together

```
def tag_list
  self.tags.collect do |tag|
```

```
3       tag.name
4     end.join(", ")
5   end
```

Another alternative is to define a new `Tag#to_s` method which overrides the default:

```
1   class Tag < ActiveRecord::Base
2
3     has_many :taggings
4     has_many :articles, through: :taggings
5
6     def to_s
7       name
8     end
9   end
```

Now, when we try to join our `tags`, it'll delegate properly to our name attribute. This is because `#join` calls `#to_s` on every element of the array.

Your form should now show up and there's a text box at the bottom named "Tag list". Enter content for another sample article and in the tag list enter 'ruby, technology'. Click save. It.... worked?

But it didn't. Click 'edit' again, and you'll see that we're back to the `#<Tag...` business, like before. What gives?

```
1   Started PATCH "/articles/1" for 127.0.0.1 at 2013-07-17 09:25:20 -
2   0400
3   Processing by ArticlesController#update as HTML
4     Parameters: {"utf8"=>"", "authenticity_token"=>"qs2M71Rmb64B7IM1
5   ASULjlI1WL6nWYjgH/eOu8en+Dk=", "article"=>{"title"=>"Sample Articl
    e", "body"=>"This is the text for my article, woo hoo!", "tag_list
    "=>"ruby, technology"}, "commit"=>"Update Article", "id"=>"1"}
      Article Load (0.1ms)  SELECT "articles".* FROM "articles" WHERE
    "articles"."id" = ? LIMIT 1  [["id", "1"]]
    Unpermitted parameters: tag_list
```

Unpermitted parameters? Oh yeah! Strong Parameters has done its job, saving us from parameters we don't want. But in this case, we *do* want that parameter. Open up your `app/helpers/articles_helper.rb` and fix the `article_params` method:

```
1     def article_params
2       params.require(:article).permit(:title, :body, :tag_list)
3     end
```

If you go back and put "ruby, technology" as tags, and click save, you'll get this new

error:

```
1  ActiveRecord::UnknownAttributeError in ArticlesController#create
2  unknown attribute: tag_list
```

What is this all about? Let's start by looking at the form data that was posted when we clicked SAVE. This data is in the terminal where you are running the rails server. Look for the line that starts "Processing ArticlesController#create", here's what mine looks like:

```
1  Processing ArticlesController#create (for 127.0.0.1) [POST]
2    Parameters: {"article"=>{"body"=>"Yes, the samples continue!", "
   title"=>"My Sample", "tag_list"=>"ruby, technology"}, "commit"=>"S
   ave", "authenticity_token"=>"xxi0A3tZtoCUDeoTASi6Xx39wpnHt1QW/6Z1j
   xCMOm8="}
```

The field that's interesting there is the `"tag_list"=>"technology, ruby"`. Those are the tags as I typed them into the form. The error came up in the `create` method, so let's peek at `app/controllers/articles_controller.rb` in the `create` method. See the first line that calls `Article.new(article_params)`? This is the line that's causing the error as you could see in the middle of the stack trace.

Since the `create` method passes all the parameters from the form into the `Article.new` method, the tags are sent in as the string `"technology, ruby"`. The `new` method will try to set the new Article's `tag_list` equal to `"technology, ruby"` but that method doesn't exist because there is no attribute named `tag_list`.

There are several ways to solve this problem, but the simplest is to pretend like we have an attribute named `tag_list`.

We can define the `tag_list=` method inside `article.rb` like this: (***do not delete your original tag_list method***)

```
1  def tag_list=(tags_string)
2
3  end
```

Just leave it blank for now and try to resubmit your sample article with tags. It goes through!

Not So Fast

Did it really work? It's hard to tell. Let's jump into the console and have a look.

IRB

```
2.1.1 :001> a = Article.last
2.1.1 :002> a.tags
```

I bet the console reported that `a` had `[]` tags – an empty list. (It also probably said something about an `ActiveRecord::Associations::CollectionProxy`) So we didn't generate an error, but we didn't create any tags either.

We need to return to the `Article#tag_list=` method in `article.rb` and do some more work.

The `Article#tag_list=` method accepts a parameter, a string like "**tag1, tag2, tag3**" and we need to associate the article with tags that have those names. The pseudo-code would look like this:

- Split the *tags_string* into an array of strings with leading and trailing whitespace removed (so `"tag1, tag2, tag3"` would become `["tag1","tag2","tag3"]`
- For each of those strings...
  - Ensure each one of these strings are unique
  - Look for a Tag object with that name. If there isn't one, create it.
  - Add the tag object to a list of tags for the article
- Set the article's tags to the list of tags that we have found and/or created.

The first step is something that Ruby does very easily using the `String#split` method. Go into your console and try "**tag1, tag2, tag3**".**split**. By default it split on the space character, but that's not what we want. You can force split to work on any character by passing it in as a parameter, like this: `"tag1, tag2, tag3".split(",")`.

Look closely at the output and you'll see that the second element is `" tag2"` instead of `"tag2"` – it has a leading space. We don't want our tag system to end up with different tags because of some extra (non-meaningful) spaces, so we need to get rid of that. The `String#strip` method removes leading or trailing whitespace – try it with `" my sample ".strip`. You'll see that the space in the center is preserved.

So first we split the string, and then trim each and every element and collect those updated items:

IRB

```
2.1.1 :001> "programming, Ruby, rails".split(",").collect{|s|
            .downcase}
```

The `String#split(",")` will create the array with elements that have the extra spaces as before, then the `Array#collect` will take each element of that array and send it into the following block where the string is named `s` and the `String#strip` and `String#downcase` methods are called on it. The `downcase` method is to make sure that "ruby" and "Ruby" don't end up as different tags. This line should give you back `["programming", "ruby", "rails"]`.

Lastly, we want to make sure that each and every tag in the list is unique. `Array#uniq` allows us to remove duplicate items from an array.

IRB

```
2.1.1 :001> "programming, Ruby, rails, rails".split(",").coll
            s.strip.downcase}.uniq
```

Now, back inside our `tag_list=` method, let's add this line:

```
1  tag_names = tags_string.split(",").collect{|s| s.strip.downcase}.u
   niq
```

So looking at our pseudo-code, the next step is to go through each of those `tag_names` and find or create a tag with that name. Rails has a built in method to do just that, like this:

```
1  tag = Tag.find_or_create_by(name: tag_name)
```

And finally we need to collect up these new or found new tags and then assign them to our article.

```
1  def tag_list=(tags_string)
2    tag_names = tags_string.split(",").collect{|s| s.strip.downcase}
3  .uniq
4    new_or_found_tags = tag_names.collect { |name| Tag.find_or_creat
5  e_by(name: name) }
     self.tags = new_or_found_tags
   end
```

Testing in the Console

Go back to your console and try these commands:

IRB

```
2.1.1 :001>  reload!
2.1.1 :002>  article = Article.create title: "A Sample Article
             going!", body: "Great article goes here", tag_list
2.1.1 :003>   technology"
             article.tags
```

You should get back a list of the two tags. If you'd like to check the other side of the Article-Tagging-Tag relationship, try this:

IRB

```
2.1.1 :001>  tag = article.tags.first
2.1.1 :002>  tag.articles
```

And you'll see that this Tag is associated with just one Article.

Adding Tags to our Display

According to our work in the console, articles can now have tags, but we haven't done anything to display them in the article pages.

Let's start with `app/views/articles/show.html.erb`. Right below the line that displays the `article.title`, add these lines:

```
1  <p>
2    Tags:
3    <% @article.tags.each do |tag| %>
4      <%= link_to tag.name, tag_path(tag) %>
5    <% end %>
6  </p>
```

Refresh your view and...BOOM:

```
1  NoMethodError in Articles#show
2  Showing app/views/articles/index.html.erb where line #6 raised:
3  undefined method `tag_path' for #<ActionView::Base:0x104aaa460>
```

The `link_to` helper is trying to use `tag_path` from the router, but the router doesn't know anything about our Tag object. We created a model, but we never created a controller or route. There's nothing to link to – so let's generate that controller from your terminal:

Terminal

```
$ bin/rails generate controller tags
```

Then we need to add tags as a resource to our `config/routes.rb`, it should look like this:

```
1  Blogger::Application.routes.draw do
2
3    root to: 'articles#index'
4    resources :articles do
5      resources :comments
6    end
7    resources :tags
8
9  end
```

Refresh your article page and you should see tags, with links, associated with this article.

Listing Articles by Tag

The links for our tags are showing up, but if you click on them you'll see our old friend "No action responded to show." error.

Open `app/controllers/tags_controller.rb` and define a show action:

```
1  def show
2    @tag = Tag.find(params[:id])
3  end
```

Then create the show template `app/views/tags/show.html.erb`:

```
1  <h1>Articles Tagged with <%= @tag.name %></h1>
2
3  <ul>
4    <% @tag.articles.each do |article| %>
5      <li><%= link_to article.title, article_path(article) %></li>
6    <% end %>
7  </ul>
```

Refresh your view and you should see a list of articles with that tag. Keep in mind that there might be some abnormalities from articles we tagged before doing our fixes to the `tag_list=` method. For any article with issues, try going to its `edit`

screen, saving it, and things should be fixed up. If you wanted to clear out all taggings you could do `Tagging.destroy_all` from your console.

Listing All Tags

We've built the `show` action, but the reader should also be able to browse the tags available at `http://localhost:3000/tags`. I think you can do this on your own. Create an `index` action in your `tags_controller.rb` and an `index.html.erb` in the corresponding views folder. Look at your `articles_controller.rb` and Article `index.html.erb` if you need some clues.

Now that we can see all of our tags, we also want the capability to delete them. I think you can do this one on your own too. Create a `destroy` action in your `tags_controller.rb` and edit the `index.html.erb` file you just created. Look at your `articles_controller.rb` and Article `show.html.erb` if you need some clues.

With that, a long Iteration 3 is complete!

Saving to GitHub.

Woah! The tagging feature is now complete. Good on you. Your going to want to push this to the repo.

Terminal

```
$ git add .
$ git commit –m "Tagging feature completed"
$ git push
```

I4: A Few Gems

In this iteration we'll learn how to take advantage of the many plugins and libraries available to quickly add features to your application. First we'll work with `paperclip`, a library that manages file attachments and uploading.

Using the *Gemfile* to Set up a RubyGem

In the past Rails plugins were distributed in zip or tar files that got stored into your application's file structure. One advantage of this method is that the plugin could be easily checked into your source control system along with everything you wrote in the app. The disadvantage is that it made upgrading to newer versions of the plugin, and dealing with the versions at all, complicated.

These days, all Rails plugins are now ‘gems.’ RubyGems is a package management system for Ruby, similar to how Linux distributions use Apt or RPM. There are central servers that host libraries, and we can install those libraries on our machine with a single command. RubyGems takes care of any dependencies, allows us to pick any options if necessary, and installs the library.

Let’s see it in action. Go to your terminal where you have the rails server running, and type `Ctrl-C`. If you have a console session open, type `exit` to exit. Then open up `Gemfile` and look for the lines like this:

```
# To use ActiveModel has_secure_password
# gem 'bcrypt-ruby', '~> 3.0.0'

# To use Jbuilder templates for JSON
# gem 'jbuilder'

# Use unicorn as the app server
# gem 'unicorn'
```

These lines are commented out because they start with the `#` character. By specifying a RubyGem with the `gem` command, we’ll tell the Rails application "Make sure this gem is loaded when you start up. If it isn’t available, freak out!" Here’s how we’ll require the paperclip gem, add this near those commented lines:

```
gem "paperclip"
```

Paperclip is dependent on ImageMagick so you will also need to add that program.

Terminal

```
$ brew install imagemagick
```

When you’re writing a production application, you might specify additional parameters that require a specific version or a custom source for the library. With that config line declared, go back to your terminal and run `rails server` to start the application again. You should get an error like this:

Terminal

```
$ rails server
  Could not find gem 'paperclip (>= 0, runtime)' in any of
  the gem sources listed in your Gemfile.
  Try running `bundle install`.
```

The last line is key – since our config file is specifying which gems it needs, the `bundle` command can help us install those gems. Go to your terminal and:

Terminal

```
$ bundle
```

It should then install the paperclip RubyGem with a version like 3.5.2. In some projects I work on, the config file specifies upwards of 18 gems. With that one `bundle` command the app will check that all required gems are installed with the right version, and if not, install them.

Note: You may need to reload your rails server for the paperclip methods to work.

Now we can start using the library in our application!

Setting up the Database for Paperclip

We want to add images to our articles. To keep it simple, we'll say that a single article could have zero or one images. In later versions of the app maybe we'd add the ability to upload multiple images and appear at different places in the article, but for now the one will show us how to work with paperclip.

First we need to add some fields to the Article model that will hold the information about the uploaded image. Any time we want to make a change to the database we'll need a migration. Go to your terminal and execute this:

Terminal

```
$ bin/rails generate migration add_paperclip_fields_to_arti
  cle
```

That will create a file in your `db/migrate/` folder that ends in `_add_paperclip_fields_to_article.rb`. Open that file now.

Remember that the code inside the `change` method is to migrate the database forward, and Rails should automatically figure out how to undo those changes. We'll use the `add_column` and `remove_column` methods to setup the fields paperclip is expecting:

```
1  class AddPaperclipFieldsToArticle < ActiveRecord::Migration
2    def change
3      add_column :articles, :image_file_name,    :string
```

```
4       add_column :articles, :image_content_type, :string
5       add_column :articles, :image_file_size,    :integer
6       add_column :articles, :image_updated_at,   :datetime
7     end
8   end
```

Then go to your terminal and run `rake db:migrate`. The rake command should show you that the migration ran and added columns to the database.

Adding to the Model

The gem is loaded, the database is ready, but we need to tell our Rails application about the image attachment we want to add. Open `app/models/article.rb` and just below the existing `has_many` lines, add these lines:

```
1   has_attached_file :image
2   validates_attachment_content_type :image, :content_type => ["image
    /jpg", "image/jpeg", "image/png"]
```

This `has_attached_file` method is part of the paperclip library. With that declaration, paperclip will understand that this model should accept a file attachment and that there are fields to store information about that file which start with `image_` in this model's database table.

As of version 4.0, all attachments are required to include a content_type validation, a file_name validation, or to explicitly state that they're not going to have either. Paperclip raises MissingRequiredValidatorError error if you do not do this. So, we add the validates_attachment_content_type line so that our model will validate that it is receiving a proper filetype.

We also have to deal with mass assignment! Modify your `app/helpers/articles_helper.rb` and update the `article_params` method to permit an `:image` as:

```
1     def article_params
2       params.require(:article).permit(:title, :body, :tag_list, :ima
3   ge)
      end
```

Modifying the Form Template

First we'll add the ability to upload the file when editing the article, then we'll add the image display to the article show template. Open your `app/views/articles/_form.html.erb` view template. We need to make two changes...

In the very first line, we need to specify that this form needs to accept "multipart" data. This is an instruction to the browser about how to submit the form. Change your top line so it looks like this:

```
1  <%= form_for(@article, html: {multipart: true}) do |f| %>
```

Then further down the form, right before the paragraph with the save button, let's add a label and field for the file uploading:

```
1  <p>
2    <%= f.label :image, "Attach an Image" %><br />
3    <%= f.file_field :image %>
4  </p>
```

Trying it Out

If your server isn't running, start it up (`rails server` in your terminal). Then go to `http://localhost:3000/articles/` and click EDIT for your first article. The file field should show up towards the bottom. Click the `Choose a File` and select a small image file (a suitable sample image can be found at http://hungryacademy.com/images/beast.png [http://hungryacademy.com/images/beast.png]). Click SAVE and you'll return to the article index. Click the title of the article you just modified. What do you see? Did the image attach to the article?

When I first did this, I wasn't sure it worked. Here's how I checked:

1. Open a console session (`rails console` from terminal)
2. Find the ID number of the article by looking at the URL. In my case, the url was `http://localhost:3000/articles/1` so the ID number is just `1`
3. In console, enter `a = Article.find(1)`
4. Right away I see that the article has data in the `image_file_name` and other fields, so I think it worked.
5. Enter `a.image` to see even more data about the file

Ok, it's in there, but we need it to actually show up in the article. Open the `app/views/articles/show.html.erb` view template. Before the line that displays the body, let's add this line:

```
1  <p><%= image_tag @article.image.url %></p>
```

Then refresh the article in your browser. Tada!

Improving the Form

When first working with the edit form I wasn't sure the upload was working because I expected the `file_field` to display the name of the file that I had already uploaded. Go back to the edit screen in your browser for the article you've been working with. See how it just says "Choose File, no file selected" – nothing tells the user that a file already exists for this article. Let's add that information in now.

So open that `app/views/articles/_form.html.erb` and look at the paragraph where we added the image upload field. We'll add in some new logic that works like this:

- If the article has an image filename *Display the image
- Then display the `file_field` button with the label "Attach a New Image"

So, turning that into code...

```
1  <p>
2    <% if @article.image.exists? %>
3        <%= image_tag @article.image.url %><br/>
4    <% end %>
5    <%= f.label :image, "Attach a New Image" %><br />
6    <%= f.file_field :image %>
7  </p>
```

Test how that looks both for articles that already have an image and ones that don't.

When you "show" an article that doesn't have an image attached it'll have an ugly broken link. Go into your `app/views/articles/show.html.erb` and add a condition like we did in the form so the image is only displayed if it actually exists.

Now our articles can have an image and all the hard work was handled by paperclip!

Further Notes about Paperclip

Yes, a model (in our case an article) could have many attachments instead of just one. To accomplish this you'd create a new model, let's call it "Attachment", where each instance of the model can have one file using the same fields we put into Article above as well as an `article_id` field. The Attachment would then `belong_to` an article, and an article would `have_many` attachments.

Paperclip supports automatic image resizing and it's easy. In your model, you'd add an option like this:

```
1  has_attached_file :image, styles: { medium: "300x300>", thumb: "10
   0x100>" }
```

This would automatically create a "medium" size where the largest dimension is 300 pixels and a "thumb" size where the largest dimension is 100 pixels. Then in your view, to display a specific version, you just pass in an extra parameter like this:

```
1  <%= image_tag @article.image.url(:medium) %>
```

If it's so easy, why don't we do it right now? The catch is that paperclip doesn't do the image manipulation itself, it relies on a package called *imagemagick*. Image processing libraries like this are notoriously difficult to install. If you're on Linux, it might be as simple as `sudo apt-get install imagemagick`. On OS X, if you have Homebrew installed, it'd be `brew install imagemagick`. On windows you need to download and copy some EXEs and DLLs. It can be a hassle, which is why we won't do it during this class.

If you do manage to get imagemagick installed, be advised that the custom sizes will only take affect on those images uploaded *after* the imagemagick installation. In otherwords, when the image is uploaded - Paperclip will use Imagemagick to create the customized sizes specified on the `has_attached_file` line. *This also means that if you change your sizes as a later time, any images that had been previously uploaded won't have versions at those new sizes.*

A Few Sass Examples

All the details about Sass can be found here: http://sass-lang.com/ [http://sass-lang.com/]

We're not focusing on CSS development, so here are a few styles that you can copy & paste and modify to your heart's content. Place the following styles in a new file and save it as styles.css.scss in `app/assets/stylesheets/`.

```
1  $primary_color: #AAA;
2
3  body {
4    background-color: $primary_color;
5    font: {
6      family: Verdana, Helvetica, Arial;
7      size: 14px;
8    }
9  }
10
11 a {
12   color: #0000FF;
13   img {
14     border: none;
15   }
```

```
16  }
17
18  .clear {
19    clear: both;
20    height: 0;
21    overflow: hidden;
22  }
23
24  #container {
25    width: 75%;
26    margin: 0 auto;
27    background: #fff;
28    padding: 20px 40px;
29    border: solid 1px black;
30    margin-top: 20px;
31  }
32
33  #content {
34    clear: both;
35    padding-top: 20px;
36  }
```

If you refresh the page, it should look slightly different! But we didn't add a reference to this stylesheet in our HTML; how did Rails know how to use it? The answer lies in Rails' default layout.

Working with Layouts

We've created about a dozen view templates between our different models. Imagine that Rails *didn't* just figure it out. How would we add this new stylesheet to all of our pages? We *could* go into each of those templates and add a line like this at the top:

```
1  <%= stylesheet_link_tag 'styles' %>
```

Which would find the Sass file we just wrote. That's a lame job, imagine if we had 100 view templates. What if we want to change the name of the stylesheet later? Ugh.

Rails and Ruby both emphasize the idea of "D.R.Y." – Don't Repeat Yourself. In the area of view templates, we can achieve this by creating a *layout*. A layout is a special view template that wraps other views. Rails has given us one already: `app/views/layouts/application.html.erb`.

Check out your `app/views/layouts/application.html.erb`:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title>Blogger</title>
```

```
    <%= stylesheet_link_tag    "application", media: "all", "data-tu
rbolinks-track" => true %>
    <%= javascript_include_tag "application", "data-turbolinks-track
" => true %>
    <%= csrf_meta_tags %>
</head>
<body>

<p class="flash">
    <%= flash.notice %>
</p>
<%= yield %>

</body>
</html>
```

Whatever code is in the individual view template gets inserted into the layout where you see the `yield`. Using layouts makes it easy to add site-wide elements like navigation, sidebars, and so forth.

See the `stylesheet_link_tag` line? It mentions 'application.' That means it should load up `app/assets/stylesheets/application.css`... Check out what's in that file:

```
/*
 * This is a manifest file that'll be compiled into application.cs
s, which will include all the files
 * listed below.
 *
 * Any CSS and SCSS file within this directory, lib/assets/stylesh
eets, vendor/assets/stylesheets,
 * or vendor/assets/stylesheets of plugins, if any, can be referen
ced here using a relative path.
 *
 * You're free to add application-wide styles to this file and the
y'll appear at the top of the
 * compiled file, but it's generally better to create a new file p
er style scope.
 *
 *= require_self
 *= require_tree .
*/
```

There's that huge comment there that explains it: the `require_tree .` line automatically loads all of the stylesheets in the current directory, and includes them in `application.css`. Fun! This feature is called the `asset pipeline`, and it's pretty new to Rails. It's quite powerful.

Now that you've tried out a plugin library (Paperclip), Iteration 4 is complete!

Saving to GitHub.

Terminal

```
$ git add .
$ git commit -m "added a few gems"
$ git push
```

I5: Authentication

Authentication is an important part of almost any web application and there are several approaches to take. Thankfully some of these have been put together in plugins so we don't have to reinvent the wheel.

There are two popular gems for authentication: One is named AuthLogic [https://github.com/binarylogic/authlogic/] and I wrote up an iteration using it for the Merchant [http://tutorials.jumpstartlab.com/projects/merchant.html] tutorial, but I think it is a little complicated for a Rails novice. You have to create several different models, controllers, and views manually. The documentation is kind of confusing, and I don't think my tutorial is that much better. The second is called Devise [https://github.com/plataformatec/devise] , and while it's the gold standard for Rails 3 applications, it is also really complicated.

Sorcery [https://github.com/NoamB/sorcery] is a lightweight and straightforward authentication service gem. It strikes a good balance of functionality and complexity.

Installing Sorcery

Sorcery is just a gem like any other useful package of Ruby code, so to use it in our Blogger application we'll need to add the following line to our Gemfile:

```
1  gem 'sorcery'
```

> NOTE
>
> When specifying and installing a new gem you will need to restart your Rails Server

Then at your terminal, instruct Bundler to install any newly-required gems:

Terminal

```
$ bundle
```

Once you've installed the gem via Bundler, you can test that it's available with this command at your terminal:

Terminal

```
$ rails generate
```

NOTE

If you receive a LoadError like `cannot load such file – bcrypt`, add this to your Gemfile: `gem 'bcrypt-ruby'`, and then run `bundle` again.

Somewhere in the middle of the output you should see the following:

Terminal

```
$ rails generate
  ...
  Sorcery:
  sorcery:install
  ...
```

If it's there, you're ready to go!

Running the Generator

This plugin makes it easy to get up and running by providing a generator that creates a model representing our user and the required data migrations to support authentication. Although Sorcery provides options to support nice features like session-based "remember me", automatic password-reset through email, and authentication against external services such as Twitter, we'll just run the default generator to allow simple login with an email and password.

One small bit of customization we will do is to rename the default model created by Sorcery from "User" to "Author", which gives us a more domain-relevant name to work with. Run this from your terminal:

Terminal

```
$ bin/rails generate sorcery:install --model=Author
```

Take a look at the output and you'll see roughly the following:

Terminal

```
create  config/initializers/sorcery.rb
generate  model Author --skip-migration
invoke  active_record
create    app/models/author.rb
invoke    test_unit
create      test/unit/author_test.rb
create      test/fixtures/authors.yml
insert  app/models/author.rb
create  db/migrate/20120210184116_sorcery_core.rb
```

Let's look at the SorceryCore migration that the generator created before we migrate the database. If you wanted your User models to have any additional information (like "department_name" or "favorite_color") you could add columns for that, or you could create an additional migration at this point to add those fields.

For this tutorial, you will need to add the username column to the Author model. To do that, open the migration file `*_sorcery_core.rb` file under `db/migrate` and add make sure your file looks like this:

```
1  class SorceryCore < ActiveRecord::Migration
2    def change
3      create_table :authors do |t|
4        t.string :username,         :null => false
5        t.string :email,            :null => false
6        t.string :crypted_password, :null => false
7        t.string :salt,             :null => false
8
9        t.timestamps
10     end
11
12     add_index :authors, :email, unique: true
13   end
14 end
```

So go to your terminal and enter:

Terminal

```
$ bin/rake db:migrate
```

Let's see what Sorcery created inside of the file `app/models/author.rb`:

```
1  class Author < ActiveRecord::Base
```

```
2     authenticates_with_sorcery!
3   end
```

We can see it added a declaration of some kind indicating our Author class authenticates via the sorcery gem. We'll come back to this later.

Creating a First Account

First, stop then restart your server to make sure it's picked up the newly generated code.

Though we could certainly drop into the Rails console to create our first user, it will be better to create and test our form-based workflow by creating a user through it.

We don't have any create, read, update, and destroy (CRUD) support for our Author model. We could define them again manually as we did with Article. Instead we are going to rely on the Rails code controller scaffold generator.

Terminal

```
$ bin/rails generate scaffold_controller Author username:st
  ring email:string password:password password_confirmation
  :password
```

Rails has two scaffold generators: **scaffold** and **scaffold_controller**. The **scaffold** generator generates the model, controller and views. The **scaffold_controller** will generate the controller and views. We are generating a **scaffold_controller** instead of **scaffold** because Sorcery has already defined for us an Author model.

As usual, the command will have printed all generated files.

The generator did a good job generating most of our fields correctly, however, it did not know that we want our password field and password confirmation field to use a password text entry. So we need to update the `authors/_form.html.erb`:

```
1  <div class="field">
2    <%= f.label :password %><br />
3    <%= f.password_field :password %>
4  </div>
5  <div class="field">
6    <%= f.label :password_confirmation %><br />
7    <%= f.password_field :password_confirmation %>
8  </div>
```

When we created the controller and the views we provided a `password` field and a `password_confirmation` field. When an author is creating their account we want to ensure that they do not make a mistake when entering their password, so we are requiring that they repeat their password. If the two do not match, we know our record should be invalid, otherwise the user could have mistakenly set their password to something other than what they expected.

To provide this validation when an author submits the form we need to define this relationship within the model.

```
1  class Author < ActiveRecord::Base
2    authenticates_with_sorcery!
3    validates_confirmation_of :password, message: "should match conf
4  irmation", if: :password
   end
```

The `password` and `password_confirmation` fields are sometimes referred to as "virtual attributes" because they are not actually being stored in the database. Instead, Sorcery uses the given password along with the automatically generated `salt` value to create and store the `crypted_password` value.

Visiting http://localhost:3000/authors [http://localhost:3000/authors] at this moment we will find a routing error. The generator did not add a resource for our Authors. We need to update our `routes.rb` file:

```
1  Blogger::Application.routes.draw do
2    # ... other resources we have defined ...
3    resources :authors
4  end
```

With this in place, we can now go to http://localhost:3000/authors/new [http://localhost:3000/authors/new] and we should see the new user form should popup. Let's enter in "admin@example.com [mailto:admin@example.com] " for email, and "password" for the password and password_confirmation fields, then click "Create Author". We should be taken to the show page for our new Author user.

Now it's displaying the password and password_confirmation text here, lets delete that! Edit your `app/views/authors/show.html.erb` page to remove those from the display.

If you click *Back*, you'll see that the `app/views/authors/index.html.erb` page also shows the hash and salt. Edit the file to remove these as well.

We can see that we've created a user record in the system, but we can't really tell if

we're logged in. Sorcery provides a couple of methods for our views that can help us out: `current_user` and `logged_in?`. The `current_user` method will return the currently logged-in user if one exists and `false` otherwise, and `logged_in?` returns `true` if a user is logged in and `false` if not.

Let's open `app/views/layouts/application.html.erb` and add a little footer so the whole `<body>` chunk looks like this:

```
1  <body>
2    <p class="flash">
3      <%= flash.notice %>
4    </p>
5    <div id="container">
6      <div id="content">
7        <%= yield %>
8        <hr>
9        <h6>
10         <% if logged_in? %>
11           <%= "Logged in as #{current_user.email}" %>
12         <% else %>
13           Logged out
14         <% end %>
15       </h6>
16     </div>
17   </div>
18 </body>
```

The go to `http://localhost:3000/articles/` and you should see "Logged out" on the bottom of the page.

Logging In

How do we log in to our Blogger app? We can't yet! We need to build the actual endpoints for logging in and out, which means we need controller actions for them. We'll create an AuthorSessions controller and add in the necessary actions: new, create, and destroy.

First, let's generate the AuthorSessions controller:

Terminal

```
$ bin/rails generate controller AuthorSessions
```

Now we'll add `new`, `create`, and `destroy` methods to `app/controllers/author_sessions_controller.rb`:

```
class AuthorSessionsController < ApplicationController
  def new
  end

  def create
    if login(params[:email], params[:password])
      redirect_back_or_to(articles_path, notice: 'Logged in succes
sfully.')
    else
      flash.now.alert = "Login failed."
      render action: :new
    end
  end

  def destroy
    logout
    redirect_to(:authors, notice: 'Logged out!')
  end
end
```

As is common for Rails apps, the `new` action is responsible for rendering the related form, the `create` action accepts the submission of that form, and the `destroy` action removes a record of the appropriate type. In this case, our records are the Author objects that represent a logged-in user.

Let's create the template for the `new` action that contains the login form, in `app/views/author_sessions/new.html.erb`: (you may have to make the directory)

```
<h1>Login</h1>

<%= form_tag author_sessions_path, method: :post do %>
  <div class="field">
    <%= label_tag :email %>
    <%= text_field_tag :email %>
    <br/>
  </div>
  <div class="field">
    <%= label_tag :password %>
    <%= password_field_tag :password %>
    <br/>
  </div>
  <div class="actions">
    <%= submit_tag "Login" %>
  </div>
<% end %>

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

The `create` action handles the logic for logging in, based on the parameters passed from the rendered form: email and password. If the login is successful, the user is

redirected to the articles index, or if the user had been trying to access a restricted page, back to that page. If the login fails, we'll re-render the login form. The `destroy` action calls the `logout` method provided by Sorcery and then redirects.

Next we need some routes so we can access those actions from our browser. Open up `config/routes.rb` and make sure it includes the following:

```
1  resources :author_sessions, only: [ :new, :create, :destroy ]
2
3  get 'login'  => 'author_sessions#new'
4  get 'logout' => 'author_sessions#destroy'
```

Terminal

```
$ bin/rake routes
  # ... other routes for Articles and Comments ...
  author_sessions POST   /author_sessions(.:format)     aut
  hor_sessions#create
  new_author_session GET    /author_sessions/new(.:format)
  author_sessions#new
  author_session DELETE /author_sessions/:id(.:format) auth
  or_sessions#destroy
  login        /login(.:format)              author_sessio
  ns#new
  logout        /logout(.:format)             author_sessi
  ons#destroy
```

Our Author Sessions are similar to other resources in our system. However, we only want to open a smaller set of actions. An author is able to be presented with a login page (:new), login (:create), and logout (:destroy). It does not make sense for it to provide an index, or edit and update session data.

The last two entries create aliases to our author sessions actions.

Externally we want our authors to visit pages that make the most sense to them:

- http://localhost:3000/login
- http://localhost:3000/logout

Internally we also want to use path and url helpers that make the most sense:

- login_path, login_url
- logout_path, logout_url

Now we can go back to our footer in `app/views/layouts/application.html.erb` and update it to include some links:

```
1  <body>
2    <div id="container">
3      <div id="content">
4        <%= yield %>
5        <hr>
6        <h6>
7          <% if logged_in? %>
8            <%= "Logged in as #{current_user.email}" %>
9            <%= link_to "(logout)", logout_path %>
10         <% else %>
11           <%= link_to "(login)", login_path %>
12         <% end %>
13       </h6>
14     </div>
15   </div>
16 </body>
```

Now we should be able to log in and log out, and see our status reflected in the footer. Let's try this a couple of times to confirm we've made it to this point successfully. (You may need to restart the rails server to successfully log in.)

Securing New Users

It looks like we can create a new user and log in as that user, but I still want to make some more changes. We're just going to use one layer of security for the app – a user who is logged in has access to all the commands and pages, while a user who isn't logged in can only post comments and try to login. But that scheme will breakdown if just anyone can go to this URL and create an account, right?

Let's add in a protection scheme like this to the new users form:

- If there are zero users in the system, let anyone access the form
- If there are more than zero users registered, only users already logged in can access this form

That way when the app is first setup we can create an account, then new users can only be created by a logged in user.

We can create a `before_filter` which will run *before* the `new` and `create` actions of our `authors_controller.rb`. Open that controller and put all this code in:

```
1  before_filter :zero_authors_or_authenticated, only: [:new, :create
2  ]
3
```

```
4   def zero_authors_or_authenticated
5     unless Author.count == 0 || current_user
6       redirect_to root_path
7       return false
8     end
    end
```

The first line declares that we want to run a before filter named `zero_authors_or_authenticated` when either the `new` or `create` methods are accessed. Then we define that filter, checking if there are either zero registered users OR if there is a user already logged in. If neither of those is true, we redirect to the root path (our articles list) and return false. If either one of them is true this filter won't do anything, allowing the requested user registration form to be rendered.

With that in place, try accessing `authors/new` when you're logged in and when you're logged out. If you want to test that it works when no users exist, try this at your console:

IRB

```
2.1.1 :001> Author.destroy_all
```

Then try to reach the registration form and it should work! Create yourself an account if you've destroyed it.

Securing the Rest of the Application

The first thing we need to do is sprinkle `before_filters` on most of our controllers:

- In `authors_controller`, add a before filter to protect the actions besides `new` and `create` like this:
  `before_filter :require_login, except: [:new, :create]`
- In `author_sessions_controller` all the methods need to be accessible to allow login and logout
- In `tags_controller`, we need to prevent unauthenticated users from deleting the tags, so we protect just `destroy`. Since this is only a single action we can use `:only` like this:
  `before_filter :require_login, only: [:destroy]`
- In `comments_controller`, we never implemented `index` and `destroy`, but just in case we do let's allow unauthenticated users to only access `create`:
  `before_filter :require_login, except: [:create]`
- In `articles_controller` authentication should be required for `new`, `create`, `edit`, `update` and `destroy`. Figure out how to write the before filter using either `:only` or `:except`

Now our app is pretty secure, but we should hide all those edit, destroy, and new article links from unauthenticated users.

Open `app/views/articles/show.html.erb` and find the section where we output the "Actions". Wrap that whole section in an `if` clause like this:

```
1  <% if logged_in? %>
2
3  <% end %>
```

Look at the article listing in your browser when you're logged out and make sure those links disappear. Then use the same technique to hide the "Create a New Article" link. Similarly, hide the 'delete' link for the tags index.

Your basic authentication is done, and Iteration 5 is complete!

Extra Credit

We now have the concept of authenticated users, represented by our `Author` class, in our blogging application, and it's authors who are allowed to create and edit articles. What could be done to make the ownership of articles more explicit and secure, and how could we restrict articles to being edited only by their original owner?

Saving to GitHub.

Terminal

```
$ git add .
$ git commit -m "Sorcery authentication complete"
$ git push
```

That is the last commit for this project! If you would like to review your previous commits, you can do so with the git log command in terminal:

Terminal

```
$ git log
  commit 0be8c0f7dc92322dd31f579d9a91ebc8e0fac443
  Author: your_name your_email
  Date:   Thu Apr 11 17:31:37 2013 -0600
  Sorcery authentication complete
  and so on...
```

I6: Extras

Here are some ideas for extension exercises:

- Add a site-wide sidebar that holds navigation links
- Create date-based navigation links. For instance, there would be a list of links with the names of the months and when you click on the month it shows you all the articles published in that month.
- Track the number of times an article has been viewed. Add a `view_count` column to the article, then in the `show` method of `articles_controller.rb` just increment that counter. Or, better yet, add a method in the `article.rb` model that increments the counter and call that method from the controller.
- Once you are tracking views, create a list of the three "most popular" articles
- Create a simple RSS feed for articles using the `respond_to` method and XML view templates